O'REILLY®

Fourth Edition

Head First C#

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

Andrew Stellman
& Jennifer Greene

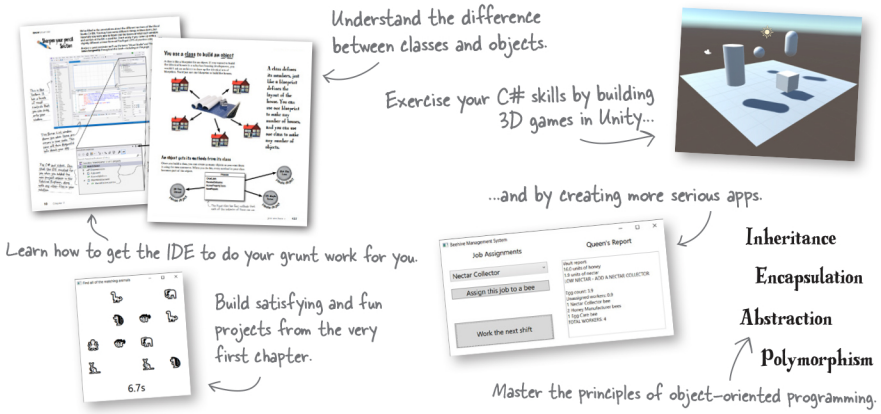
A Brain-Friendly Guide

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

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

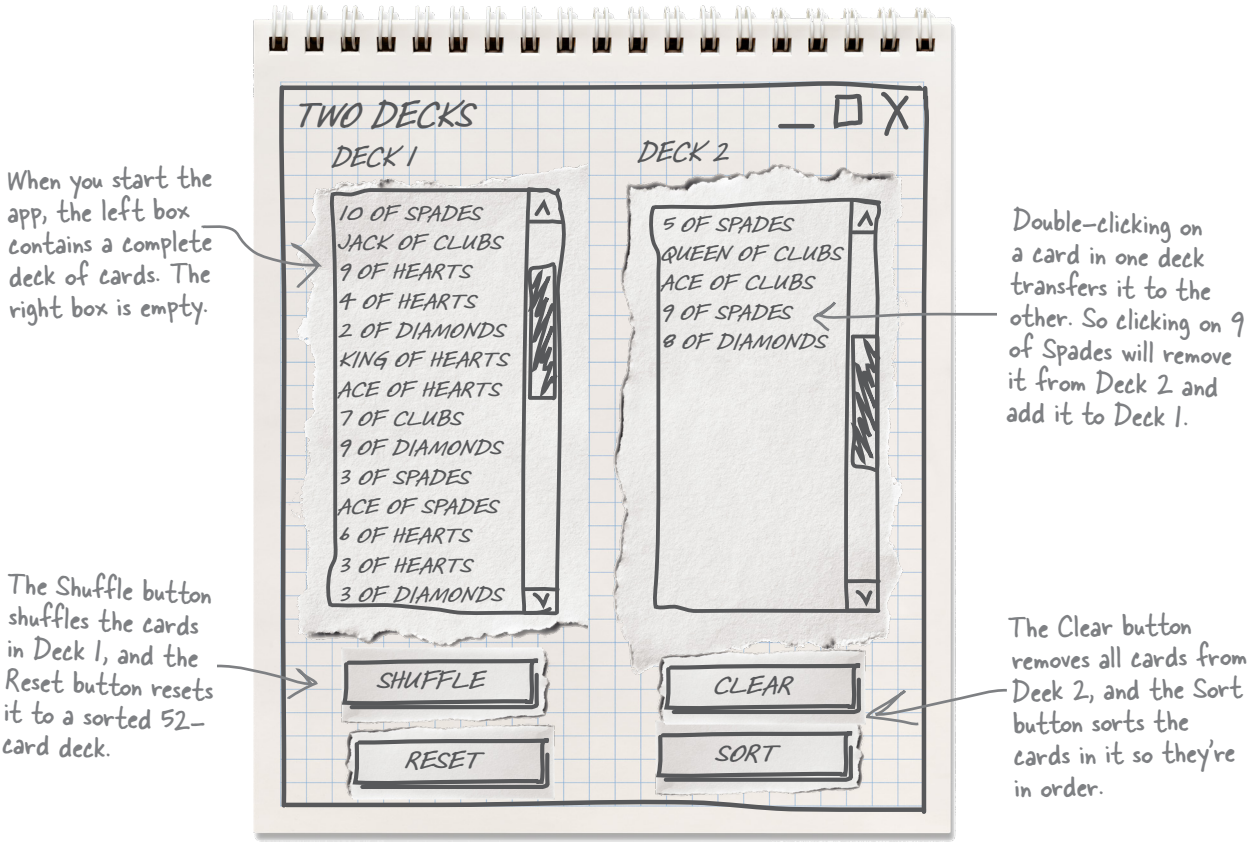
No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI] [2020-11-13]

