O'REILLY®

Fourth Edition

Head First C#

A Learner's Guide to Real-World Programming with C# and .NET Core

Andrew Stellman
& Jennifer Greene

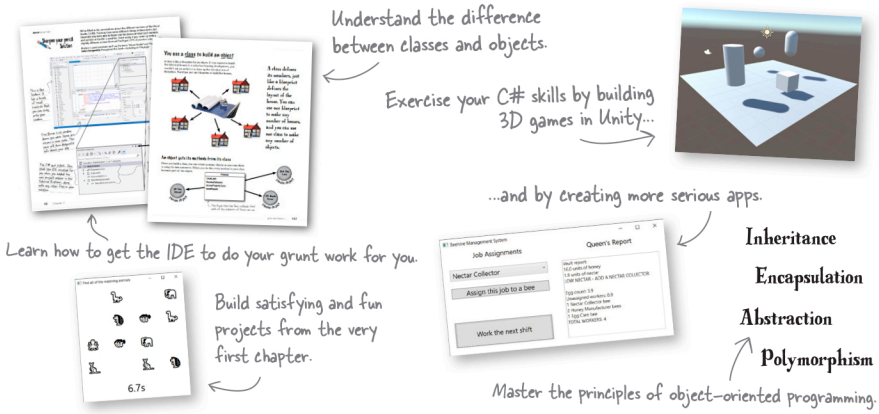
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then you'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



"Thank you so much! Your books have helped me to launch my career."

—Ryan White
Game Developer

"Andrew and Jennifer have written a concise, authoritative, and most of all, fun introduction to C# development."

—Jon Galloway
Senior Program Manager on the .NET Community Team at Microsoft

"If you want to learn C# in depth and have fun doing it, this is THE book for you."

—Andy Parker
Fledgling C# programmer

What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

.NET

US $64.99 **CAN $85.99**

ISBN: 978-1-491-97670-8

9 781491 976708 56499

O'REILLY®

Head First C#

Fourth Edition

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

Copyright © 2021 Jennifer Greene, Andrew Stellman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra, Bert Bates
Cover Designer:	Ellie Volckhausen
Brain Image on Spine:	Eric Freeman
Editors:	Nicole Taché, Amanda Quinn
Proofreader:	Rachel Head
Indexer:	Potomac Indexing, LLC
Illustrator:	Jose Marzan
Page Viewers:	Greta the miniature bull terrier and Samosa the Pomeranian

Printing History:

November 2007: First Edition
May 2010: Second Edition
August 2013: Third Edition
December 2020: Fourth Edition

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, Windows, Visual Studio, MSDN, the .NET logo, Visual Basic, and Visual C# are registered trademarks of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI] [2020-12-18]

Bonus Chapter architecting apps with the mvvm pattern

Great apps on the inside and outside

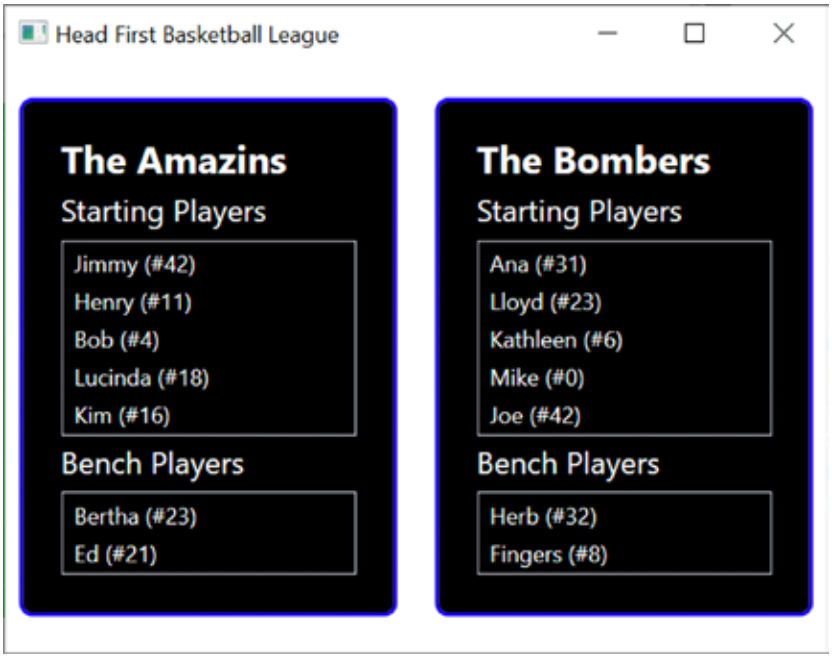
Your apps need to be more than just visually stunning.

When you think of *design*, what comes to mind? An example of great building architecture? A beautifully-laid-out page? A product that's as aesthetically pleasing as it is well-engineered? Those same ideas behind great visual and aesthetic design apply to your code—and we're not just talking about the way your program looks when it runs, but *the way that the code itself is designed*. In this chapter, you'll learn about **the Model-View-ViewModel design pattern** and how you can use it to build well-architected, loosely coupled apps. Along the way, you'll learn about what it means to keep track of your application's **state**, and you'll learn how the MVVM can help you build code that's ***easier to maintain***—even if you haven't looked at it in a year.

The Head First Basketball League needs an app

Jimmy and Ana are the captains of the two top teams in the Head First Basketball League, Objectville's amateur basketball league. They've got some great players, and those players deserve a great app to keep track of who's starting and who's on the bench.

Each team has starting players and bench players, and each player has a name and a number.



There are four different ListBox controls in this window that list the starting and bench players for each team. They will use their ItemsSource properties to bind to ObservableCollection objects. Flip back to the downloadable "Two Decks" project from Chapter 8 if you need a refresher on how ListBox controls and ObservableCollection objects work.

But can they agree on how to build it?

Uh oh—Ana and Jimmy have different ideas about how to build this app, and the argument's starting to get a little heated. It sounds like Ana really wants it to be easy to manage the data that's displayed on the page, while Jimmy cares a lot about simplifying the data binding. This may make for a great off-court rivalry, but it's not going to make it any easier to build the app!

Jimmy: Hold on there, cowboy. Sounds a little short-sighted.

Ana: I'm sure you just don't understand what I'm telling you, so I'll talk real slow and spell it out for you. We'll start with a simple `Player` class that has properties for the name, number, and whether the player's a starter.

Jimmy: Yes, I *understand* what you're saying. But you're not listening to me. You're thinking about how to model the data.

Ana: Clearly. That's where everything starts.

Jimmy: That makes it convenient to create the data.

Ana: You're getting it—

Jimmy: I'm not done. What about the rest of the app? We've got ListBox and TextBlock controls that need to display the data. If we don't have collections for the controls to bind to, they won't work.

Ana: Um...

Jimmy: Exactly. So we may need to make a couple of, ah, tactical decisions in our object model.

Ana: You mean, we need to compromise by creating a lousy object model that's hard to work with, because we need something to bind to.

Jimmy: Unless you've got a better idea.

BRAIN POWER

How can you create classes that are easy to bind to, but still have an object model that makes it easy to work with the data?

Do you design for binding or for working with data?

You already know how important it is to build an object model that makes your data easy to work with. But what if you need to do **two different things** with those objects? That's one of the most common problems that you face as an app designer. Your objects need to have public properties and `ObservableCollections` to bind to your XAML controls. But sometimes that makes your data harder to work with, because it forces you to build an unintuitive object model that's difficult to work with.

Player
Name: string Number: int Starter: bool

Roster
TeamName: string Players: IEnumerable<Player>

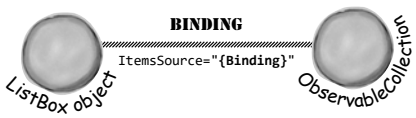
It's hard to optimize your classes to make it easy to slice and dice your data with LINQ queries...

```
var startingPlayers = _roster.Players
    .Where(player => player.Starter)
    .Select(player =>
            new PlayerViewModel(player.Name, player.Number));
```

If you model your data like this, it limits your ability to build the pages that you want.

If you model your data like this, it's easier to build your pages but harder to write code to query and manage the data.

...if you also need to be able to bind that data to the XAML controls on your app's pages.



Player
Name: string Number: int

Roster
TeamName: string Starters: ObservableCollection Bench: ObservableCollection

League
JimmysTeam: Roster AnasTeam: Roster
It would be really convenient to have private methods here to create data.

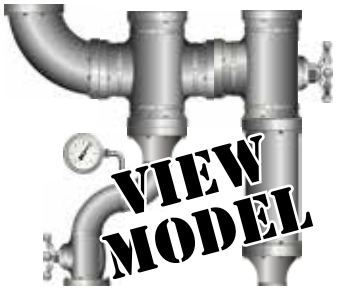
MVVM lets you design for binding and data

Almost all apps that have a large or complex enough object model face the problem of having to either compromise the class design or compromise the objects available for binding. Luckily, there's a design pattern that app developers use to solve this problem. It's called **Model-View-ViewModel** (or **MVVM**), and it works by splitting your app into three layers: the ***model*** to hold the data and state of the app, the ***view*** that contains the pages and controls that the user interacts with, and the ***viewmodel*** that converts the data in the model into objects that can be bound and listens for events in the view that the model needs to know about.

MVVM is a pattern that uses the existing tools you already have, just like the callback and Observer patterns in the last chapter.

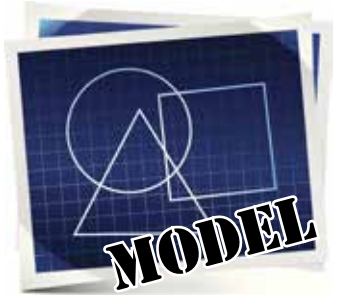
Any object that the user directly interacts with goes in the view.

That includes pages, buttons, text, grids, StackPanels, ListBoxes, and any other controls that can be laid out using XAML. The controls are bound to objects in the viewmodel, and the controls' event handlers call methods in the viewmodel objects.



The viewmodel has the properties that can bind to the controls in the view.

The properties in the view get their data from the objects in the model, convert that data into a form that the view's controls can understand, and notify the view when the data changes.



All of the objects that hold the state of the app live in the model.

This is where your app keeps its data. The viewmodel calls the properties and methods in the model. If there are objects that change throughout the app's lifetime, or if data needs to be saved or loaded from files, those things go here.

The viewmodel is like the plumbing that connects the objects in the view to the objects in the model, using tools you already know how to work with.

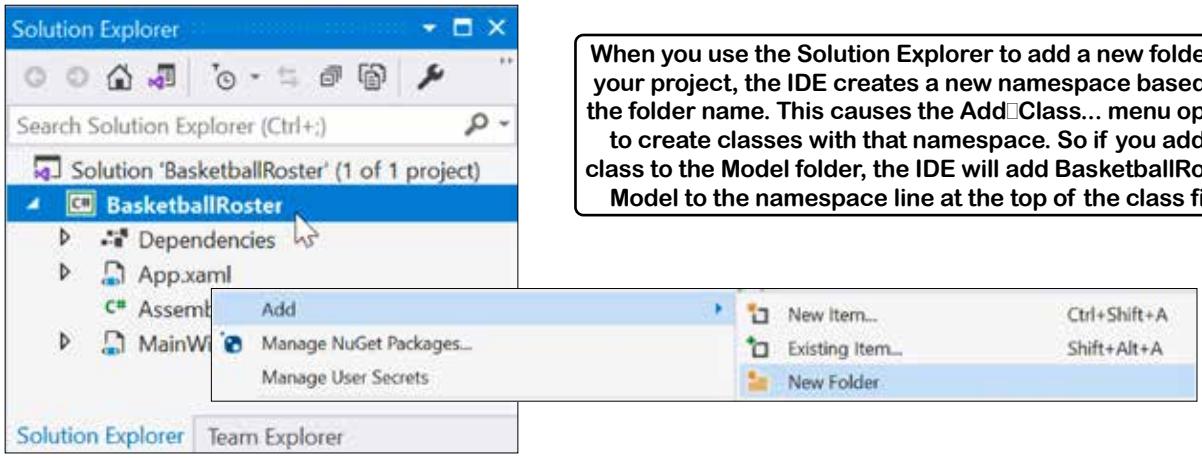
Use the MVVM pattern to start building the basketball roster app

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).

Do this

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:

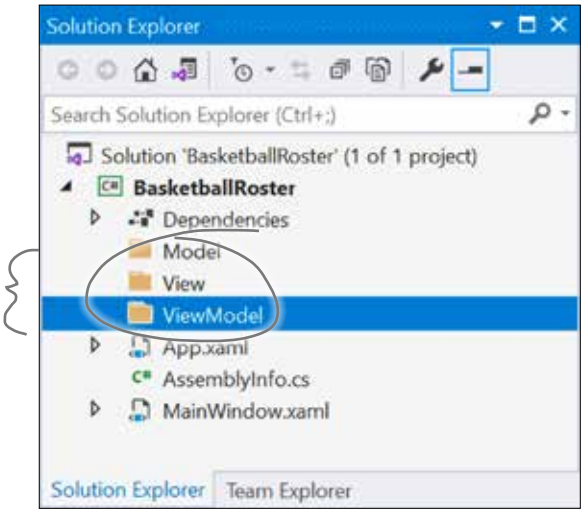


When you use the Solution Explorer to add a new folder to your project, the IDE creates a new namespace based on the folder name. This causes the Add Class... menu option to create classes with that namespace. So if you add a class to the Model folder, the IDE will add BasketballRoster.Model to the namespace 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...

Player
Name: string
Number: int
Starter: bool

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

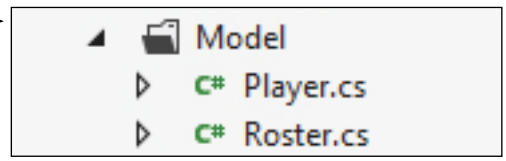
Next, **add the Roster class to the *Model* folder**. Here's the code for it.

```
namespace BasketballRoster.Model {
    class Roster {
        public string TeamName { get; private set; }

        private readonly List<Player> _players = new List<Player>();
        public IEnumerable<Player> Players {
            get => new List<Player>(_players);
        }

        public Roster(string teamName, IEnumerable<Player> players) {
            TeamName = teamName;
            _players.AddRange(players);
        }
    }
}
```

The _ tells you that this field is private.

Roster
TeamName: string
Players: IEnumerable<string>

Your *Model* folder should now look like this:

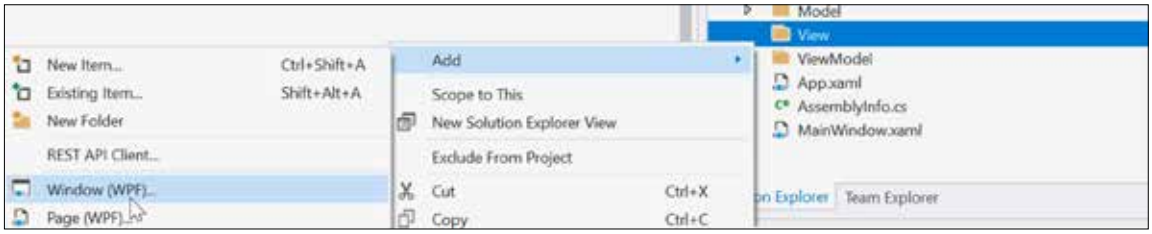


We added an underscore to the beginning of the name of the _players field. Adding an underscore to the beginning of private fields is a very common naming convention. We're going to use it throughout this chapter so you can get used to seeing it.

We'll add the view on the next page →

4 Add a new main window to the View folder.

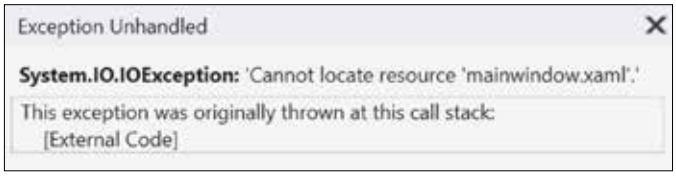
Right-click on the *View* folder and **add a new Window called *LeagueWindow***.



Your project's View folder should now have a XAML file 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 defined with XAML. The only difference is that it's called LeagueWindow instead of MainWindow, and the `.xaml` and `.xaml.cs` files live inside the View folder instead of the root folder of the project.

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:



When you delete MainWindow.xaml, the IDE will also delete MainWindow.xaml.cs.

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 in *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"
             xmlns:local="clr-namespace:BasketballRoster"
             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"
             xmlns:local="clr-namespace:BasketballRoster"
             StartupUri="View/LeagueWindow.xaml">
```

Once you make that change, rebuild and rerun your app. Now it should start and show an empty 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 ListBox, another TextBlock, and another ListBox, 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 lets you define your own controls to use in your XAML apps. You design your code with XAML and code-behind, just like a window.

1 Add a new user control to your View folder.

Right-click on the *View* folder and add a new item. **Choose User Control** 

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>`. It looks just like the XAML for a window, but instead of defining the layout of a window, it **defines the layout of a control**. The control that you're designing works just like any other control you've used. Once you're done with it, you'll be able to use it just like you use a TextBlock, Button, or Listbox.

```
<UserControl x:Class="BasketballRoster.View.UserControl1"
             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"
             xmlns:local="clr-namespace:BasketballRoster.View"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
    <Grid>

    </Grid>
</UserControl>
```

You'll be replacing this Grid with XAML code to display your basketball roster.

A UserControl has DesignHeight and DesignWidth tags where a Window would normally have Title, Height, and Width. These properties determine how the control is displayed in the designer. The control's actual width and height at runtime are set by its layout properties.

4 Finish the RosterControl XAML.

Here's the code for the `RosterControl` user control that you added to the *View* folder. Your RosterControl will a new **Border** control that you haven't seen yet. This line of XAML code creates a container with a 2-pixel blue border with curved corners and a black background:

```
<Border BorderThickness="2" BorderBrush="Blue" CornerRadius="6" Background="Black">
```

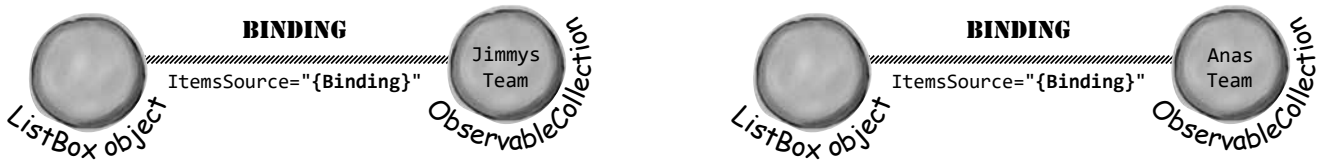
The Border control can contain one other control—in this case, a vertical StackPanel that contains TextBLock and ListBox controls. Here's the XAML for the UserControl:

```
<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"
             xmlns:local="clr-namespace:BasketballRoster.View"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="300">

    <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"/>
            <ListBox Background="Black" Foreground="White"  Margin="0,5,0,0"
                                        ItemsSource="{Binding Starters}" />
            <TextBlock Foreground="White" FontFamily="Segoe" FontSize="16px"
                       Text="Bench Players" Margin="0,5,0,0"/>
            <ListBox Background="Black" Foreground="White"
                       ItemsSource="{Binding Bench}" Margin="0,5,0,0"/>
        </StackPanel>
    </Border>
</UserControl>
```

This user control uses familiar XAML controls and data binding to display a list of players. The ListBox controls work just like the ones in your "Two Decks" app in Chapter 8.

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 window will set different data contexts for each of them:

```
<view:RosterControl Width="200" DataContext="{Binding JimmysTeam}" Margin="0,0,20,0" />
<view:RosterControl Width="200" DataContext="{Binding AnasTeam}" />
```



Each user control has its own instance of the viewmodel object, which it keeps 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.

Set the window's title to **`Head First Basketball League`**, its height to **`350`**, and its width to **`450`**. Next, **create an empty class called LeagueViewModel *in the ViewModel folder***.

Then add an `xmlns:viewmodel` propery to *LeagueWindow.xaml* to use the viewmodel objects:

```
xmlns:local="clr-namespace:BasketballRoster.View"
xmlns:viewmodel="clr-namespace:BasketballRoster.ViewModel"
```

The `xmlns:` properties are like `using` directives for your XAML code. When you created the user control, it automatically added `xmlns:local` to access the objects in the namespace it lives in. You'll add xmlns:viewmodel to access the objects in the ViewModel folder.

Run your app to make sure it still works (and to rebuild it). Then add an instance of `LeagueViewModel` as a static resource:

```
<Window.Resources>
    <viewmodel:LeagueViewModel x:Key="LeagueViewModel"/>
</Window.Resources>
```

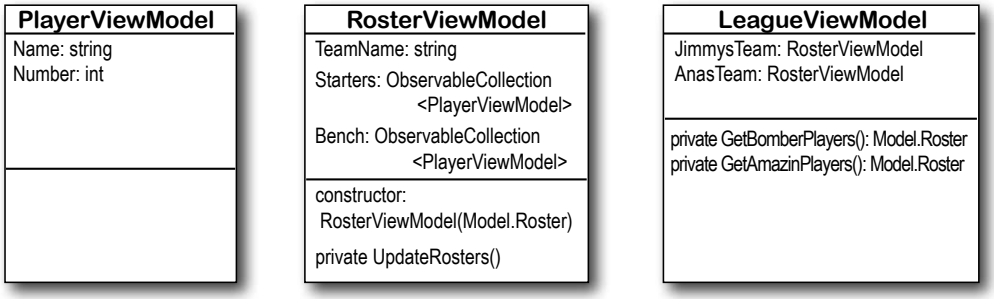
Now you can **replace the grid with a StackPanel** that contains two `RosterControl` controls:

```
<StackPanel Orientation="Horizontal" Margin="5"
            VerticalAlignment="Center" HorizontalAlignment="Center"
            DataContext="{StaticResource ResourceKey=LeagueViewModel}" >
    <local:RosterControl Width="200" DataContext="{Binding JimmysTeam}" Margin="0,0,20,0" />
    <local:RosterControl Width="200" DataContext="{Binding AnasTeam}" />
</StackPanel>
```

2 Create the viewmodel classes.

Make sure you created the classes and pages in the right folders. Otherwise, the namespaces won't match the code in the solution.

Create these three classes in the *ViewModel* folder.



You already created the LeagueViewModel class in Step 1.

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 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.

You learned all about LINQ in Chapter 9.

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, right-click on the BasketballRoster project and choose 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 AnasTeam { get; set; }
        public RosterViewModel JimmysTeam { get; set; }

        public LeagueViewModel() {
            var anasRoster = new Roster("The Bombers", GetBomberPlayers());
            AnasTeam = new RosterViewModel(anasRoster);

            var jimmysRoster = new Roster("The Amazins", GetAmazinPlayers());
            JimmysTeam = new RosterViewModel(jimmysRoster);
        }

        private IEnumerable<Player> GetBomberPlayers() {
            return new List<Player>() {
                new Player("Ana", 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),
            };
        }

        private IEnumerable<Player> GetAmazinPlayers() {
            return 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),
            };
        }
    }
}
```

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 data for the Bombers by creating a new List of Player objects.

Hard-coded data—or data that's explicitly added in the code—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.

```
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;
        }

        public override string ToString() => $"{Name} (#{Number})";
    }
}
```

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, right-click on the BasketballRoster project and choose 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.

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.Linq;

    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 = _roster.Players
                .Where(player => player.Starter)
                .Select(player => new PlayerViewModel(player.Name, player.Number));

            foreach (var playerViewModel in startingPlayers)
                Starters.Add(playerViewModel);

            var benchPlayers = _roster.Players
                .Where(player => !player.Starter)
                .Select(player => new PlayerViewModel(player.Name, player.Number));