Chapter 8 downloadable exercise: Two Decks

In the next exercise, you'll build an app that lets you move cards between two decks. The left deck has buttons that let you shuffle it and reset it to 52 cards, and the right deck has buttons that let you clear and sort it.



One of the most important ideas we've emphasized throughout this book is that writing C# code is a skill, and the best way to improve that skill is to ***get lots of practice***. We want to give you as many opportunities to get practice as possible!

That's why we created **additional Windows WPF and macOS ASP.NET Core Blazor projects** to go with some of the chapters in the rest of this book. We've included these projects at the end of the next few chapters too. We think you should take the time and do this project before moving on to the next chapter, because it will help reinforce some important concepts that will help you with the material in the rest of the book.

This PDF contains the WPF version of the project. You can find the ASP.NET Blazor Web Application version on the book's GitHub page:

https://github.com/head-first-csharp/fourth-edition

In this project, we're going to use access keys to add keyboard shortcuts. So to prepare for that, let's take step back and learn a bit about the history of control schemes—a perfect opportunity to learn another lesson from video game design.

Keyboards and controllers

Game design... and beyond

You've probably played games that use a familiar control scheme: video game controllers with two joysticks, four buttons, and a D-pad, or a mouse to look, W-A-S-D keys to move, spacebar to jump. That's the result of decades of evolution.

- The **WASD layout** dates back to the mid-1980s. Early PC games from the '80s and early '90s were more likely to use arrow keys, taking advantage of their "inverted T" layout. It wasn't until the popularity of first-person shooters in the late '90s—especially Quake (1996) tournaments and Half Life (1998)—that the WASD layout really took hold.
- The first video game console, the Magnavox Odyssey (1972), had a controller with two **paddles**, or knobs that would each control horizontal or vertical movement. While we don't see paddles much anymore, in many ways they're the predecessor to modern racing wheel controllers.
- The Atari 2600 (1977) was enormously successful, and its **joystick**, with one button and an 8-directional stick, was the first ubiquitous game controller. It became a standard for many game systems and computers in the 1980s—there were even adaptors for the IBM PC and Apple][.

Modern game controllers have similar layouts across different consoles. Game controllers, like keyboard layouts, actually evolved in a symbiotic relationship along with the games that used them.

- Nintendo introduced the **D-Pad** (or control pad), a flat rocker button, with their 1983 Famicom/NES console. Since then, a 4- or 8-direction D-Pad has been a mainstay of controllers, including the current Xbox One controller.
- Their SNES (1990) also introduced the **popular layout** still seen in modern game controllers, with the D-Pad on the left, select and start buttons in the middle, a diamond of four buttons on the right, and shoulder buttons on the rear.
- Sony **set the standard for modern video game controllers** with its PlayStation DualShock (1997) that featured dual analog joysticks, a familiar button layout, rumble for physical feedback, and an ergonomic shape.
- Nintendo continued to push the boundaries, popularizing **motion controllers** that track the player's physical movements with their 2006 Wii console, an important step in making games more immersive and engaging.