            foreach (var playerViewModel in benchPlayers)
                Bench.Add(playerViewModel);
        }
    }
}
```

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 creates PlayerViewModel objects to add to the Starters collection.

Here's a similar LINQ query to find the bench players.

In a typical MVVM WPF 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 Ana and Jimmy change their team names, you'd need to fire a PropertyChanged event in the TeamName set accessor.

ISN'T A USER CONTROL BASICALLY JUST A WAY TO SPLIT YOUR XAML ACROSS A FEW FILES?

User controls are fully functional controls that you build.

And like every other control, a user control is an object—in this case, an object that extends the UserControl base class, which gives you familiar properties like Height and Visibility, and routed events like Tapped and PointerEntered. You can also add your own properties, and you can use the other XAML controls to make very intricate, even visually stunning user interfaces. But most importantly, a user control lets you **encapsulate** those other controls into a single XAML control that you can reuse.

WAIT A MINUTE...THIS STUFF ABOUT ENCAPSULATION AND SEPARATING OBJECTS INTO LAYERS SOUNDS REALLY FAMILIAR. DOES THIS HAVE SOMETHING TO DO WITH ***SEPARATION OF CONCERNS***?

That's right! The model, view, and viewmodel divide up the concerns of the program.

One of the most challenging parts of designing a large, robust app is choosing which objects do what. There's an almost infinite number of ways to design your app. That's great, because it means that C# gives you flexible tools to work with. But it's also a challenge, because today's decisions can make tomorrow's changes very difficult to manage. MVVM helps you separate the concerns about the data in your app from the concerns about its UI. This makes it easier to design your app by helping you figure out exactly where data goes and where UI elements go, and by giving you patterns to help connect them together.

When a change to one class requires changes to two more, which then require more changes to additional classes, there's a name for that. Programmers call it "shotgun surgery," and it's very frustrating—especially when you're in a hurry.

Separation of concerns is a great way to prevent problems like that, and MVVM is a very useful tool to help you separate some important things that almost every app is concerned with.

there are no Dumb Questions

Q: So what's stopping me from putting controls in the viewmodel or `ObservableCollections` in the model?

A: Nothing at all—except that once you do, you're no longer really separating concerns. When you mix up objects that have to do with the way your app displays its UI to the user with parts that manage its state, you're no longer separating those different concerns, and that defeats the purpose of the MVVM design pattern. Classes like windows, buttons, user controls, and observable collections are concerned with creating the user interface that's displayed to the user. If you put them in the view, that makes it easier for you to manage your codebase as your app grows larger. When you trust the MVVM pattern today, your life is better tomorrow because your code is easier to manage.

Q: I still don't get what *state* means.

A: When people talk about **state** they mean the objects in memory that determine what your app "knows": the text being edited by a text editor, the location of the enemies and player and the score in a video game, the values of the cells in a spreadsheet, the amount of honey in the vault and the bees in the hive in the Beehive Management System that you built in Chapter 6. It's okay if the concept of state seems weird—it may take some time to wrap your brain around it, because it's sometimes difficult to say "this object is part of the state" and "that object isn't." One of the goals of the next project in this chapter is to help you get a practical, realistic handle on what *state* really means.

Q: Why do I need `using Model;` at the top of my viewmodel classes?

A: When you created classes in the *Model* folder, the IDE automatically created them in the `BasketballRoster.Model` namespace. The dot in the middle of that namespace means that Model is underneath BasketballRoster. Any other class in a namespace under BasketballRoster can access classes in Model by either adding **`Model.`** to the beginning or adding a using line. Outside the `BasketballRoster` namespace, classes will need to add `using BasketballRoster.Model;` instead.

Q: I keep seeing a triangle with an exclamation point on my page. What's that about?

A: The IDE's XAML designer is a pretty sophisticated piece of machinery. It works so well that we sometimes forget just how much work it has to do to display a page and update it as we modify the XAML. Now that the BasketballRoster program is finished, the designer shows you the data for both teams. But wait a minute—isn't that data created in private methods in the viewmodel? That means the designer must be running those methods every time it updates the page. So for it to be able to run properly, those methods have to be compiled. If you modify the controls that are on the page, then the latest C# code hasn't been compiled yet, so the designer is telling you that the page that it's displaying may be out of date. Rebuild the code and the exclamation points usually disappear.

Q: The BasketballRoster app I just built only has data that's created when it starts up. What if I want to add a feature to modify the data in the model—how would that work?

A: Let's say you wanted to modify your BasketballRoster program to let Jimmy and Ana trade players. You already know that the ListBox controls in the view are bound to ObservableCollection objects, so the viewmodel communicates with the view using PropertyChanged and CollectionChanged events. And you can have the model communicate with the viewmodel in exactly the same way. You could add an event, RosterUpdated, to the Roster object. The RosterViewModel would listen to that event, and its event handler would refresh the Starters and Bench collections, which would then fire off CollectionChanged events, which would update the ListBox controls.

Events are a good way for the model to communicate to the rest of the app because the model doesn't need to know if any other classes are listening to the event. It can go about its business managing the state, and let some other class worry about getting input and updating the user interface because it's ***decoupled*** from the classes in the viewmodel and the view.

When you trust the MVVM pattern today, your project will be better tomorrow, because your app's code will be easier to manage.

Tonight's talk: **A model and a viewmodel have a heated debate over the critical issue of the day, "Who's needed more?"**

Model:

I'm not quite sure why we're even having this discussion. Where would you be without me? I've got the data; I've got the important logic that determines how the app works. Without me, you'd have nothing to do.

Well, as far as you're concerned, I may as well be.

You wouldn't dare.

Now you know why I only speak to you through events. You're just so annoying!

Of course I do! If I didn't encapsulate my data, who knows what damage you might cause?

Absolutely! I don't trust anything except my own private methods to manage my data; otherwise, the whole state of our app could go haywire. But I'm not the only one who plays this game! Why don't you ever let me talk to the view? He seems like a good guy.

How dare you! Raise `PropertyChanged` events? No self-respecting Model has ever raised a `PropertyChanged` event! I'm insulted you'd even suggest I'm concerned with anything but data. What kind of layer do you think I am?

Viewmodel:

There you go again, thinking that you're the center of the universe.

Ha! What would happen if I decided to stay home?

Try me! Without me, you'd be useless. The view would have no idea how to talk to you. The controls would be empty, and the user would be left in the dark.

You know what? Let's talk about that for a minute. Why is it that you can't even let me see your internals? You only expose methods and properties to me, and you'll only ever send me messages through event arguments.

It sounds like ***someone*** has trust issues.

You barely even speak the same language as the view! I've never seen you fire a `PropertyChanged` event—in fact, I don't think any of your objects even implement `INotifyPropertyChanged`.

The ref needs a stopwatch

Jimmy and Ana had to call off their last game because the referee forgot his stopwatch. Can we use the MVVM pattern to build a stopwatch app for them?

How's the ref going to enforce the three-second rule without it?

In this project, we'll take a deep dive into MVVM: what it means to store state, how the viewmodel bridges the gap between the model and the view, and how the MVVM pattern can help you build more maintainable code. We'll do this by first creating a .NET Core Console app that follows the pattern, and then adding a WPF app that uses the same viewmodel.

MVVM means thinking about the state of the app

MVVM apps use the model and view to separate the state from the user interface. So when you start building an MVVM app, the first thing you usually do is think about exactly what it means to manage the state of the app. Once you've got the state under control in your brain, you can start building the model, which will use fields and properties to keep track of the state—or everything the app needs to keep track of to do its job. Most apps need to modify the state as well, so the model exposes public methods that change the state. The rest of the app needs to be able to see the current state, so the model provides public properties.

So what does it mean to manage the state of a stopwatch?

The stopwatch knows whether or not it's running.

You can see at a glance whether or not the hands are moving, so the stopwatch Model needs to have a way to tell whether or not it's running.

The elapsed time is always available.

Whether it's the hands on an analog stopwatch or numbers on a digital one, you can always see the elapsed time.

The lap time can be set and viewed.

Most stopwatches have a lap time function that lets you save the current time without stopping the clock. Analog stopwatches use an extra set of hands to show the lap time, while digital stopwatches usually have a separate lap time readout.

The stopwatch can stop, start, and reset.

The app will need to provide a way to start the stopwatch, stop it, and reset the time, which means the model will need to give the rest of the app a way to do this.

The Model keeps track of the state of the app: what the app knows right now. It provides actions that modify the app's state and properties to let the rest of the app see the current state.

Your stopwatch app's state up close

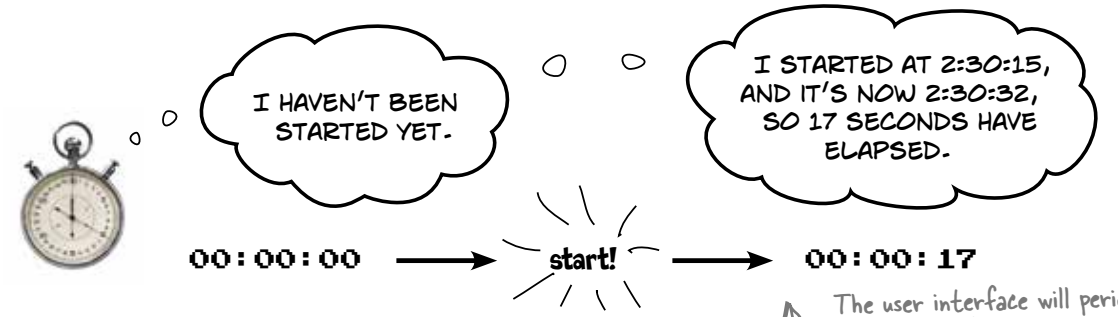
What does it really mean to store the state of a stopwatch?

The state of the stopwatch is all of the information it "knows" at any particular moment in time. Let's take a closer look at how you'll keep track of the state in your stopwatch app.

Part 1: The stopwatch knows if it's running and when it was started

When you pick up a stopwatch it shows zero until you start it. When you do, it starts counting up. You'll break this project down into three parts, and in the first part your stopwatch will either display 00:00:0.0 or it the elapsed time What information does the stopwatch model need to keep track of so it can give the viewmodel and view to do that?

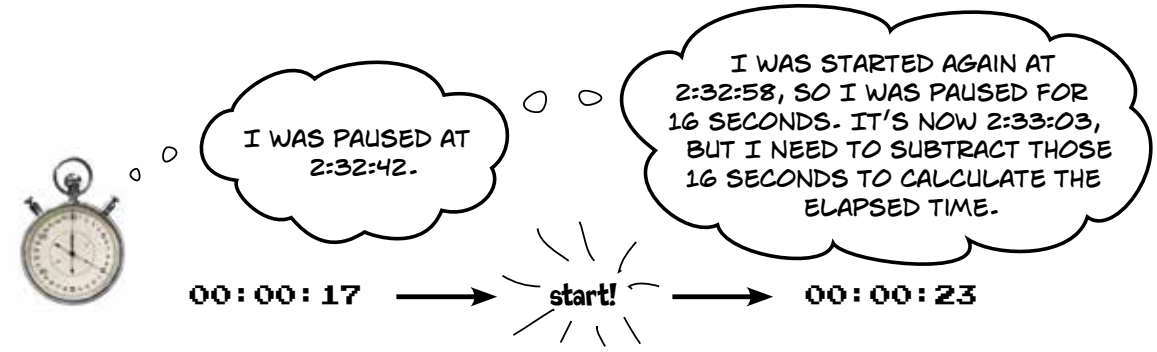
In your stopwatch model, all you need to keep track of is whether the stopwatch is running, and if it is, when it started. Then you'll use that information to calculate the elapsed time:



The user interface will periodically refresh itself and check the state to display how much time has elapsed.

Part 2: The stopwatch also knows when it's paused

In the second part of the project, you'll let your user pause the stopwatch. To do this, the model will need to keep track of whether or not it's paused. When the user pauses the stopwatch, the model will keep track of when it was paused. When the user unpauses, the model will figure out the total timespan that it was paused and subtract it from the elapsed time.



Part 3: The stopwatch keeps track of the lap time

In the third part of the project, you'll let the user tell the stopwatch to save the current lap time. The model will save the elapsed time at the moment the user asks for it, so that becomes part of the state, too.

You'll use a few new tools for this project

You learned about structs in Chapter 11. In this project, we'll use two really useful .NET structs: **TimeSpan and DateTime**. TimeSpan keeps track of a length of time, and DateTime keeps track of a specific date and time.

You've been creating .NET Core console apps throughout the book, using the Console class to write text to the console and read user input. You used its Write and WriteLine methods to make your apps write output, and its ReadLine and ReadKey methods to get input from your user. Now you'll take your console apps to the next level, using methods and properties on the Console class to **set the cursor position and make the cursor invisible**. You'll also make combine the ReadLine method with the **KeyAvailable property**, which returns true if the user has pressed a key that's now available. That will help make your console app respond to user presses in real time, without pausing for input.

You've been using the Console class to read and write characters throughout the book. Now you'll take control of your console so you can make your stopwatch fancy! The static Clear method clears the console, and its CursorTop and CursorLeft properties move the cursor so you can write to a specific row and column (the numbers start at 0). You'll use the static CursorVisible property to turn off the cursor while it's running so it doesn't get blinky. Your stopwatch will need to read characters from the console without pausing. You'll use the KeyAvailable property, which returns true if the user hit a key and it's waiting to be read by Console.ReadKey.

TimeSpan and DateTime structs

There are two very useful structs for managing time in an app, and you'll use them both in this project. DateTime stores a date and time. TimeSpan represents an interval of time. The interval is stored in ticks (a tick is one ten-millionth of a second, or 10,000 ticks per millisecond), The TimeSpan has properties to break down the span of time that it's holding into its hours, minutes, seconds, or milliseconds.

The TimeSpan and DateTime structs have some useful members that you'll use in your project:

- DateTime has a static **MinValue** property that represents the earliest date that it can hold.
- The static **DateTime.Now** returns property the current date and time.
- Use + and - operators to add and subtract DateTime and TimeSpan values. Subtracting one DateTime from another and returns a TimeSpan. Subtracting a TimeSpan from a DateTime returns a DateTime.
- TimeSpan has an int property **Hours** that returns the number of hours in the span. It also has similar **Minutes**, **Seconds**, and **Milliseconds** properties.
- The static **TimeSpan.Zero** property returns a zero-length time span.
- TimeSpan and DateTime both have a **Compare** method that takes two values, returning -1 if the first value is less than the second, 1 if the first value is greater, and 0 if they're equal.

This project uses the System.Threading.Timer—it's a different kind of timer than the ones used earlier in the book. The class uses a callback delegate. If you haven't learned about delegates yet, download the free PDF of the *Events and Delegates* chapter from our GitHub page.

Create the Stopwatch project and add the model

We'll build this project in parts. In this first part, we'll just create a simple stopwatch that starts, resets, and shows the elapsed time. **Create a new Console App project called *Stopwatch***. Then create the View, ViewModel, and Model folders, just like you did with the basketball roster app.

Add this StopwatchModel class to the Model folder:

When you create the class in the Model folder, the IDE adds it to the Stopwatch.Model namespace.

```
namespace Stopwatch.Model
{
    class StopwatchModel
    {
        private DateTime _startedTime;

        /// <summary>
        /// The constructor resets the stopwatch
        /// </summary>
        public StopwatchModel() => Reset();

        /// <summary>
        /// Returns true if the stopwatch is running
        /// </summary>
        public bool Running
        {
            get => _startedTime != DateTime.MinValue;
            set
            {
                if (value && !Running) _startedTime = DateTime.Now;
            }
        }

        /// <summary>
        /// Returns the elapsed time, or zero if the stopwatch is not running
        /// </summary>
        public TimeSpan Elapsed => Running ? DateTime.Now - _startedTime : TimeSpan.Zero;

        /// <summary>
        /// Resets the stopwatch by setting its started time to DateTime.MinValue
        /// </summary>
        public void Reset() => _startedTime = DateTime.MinValue;
    }
}
```

The model knows the stopwatch is reset when its time is equal DateTime.MinValue, a special value with the smallest valid time. If _startedTime is equal to that time, then the stopwatch isn't running. If not, that's when the stopwatch was started. It can calculate the elapsed time by subtracting _startedTime from the current date and time.

The Running property compares _startTime to DateTime.MinValue to see if the stopwatch is currently running.. It sets _startedTime to the current time to start the stopwatch – but only if it's not running yet.

The Elapsed property returns the elapsed time by subtracting the time the stopwatch started from the current time

To reset the stopwatch and stop it running, we set its started time to DateTime.MinValue.

The stopwatch model keeps track of the time it started. Its Elapsed property calculates the elapsed time by subtracting the time it started from the current time.

Add skeletons for view and viewmodel to your stopwatch

Next, you'll create the view. It uses methods and properties from the viewmodel, which you'll build next in an exercise. So start by **adding this skeleton of the StopwatchViewModel class in the ViewModel folder**:

```
class StopwatchViewModel
{
    public void StartStop() => throw new NotImplementedException();

    public void Reset() => throw new NotImplementedException();

    public string Hours => throw new NotImplementedException();

    public string Minutes => throw new NotImplementedException();

    public string Seconds => throw new NotImplementedException();

    public object Tenths => throw new NotImplementedException();
}
```

For now the StartStop method will just stop the stopwatch. In the next part we'll make it pause and unpause the stopwatch too.

Now **add this *partially completed* StopwatchView class in the View folder**:

```
using System.Threading;
using ViewModel;

class StopwatchView
{
    private StopwatchViewModel _viewModel = new StopwatchViewModel();
    private bool _quit = false;

    /// <summary>
    /// Clears the console and displays the stopwatch
    /// </summary>
    public StopwatchView()
    {
        ClearScreenAndAddHelpMessage();

        TimerCallback timerCallback = UpdateTimeCallback;
        var _timer = new Timer(timerCallback, null, 0, 10);
        while (!_quit)
            Thread.Sleep(100);

        Console.CursorVisible = true;
    }

    /// <summary>
    /// Clears the screen, adds the help message to fourth row, and makes the cursor invisible
    /// </summary>
    private static void ClearScreenAndAddHelpMessage()
    {
        Console.Clear();
        Console.CursorTop = 3; // This moves the cursor to the fourth row (rows start at 0)
        Console.WriteLine("Space to start, R to reset, any other key to quit");
        Console.CursorVisible = false;
    }

    /// <summary>
    /// Callback to update the time dispay that the time calls each time it ticks
    /// </summary>
    private void UpdateTimeCallback(object? state) => throw new NotImplementedException();
}
```

The view uses a System.Threading.Timer. It's a different kind of timer than the one you used in Chapter 1. That timer raised an event every time it ticked. The timer you'll use now is different—it uses a callback. Its constructor takes a TimerCallback delegate with a reference to the method that it will call each time it ticks. After the constructor starts the timer, it uses a loop to wait until the user quits. Thread.Sleep keeps the app from using CPU cycles while it's waiting for the next timer tick.

You've started building the first part of the MVVM Stopwatch project by adding the model, a skeleton of the viewmodel, and a partially completed view. Now it's time to get this first part up and running by finishing the viewmodel and view. **Update your Main method to instantiate StopwatchView**. Then:

Implement the StopwatchViewModel class.

We gave you a skeleton for the StopwatchViewModel class—all of its methods and properties throw an exception. Your job is to replace these stub members with fully implemented methods and properties:

- You'll need to create an instance of Model and store it in a private readonly field called `_model`.
- The Start method starts the stopwatch, but only if it's not already running.
- The Reset method resets the stopwatch.
- Here's the Hours property: `public string Hours => _model.Elapsed.Hours.ToString("D2");` It gets the elapsed time from the model as a TimeSpan, and uses its Hours property to get the number of hours. It calls the ToString method with a D2 format specifier to pad it to two digits (so 1 is displayed as `01`).
- Fill in the rest of the properties so they return the current minutes, seconds, and milliseconds from the model.

Add the StopwatchView.WriteCurrentTime method.

Your console stopwatch will be ~~fancy~~! Instead of just printing lines to the console and seeing them scroll off the bottom of the page, you'll use the Console class to show your running stopwatch on the second row of the screen. Add this WriteCurrentTime method to your view class:

```
/// <summary>
/// Writes the current time to the second row and 23rd column of the screen
/// </summary>
private void WriteCurrentTime()
{
    Console.CursorTop = 1;    // This moves the cursor to the second row (rows start at 0)
    Console.CursorLeft = 23;  // This moves the cursor to the 23rd column (starting at 0)
    var time = new NotImplementedException();
    Console.Write(time);
}
```

You'll need to replace this with something.

You'll need to **replace the third line** of the method so it creates a string that has the current elapsed time. You'll do that by using properties from the StopwatchViewModel. If the stopwatch has been running for one hour, 32 minutes, and 19.7 seconds, the time should look like this: `01:32:19.7` (you can get 10ths of seconds by dividing milliseconds by 100M).

Finish the StopwatchView.UpdateTimeCallback method.

The StopwatchView class uses a System.Threading.Timer to write the current time to the console. That Timer uses a callback. You learned about callbacks in the ***Events and Delegates*** bonus chapter—and if you haven't read it yet, this is a great time to do that! You can download it from our GitHub page: https://github.com/head-first-csharp/fourth-edition

Here's what your UpdateTimeCallback method does:

- It uses Console.KeyAvailable to check if a key is available. If it is, it uses Console.ReadKey(true) to read the key.
- If the user pressed the spacebar, it starts the stopwatch. If the user pressed the R key, it rests the stopwatch.
- If the user pressed anything else, it makes the cursor visible, moves it to the start of the 6th line, and quits the app.

You've started building the first part of the MVVM Stopwatch project by adding the model, a skeleton of the viewmodel, and a partially completed view. Now it's time to get this first part up and running by finishing the viewmodel and view.

Here's the StopwatchView.UpdateTimeCallback method:

```
/// <summary>
/// Callback to update the time dispay that the time calls each time it ticks
/// </summary>
private void UpdateTimeCallback(object? state)
{
    if (Console.KeyAvailable)
    {
        switch (Console.ReadKey(true).KeyChar.ToString().ToUpper())
        {
            case " ":
                _viewModel.StartStop();
                break;
            case "R":
                _viewModel.Reset();
                break;
            default:
                Console.CursorVisible = true;
                Console.CursorLeft = 0;
                Console.CursorTop = 5;
                _quit = true;
                break;
        }
    }
    WriteCurrentTime();
}
```

Checking Console.KeyAvailable keeps Console.ReadKey from pausing your app because it only reads the key if it's available.

In Chapter 4 we learned that passing "true" to Console.ReadKey intercepts the key, which keeps it from writing a letter to the console.

Making the cursor visible again and positioning it below the stopwatch resets the app so the operating system's prompt looks normal.

The WriteCurentTime method positions the cursor, gets the time from the model, and writes it to the console.

Here's the StopwatchView.WriteCurrentTime method:

```
/// <summary>
/// Writes the current time to the second row and 23rd column of the screen
/// </summary>
private void WriteCurrentTime()
{
    Console.CursorTop = 1;    // This moves the cursor to the second row (rows start at 0)
    Console.CursorLeft = 23;  // This moves the cursor to the 23rd column (starting at 0)
    var time = $"{_viewModel.Hours}:{_viewModel.Minutes}:"
               + $"{_viewModel.Seconds}.{_viewModel.Tenths}";
    Console.Write(time);
}
```

Here's the complete StopwatchViewModel class:


```
using Model;

class StopwatchViewModel
{
    private readonly StopwatchModel _model = new StopwatchModel();

    public void StartStop() => _model.Running = true;

    public void Reset() => _model.Reset();

    public string Hours => _model.Elapsed.Hours.ToString("D2");

    public string Minutes => _model.Elapsed.Minutes.ToString("D2");

    public string Seconds => _model.Elapsed.Seconds.ToString("D2");

    public object Tenths => ((int)(_model.Elapsed.Milliseconds / 100M)).ToString();
}
```

Make sure you add "using Model;" to the top of your file so you can use StopwatchModel.

The viewmodel uses properties that convert the model's elapsed time into a format that's easy for the view to consume.

You can get the tenths of seconds by dividing the milliseconds by 100M. You also need to cast to an int so it doesn't display a decimal value.

Here's the program's Main method that shows the view:

```
class Program
{
    static void Main(string[] args)
    {
        new View.StopwatchView();
    }
}
```

THE VIEW HAS A TIMER THAT DETERMINES HOW OFTEN THE STOPWATCH UPDATES. THE MODEL JUST PROVIDES THE ELAPSED TIME, AND THE VIEWMODEL ***MAKES IT EASY FOR THE VIEW*** TO USE THAT ELAPSED TIME, BUT ONLY THE VIEW ACTUALLY "TICKS."

That's right! Only the view is concerned about screen updates.

Try changing the line in StopwatchView that instantiates the timer so it only ticks every 300 milliseconds:

```
var _timer = new Timer(timerCallback, null, 0, 300);
```

Run your app and start the stopwatch again—now it ticks more slowly, only adding three-tenths of a second each time it ticks. The model and viewmodel don't need to know how often the view ticks, they'll make sure to give it the correct elapsed time no matter how frequently they're asked for it.

This might remind you of how you used delta times in Unity to make your games work the even when the frame rate differs on different computers.

Make your stopwatch pause

In the second part of the stopwatch project, we'll make the stopwatch pause and resume when the user presses the spacebar.

Do this!

1 Add fields to your StopwatchModel to manage the paused state.

The model will set _paused to true when the stopwatch is paused, and _false when it's not. The _pausedAt field will work just like _startedTime – it will contain DateTime.MinValue when it's not set. The model will use _totalPausedTime to keep track of the total amount of time paused.

```
private bool _paused;

private DateTime _pausedAt;

private TimeSpan _totalPausedTime;
```

2 Modify the Running property to pause and unpause the stopwatch.

The model already uses the Running property's get accessor to indicate if the stopwatch is running, and its set accessor to start it. That means you can update the Running property to pause and unpause the stopwatch without having to make many other changes to the rest of the app.

```
/// <summary>
/// Starts and stops the stopwatch, returns true if the stopwatch is running
/// </summary>
public bool Running
{
    get => (_startedTime != DateTime.MinValue) && !_paused;

    set
    {
        if (value)
        {
            _paused = false;

            if (_pausedAt != DateTime.MinValue)
                _totalPausedTime += DateTime.Now - _pausedAt;

            if (_startedTime == DateTime.MinValue)
                _startedTime = DateTime.Now;
        } else {
            _paused = true;
            _pausedAt = DateTime.Now;
        }
    }
}
```

The get accessor is almost the same as before. The only difference is that now it also checks the _paused field to determine if the stopwatch is running.

When Running is set to true, the model unpauses, setting the _paused, _pausedAt, and _startedTime fields.

If the stopwatch was previously paused, the time that it was paused is added to _totalPausedTime.

When Running is set to false, the model sets _paused to true and _pausedAt to the current time.

3 Modify the Elapsed property to subtract the time paused.

The _totalPausedTime field will always contain the total span of time the stopwatch was paused, so we just need to modify the Elapsed field to subtract it.

```
/// <summary>
/// Returns the elapsed time, or zero if the stopwatch is not running
/// </summary>
public TimeSpan Elapsed => _paused ? _pausedAt - _startedTime - _totalPausedTime
    : _startedTime != DateTime.MinValue ? DateTime.Now - _startedTime - _totalPausedTime
    : TimeSpan.Zero;
```

The elapsed time now subtracts the started time and the total time the stopwatch was paused from the current time.

4 Reset the paused state when the stopwatch is reset.

Modify the Reset method to reset the three fields that you added to track the paused state.

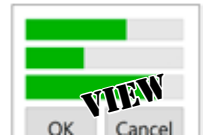
```
/// <summary>
/// Resets the stopwatch by setting its started time and unpausing it
/// </summary>
public void Reset()
{
    _startedTime = DateTime.MinValue;
    _pausedAt = DateTime.MinValue;
    _paused = false;
    _totalPausedTime = TimeSpan.Zero;
}
```

Here's a great example of how separation of concerns can help make your code easier to change. Making your stopwatch stop and start mainly affected the model, so you only need to change one line of code in the viewmodel and one line in the view.

5 Make ViewModel.StartStop toggle Model.Running.

In Part 1, the ViewModel.StartStop property just set Model.Running to true. Modify it to toggle Model.Running so it starts the stopwatch if stopped, and stops it if started.

```
public void StartStop() => _model.Running = !_model.Running;
```



6 Modify the view to print a different instruction message.

Change the view so it prints *Space to start or stop* instead of *Space to start*.

```
Console.WriteLine("Space to start or stop, R to reset, any other key to quit");
```

Now run your app. The spacebar should now pause and unpause the stopwatch.

Before you go to the next page, see if you can figure out how to add the lap time yourself. How far can you get on your own?

In Part 3 of this project, you'll add a lap time. How do you think you'll do that? What changes will you need to make to the view, viewmodel, and model?

Add lap time to your stopwatch

Most stopwatches have a lap time feature that lets you press a button to save the current elapsed time and keep it visible as the stopwatch keeps ticking. Let's add lap time to your stopwatch app by adding it to the model, then updating the view to include it, and finally filling in the viewmodel to connect the view and the model.

Do this!

1 Update the model to add lap time to its state.

Your model will keep track of the lap time in a read-only TimeSpan property that gets set to TimeSpan.Zero when the model is reset. You'll also add a SetLapTime method and set to the current elapsed time.

```
public TimeSpan LapTime { get; private set; }

public void SetLapTime() => LapTime = Elapsed;

/// <summary>
/// Resets the stopwatch by setting its started time and unpausing it
/// </summary>
public void Reset()
{
    _startedTime = DateTime.MinValue;
    _pausedAt = DateTime.MinValue;
    _paused = false;
    _totalPausedTime = TimeSpan.Zero;
    LapTime = TimeSpan.Zero;
}
```

2 Modify the view to display the lap time.

First, change the line in the view that displays the instruction to the user:

```
Console.WriteLine("Space to start or stop, R to reset, L for lap time, any other key to quit");
```

Next, add a case to the switch statement to set the lap time when the user presses the L key:

```
   case "L":
        _viewModel.LapTime();
        break;
```

The LapTime method doesn't exist in the viewmodel yet, so use the quick fix menu to generate the method.

Your viewmodel doesn't have a LapTime method yet, so use **the quick fix menu to generate it**. Finally, modify the WriteCurrentTime method to display the lap time, using the quick fix menu to generate any properties missing from the viewmodel.

```
/// <summary>
/// Writes the current time to the second row and 24th column of the screen
/// </summary>
private void WriteCurrentTime()
{
    Console.CursorTop = 1;    // This moves the cursor to the second row (rows start at 0)
    Console.CursorLeft = 23;  // This moves the cursor to the 24th column
    var time = $"{_viewModel.Hours}:{_viewModel.Minutes}:"
               + $"{_viewModel.Seconds}.{_viewModel.Tenths}";
    var lapTime = $"{_viewModel.LapHours}:{_viewModel.LapMinutes}:"
               + $"{_viewModel.LapSeconds}.{_viewModel.LapTenths}";
    Console.Write($"{time} - lap time {lapTime}");
}
```

Use the quick fix menu to generate the properties in the viewmodel used in this statement.

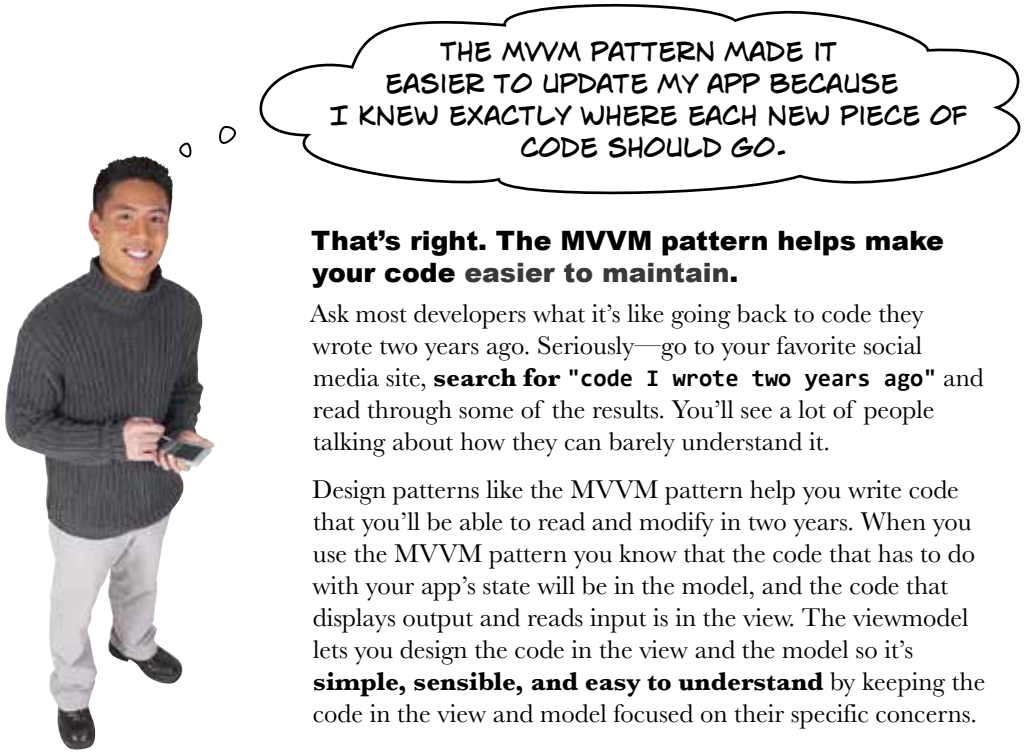
3 Fill in the viewmodel to connect the view to the model.

In the last step, you generated four properties and a method in the viewmodel. Implement those methods so they connect the view's behavior to the model's state.

```
public void LapTime() =>_model.SetLapTime();

public string LapHours => _model.LapTime.Hours.ToString("D2");

public string LapMinutes => _model.LapTime.Minutes.ToString("D2");

public string LapSeconds => _model.LapTime.Seconds.ToString("D2");

public string LapTenths => ((int)(_model.LapTime.Milliseconds / 100M)).ToString();
```

Now run your app. Pressing L will record the lap time and keep it displayed.

```
                    00:00:19.5 - lap time 00:00:15.4

Space to start or stop, R to reset, L for lap time, any other key to quit
```



That's right. The MVVM pattern helps make your code easier to maintain.

Ask most developers what it's like going back to code they wrote two years ago. Seriously—go to your favorite social media site, **search for `"code I wrote two years ago"`** and read through some of the results. You'll see a lot of people talking about how they can barely understand it.

Design patterns like the MVVM pattern help you write code that you'll be able to read and modify in two years. When you use the MVVM pattern you know that the code that has to do with your app's state will be in the model, and the code that displays output and reads input is in the view. The viewmodel lets you design the code in the view and the model so it's **simple, sensible, and easy to understand** by keeping the code in the view and model focused on their specific concerns.

there are no Dumb Questions

Q: I understand how the stopwatch and basketball roster apps work, but I'm still not clear on the idea behind MVVM. Can you explain it one more time?

A: Model-View-ViewModel, or MVVM, is a **design pattern**. That means it's an *approach*—you can even think of it as a sort of *template*—to help you design your code in a way that really makes the relationships and interactions between your classes clear, and helps you think about the design of your app. The general idea is that the code that keeps track of the state of your app goes in the model, the code that has to do with what's displayed to the user goes in the view, and the code that connects the view and model together goes in the viewmodel.

Q: I used string formatting in the viewmodel to make the numbers look right. But wouldn't that make more sense in the view, since it has to do with the way the numbers are displayed?

A: We thought the string formatting made more sense in the viewmodel. But there are no hard-and-fast rules that say specific behavior must be in the view, viewmodel, or model. If you think the string formatting makes more sense in the view, then that's where you should put it!

(Spoiler alert: we also put the string formatting in the viewmodel because we'll reuse it in the next part of the project.)

Q: You just said there's no hard-and-fast rule about where specific things go, but earlier you said that hard-coded data goes in the viewmodel, and in a WPF project the view contains XAML with data binding to objects in the viewmodel. Aren't those hard-and-fast rules?

A: That's right, we did say those things! The idea behind the MVVM pattern is that the code for the user interface (the view) is decoupled from the code for the state (the model). The way we decouple that code is by creating a separate set of objects (the viewmodel) that connects them.

When we say that hardcoded data goes in the viewmodel, we're not laying down a rule. We're just applying common sense. For example, in the basketball roster app, the view consisted of a UserControl. The UserControl created an instance of a viewmodel object, and used it for data binding. The app needed to display some data—the teams and their players—so that data needed to go somewhere. It doesn't make sense to put it in the model, because the job of the model is to manage the app's state, no matter what data it stores. And it doesn't make sense to put it in the view, because the view's job is to manage the UI. So that just leaves the viewmodel as the only logical place to put the data.

Q: Can you explain decoupling one more time?

A: Sure. When we say that one object is **decoupled** from another, that means there are no dependencies between them: they don't have any references to each other, call each others' methods, use each others' properties, etc. When you hear developers complain that their code is difficult to modify, it's usually because there are a lot of dependencies between classes, so making a change in one class requires them to make changes in lots of others. It can be really frustrating to start making a tiny change to one class, only to discover that you need to change a bunch of others along the way.

When you follow the MVVM pattern, your view is decoupled from the view because it has no references to it. The view is **loosely coupled** to the model: it only has minimal dependencies on the model, only then those references go through the viewmodel. That loose coupling makes your code easier (and much less frustrating!) to maintain.

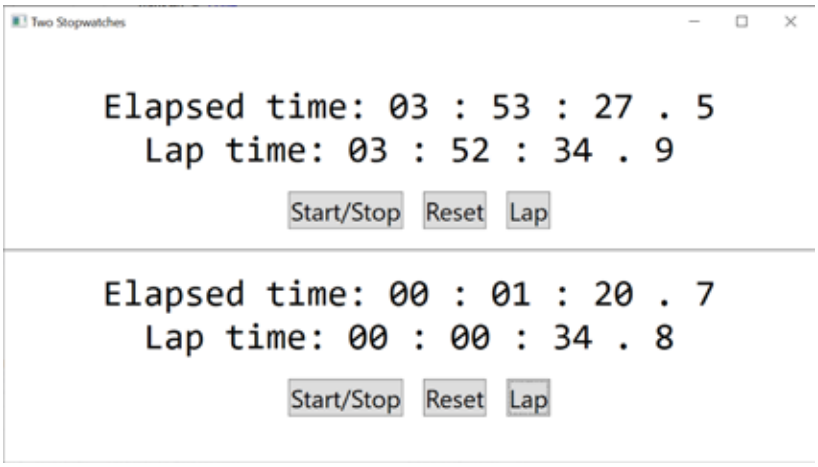
Q: Remind me again why you started adding an underscore to the beginning of your private fields?

A: If you go to GitHub and look at the code for a lot of C# projects, you'll notice that many people put an underscore in front of their private fields. They do this to make it easier to spot those fields when they're using them. Some people feel that it's easier to read code when they can tell at a glance the difference between a local variable and a private field, the same way that they use PascalCase versus camelCase for public fields and properties. We included the naming convention of underscores for private fields to give you the opportunity to decide for yourself whether you think it helps you make your code more readable.

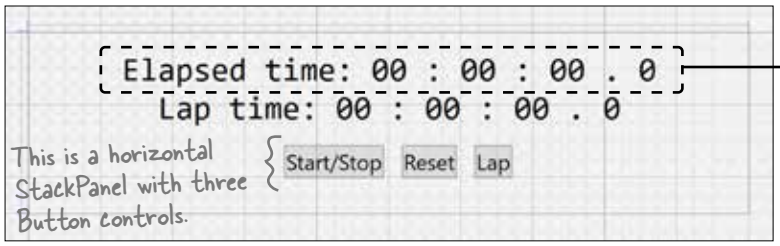
The viewmodel allows the view and the model to be loosely coupled because the view doesn't directly refer to anything in model, and the doesn't refer to the view at all.

Next we'll build a WPF version of the stopwatch

The WPF app will use the same model and viewmodel as the console app, but we'll use a different view. Here's what the WPF will look like—it will have two independent stopwatches:



The main window of the WPF app contains a StackPanel, which contains two instances of a stopwatch user control. The UserControl contains a StackPanel with two TextBlock controls and a StackPanel with three buttons.



Here's an excerpt from the XAML that you'll build later on in the app.

```
<TextBlock FontFamily="Consolas" FontSize="36" HorizontalAlignment="Center">
    <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}"/>
    <Run>.</Run>
    <Run Text="{Binding Tenths, Mode=OneWay}"/>
</TextBlock>
```

This <Run> tag is bound to the Tenths property in the model.

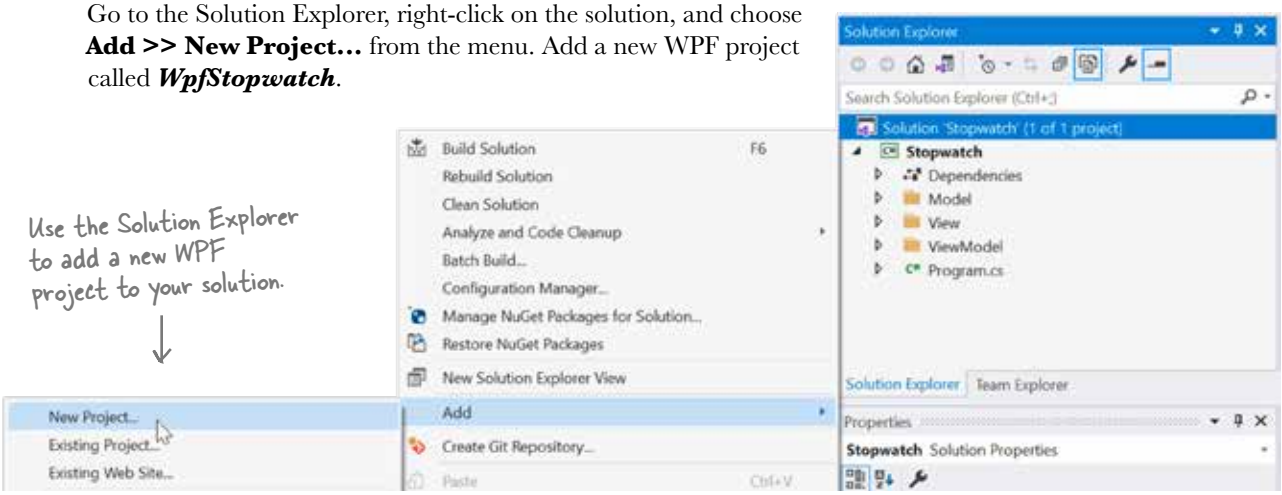
This is a TextBlock control. It uses data binding (just like you used in Chapter 6) to get its value from a static resource—in this case, an instance of a view object. But we want to use separate properties in the view to set the hours, minutes, seconds, and milliseconds. We'll use <Run> tags to accomplish that. A run is a block of text that lives inside the TextBlock. A run can contain static text, or it can be set using data binding.

Add a WPF project to your solution

In Chapter 9 you added a unit test project to the solution for your app to manage Jimmy's comics and added a project reference so your new project could use the classes in the console app project. You'll do the same thing here, except this time you'll **add a WPF project to your solution**.

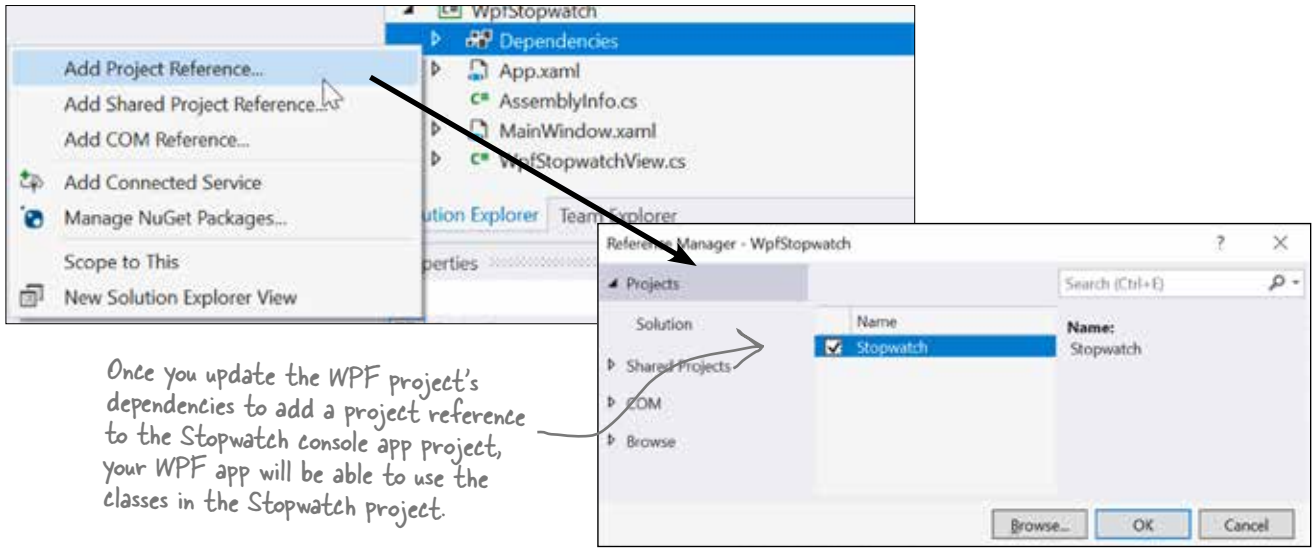
1 Add a WPF project called WpfStopwatch to your solution.

Go to the Solution Explorer, right-click on the solution, and choose **Add >> New Project...** from the menu. Add a new WPF project called ***WpfStopwatch***.



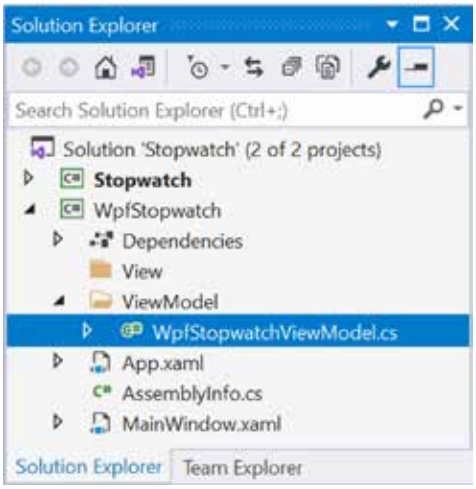
2 Add project reference to your WpfStopwatch project.

Expand to your new project in the Solution Explorer, right-click on Dependencies, and **add a new project reference to the Stopwatch console app project**. Now your new WPF project will be able to use classes in the console app project.



3 Create the View and ViewModel folders in your WpfStopwatch project.

Your WPF app will have a model, view and viewmodel. But while you'll create a new view, you'll ***reuse*** the same viewmodel from the stopwatch console app.



Since you'll be reusing the StopwatchViewModel class from your console app, so you'll need to make it public (just like you did for the classes you were testing in Chapter 9). Edit StopwatchViewModel and **add the `public` keyword**:

```
namespace Stopwatch.ViewModel
{
    using Model;

    public class StopwatchViewModel
    {
```

Now you're ready to add your view that uses the viewmodel.

4 Create the StopwatchControl user control and add the viewmodel as a static resource.

Create a new user control called StopwatchControl. You don't need to create a View folder, because everything in this new project is part of the view. Just create it in the root of the WpfStopwatch project.

Next, you'll add an instance of StopwatchViewModel as a static resource, just like you added a static resource to your WPF project in Chapter 6. Add this `<UserControl.Resources>` above the `<Grid>` tag:

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Key="viewModel"/>
</UserControl.Resources>
```

The IntelliSense won't help you with this line yet, because you haven't added the xmlns property yet.

Notice how the IDE added a red squiggly underline under the resource? That's because you can only use a class in your XAML if the container has an **`xmlns` (or XML namespace) property**. Luckily, the IDE will add it for you automatically. Open the Quick Fix menu and choose Add xmlns Stopwatch.ViewModel:

When you choose the QuickFix menu option, the IDE will add this line to your <UserControl> tag:

```
xmlns:viewmodel="clr-namespace:Stopwatch.ViewModel;assembly=Stopwatch"
```

As soon as the `viewmodel` namespace is defined as Stopwatch.ViewModel in your XAML, you can use classes from that namespace, and the red squiggly lines should disappear.

You may still see a compile error:

If you get a compile error, rebuild your solution, and if that doesn't work, unload and reload the project.

If you see that error, **choose Rebuild Solution from the Build menu** to rebuild the solution. If that doesn't work, use the Solution Explorer to unload and reload the WPF project.

Make your viewmodel implement INotifyPropertyChanged

You're going to be binding your StopwatchControl to the static StopwatchViewModel instance. We learned in Chapter 6 that when you bind XAML to properties in a class, if you want the class to change its properties and update the user interface, it needs to implement the INotifyPropertyChanged property and raise its PropertyChanged event when its properties change.

1 Add INotifyPropertyChanged to the StopwatchViewModel class.

Go back to the Stopwatch project and modify your StopwatchViewModel class to make it implement the INotifyPropertyChanged interface. Then use the Quick Fix menu to add the necessary using directive.

```
namespace Stopwatch.ViewModel
{
    using Model;

    public class StopwatchViewModel : INotifyPropertyChanged
    {
```

The Quick Fix menu will let you automatically add the using directive for the namespace that includes the interface.

2 Modify StopwatchViewModel to implement INotifyPropertyChanged.

The INotifyPropertyChanged interface will still have a squiggly underline. Use the **Quick Fix menu** again, this time to **implement the INotifyPropertyChanged interface**. Your code should now look like this:

```
namespace Stopwatch.ViewModel
{
    using Model;
    using System.ComponentModel;

    public class StopwatchViewModel : INotifyPropertyChanged
    {
        private readonly StopwatchModel _model = new StopwatchModel();

        public event PropertyChangedEventHandler PropertyChanged;
```

First you used the Quick Fix menu to add this using directive.

Then you used the Quick Fix menu to implement the interface, which added this event declaration. If you need a refresher on events, have a look at the Events and Delegates bonus chapter.

3 Add an OnPropertyChanged method to raise the PropertyChanged event.

In the *Events and Delegates* bonus chapter, we learned about the standard pattern for adding methods to raise events. We'll follow that pattern by **adding a method called OnPropertyChanged** to raise the event:

```
public void OnPropertyChanged(string propertyName) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
```

Now your StopwatchViewModel class is ready for the StopwatchControl to use it as a data binding context.

If you need a refresher on how ?.Invoke works, download the Events and Delegates bonus chapter from our GitHub page.

3 Add the code-behind for StopwatchControl.

Here's the code-behind for your user control. Everything that's in it is something you've seen before earlier in the book. It has a DispatchTimer that ticks every tenth of a second (or 100 milliseconds), and every time it ticks it uses the viewmodel's OnPropertyChanged method to get it to fire a PropertyChanged event.

```
namespace WpfStopwatch
{
    using Stopwatch.ViewModel;
    using System.Windows.Threading;

    /// <summary>
    /// Interaction logic for StopwatchControl.xaml
    /// </summary>
    public partial class StopwatchControl : UserControl
    {
        DispatcherTimer _timer = new DispatcherTimer();
        StopwatchViewModel _stopwatchViewModel;

        public StopwatchControl()
        {
            InitializeComponent();

            _stopwatchViewModel = Resources["viewModel"] as StopwatchViewModel;

            _timer.Interval = TimeSpan.FromMilliseconds(100);
            _timer.Tick += TimerTick;
            _timer.Start();
        }

        private void TimerTick(object sender, EventArgs e)
        {
            _stopwatchViewModel.OnPropertyChanged(String.Empty);
        }

        private void StartStopButton_Click(object sender, RoutedEventArgs e)
        {
            _stopwatchViewModel.StartStop();
        }

        private void ResetButton_Click(object sender, RoutedEventArgs e)
        {
            _stopwatchViewModel.Reset();
        }

        private void LapButton_Click(object sender, RoutedEventArgs e)
        {
            _stopwatchViewModel.LapTime();
        }
    }
}
```

These using directives let you add the StopwatchViewModel class from your console app project and the DispatchTimer that you used in your beehive management system and animal matching game.

This is the same DispatcherTimer that you used in Chapters 1 and 6. Recognize the TimeSpan struct we use below? The timer uses a static method from that struct to set the tick interval.

You used the Resources dictionary in Chapter 6 to get a references to the Queen object from the window's static resources.

When an object calls its PropertyChange event with PropertyName set to an empty string, it causes a control using it as data context to update all of its bound properties

The buttons' Click event handlers call the methods in the viewmodel.

Add the XAML for the user control

Here's the XAML for the user control. It has two TextBlock controls with the <Run> tags that we showed you earlier. The timer in the code-behind causes the bound properties to refresh every tenth of a second.

```
<UserControl x:Class="WpfStopwatch.StopwatchControl"
             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"
             xmlns:local="clr-namespace:WpfStopwatch"
             xmlns:viewmodel="clr-namespace:Stopwatch.ViewModel;assembly=Stopwatch"
             mc:Ignorable="d"
             d:DesignHeight="200" d:DesignWidth="800">
    <UserControl.Resources>
        <viewmodel:StopwatchViewModel x:Key="viewModel"/>
    </UserControl.Resources>

    <Grid DataContext="{StaticResource ResourceKey=viewModel}">
        <StackPanel Margin="20">
            <TextBlock FontFamily="Consolas" FontSize="36" HorizontalAlignment="Center">
                <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}"/>
                <Run>.</Run>
                <Run Text="{Binding Tenths, Mode=OneWay}"/>
            </TextBlock>

            <TextBlock FontFamily="Consolas" FontSize="36" HorizontalAlignment="Center">
                <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}"/>
                <Run>.</Run>
                <Run Text="{Binding LapTenths, Mode=OneWay}"/>
            </TextBlock>

            <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
                <Button Margin="20,20,20,0" FontSize="24" Click="StartStopButton_Click">
                    Start/Stop
                </Button>
                <Button Margin="0,20,20,0" FontSize="24" Click="ResetButton_Click">
                    Reset
                </Button>
                <Button Margin="0,20,0,0" FontSize="24" Click="LapButton_Click">
                    Lap
                </Button>
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```

We set the DesignHeight to 200 because the control is not very tall.

This is just like the data binding that you've used in previous WPF projects.

Here's the TextBlock with <Run> tags that we explained earlier in the chapter.

The TextBlock uses <Run> tags to bind specific parts of the text to properties in the viewmodel.

We gave the buttons a larger font size.

Create a window with two StopwatchControl controls

Now that your view is finished, you can use it in the main window. Here's the XAML for it. You'll replace the Grid control with a StackPanel that contains two stopwatch controls. Since your StopwatchControl is in the same namespace as MainWindow and it has the `xmlns:local` property, you'll add StopwatchControls to your window like this: `<local:StopwatchControl/>`

You'll also add a Rectangle control to the StackPanel separate the two stopwatch controls.

```
<Window x:Class="WpfStopwatch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfStopwatch"
        mc:Ignorable="d"
        Title="Two Stopwatches" Height="450" Width="800">

    <StackPanel VerticalAlignment="Center">

        <local:StopwatchControl/>

        <Rectangle Fill="Black" Width="800" Height="1"/>

        <local:StopwatchControl/>

    </StackPanel>
</Window>
```

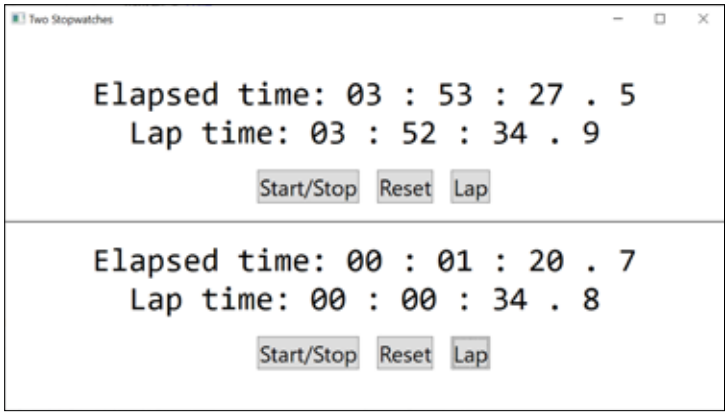
The Rectangle control draws a rectangle in the window. You used it in the Events and Delegates bonus chapter when you were experimenting with routed events..

Like before, if you run into an error where your window doesn't recognize the StopwatchControl, try rebuilding or unloading and reloading the project.

Run your app

Your app is in a separate project, so you'll need to run it differently. Right-click on the WpfStopwatch project in the Solution Explorer and choose **Debug >> Start New Instance**. You can also right-click on it and choose **Set as Startup Project** – once you do that, you can use the Start Debugging button in the toolbar. Your app should have two stopwatches that work independently of each other.

The app has two independent stopwatches. Each StopwatchControl has its own timer and instance of StopwatchViewModel.



BULLET POINTS

- When you use the **Model-View-ViewModel** (or **MVVM**) design pattern, you split your app into three layers: the view, the viewmodel, and the model.
- The objects that hold the state of the app live in the **model**. The state of the app is managed by the objects in memory that determine what your app "knows."
- Objects that the user interacts with go in the **view**.
- The **viewmodel** has the properties that can bind to the controls in the view. The viewmodel is like the plumbing that connects the objects in the view to the objects in the model, using tools you already know how to work with.
- **UserControl** is a base class that lets you define your own controls to use in your XAML apps. You design your code with XAML and code-behind, just like a window.
- **`xmlns:` properties** are like using directives for your XAML code. They let you include controls and resources from different namespaces in your project.
- **Hard-coded data**—or data that's explicitly added in the code—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.
- The classes in the view, viewmodel, and model are **loosely coupled** with each other.
- In a typical MVVM WPF app, only classes in the viewmodel implement INotifyPropertyChanged, because the viewmodel contains the only objects that XAML controls are bound to.
- The TimeSpan struct keeps track of a length of time, and
- The DateTime struct keeps track of a specific date and time. DateTime.MinValue is a special value with the smallest valid time. Subtracting one DateTime value from another returns a TimeSpan.
- The Timer in the System.Threading namespace uses a TimeSpan to set its tick interval, and calls a callback method every time it ticks.
- **Thread.Sleep** lets you pause your app without wasting CPU cycles.
- Many people put an **underscore** in front of their private fields to make it easier to spot those fields when they're used.
- **`<Run>` tags** represent blocks of text that lives inside the TextBlock. A run can contain static text, or it can be set using data binding.