Video game controls are about more than just hardware. A game's control scheme—or what the different keys, clicks, buttons, and stick movements, actually do—makes a huge difference in gameplay.

- While not the first game to use a joystick, **Pac Man**, released in 1980—during a period known as the golden age of video games— featured the now-iconic ball top stick. It stood out for its intuitive play: move the stick up, the player goes up; move it down, the player goes back down.
- A control scheme can **affect the difficulty** of a game. The arcade game *Defender* (1980), one of the first games with a side-scrolling, multi-screen playfield, is remembered as one of the most difficult of the era, in part due to its control scheme with less-intuitive buttons to thrust and reverse direction.
- Many modern games feature a familiar scheme for a two-stick controller: move with the left stick, look with the right stick. This was the result of many years of **symbiotic evolution** between game designers, hardware, and players.

- In the early 2000s, when all of the major consoles featured dual-stick controllers, game designers were still figuring out how to work with them, and some games introduced **multiple options** with different control schemes.
- As players and game designers got used to these new controller designs, they found new ways to use them in games. With more buttons came **combos**, a gameplay element that involves an often complex set of actions that the player must be performed in sequence—like a set of button presses in a fighting game that yields an unblockable attack.

Now let's pick up where we left off at the end of Chapter 8.

WE JUST USED A KEYBOARD SHORTCUT TO BRING UP THE QUICK FIX MENU. LEARNING ALL THE COMBOS IN A FIGHTING GAME MAKES YOU A BETTER PLAYER. IS LEARNING IDE KEY COMBOS LIKE THAT?

Getting IDE shortcuts into your muscle memory helps you code.

When you've been working on one of the projects in this book, have you had the feeling of time flying by? If you haven't felt that yet, don't worry... it'll come! Developers call that "flow"—it's a state of high concentration where you start working on the project, and after a while (and if there are no interruptions) the rest of the world sort of "slips away" and you find time passing quickly. This is a real state of mind that psychological researchers have studied—especially sports psychologists (if you've ever heard an athlete talk about being "in the zone," it's the same idea). Artists, musicians, and writers also get into a state of flow. Getting familiar with the IDE's keyboard shortcuts can help you more quickly get into—and stay in—your own state of flow.

IDE Tip: Keyboard Shortcuts

Get familiar with the keyboard shortcuts in your IDE! And Microsoft has put together a really handy reference with some of the handiest shortcuts: https://aka.ms/vsm-vs-keys – print that out and stick it on the wall near your computer.

Also take a look at Microsoft's documentation page on navigating code in Visual Studio, which has more useful shortcuts: https://docs.microsoft.com/en-us/visualstudio/ide/navigating-code

But don't take our word for it. Here's some great advice from Tatiana Mac, a great developer, and also a concert pianist—which gives her a unique viewpoint:

> *For me, effective coding is about transferring rote actions to muscle memory to maintain my mind palace. Being a power keyboard user is the trick (natural transition for a former concert pianist). Even if you don't play the piano, here's how you can keyboard more. You will:*
>
> - ★ *Appreciate keyboard accessible programs/sites!*
> - ★ *Learn common patterns. Mistake keys will reveal other new shortcuts!*
> - ★ *Be slow at first, but that's okay!*
>
> *Shortcuts will speed you up and, more importantly, reduce cognitive load and help you keep focus. You can get fancy and learn more complex ones. I suggest getting adept at these first. Small learnings like this compound over time. You won't notice how much it helps until you switch code editors and realize how engrained you are.*

You can see her full advice here, including specific keyboard commands that she recommends you learn to get a great start: https://bit.ly/Keyboard_Tips

Take the time to get especially familiar with the various options that pop up in the Quick Fix menu. You've already used it to implement interfaces and generate methods... but it does so much more! Next time you need to add a using declaration, add the code that needs it and then pop up the Quick Fix menu – you'll see an option to add the missing using declaration.

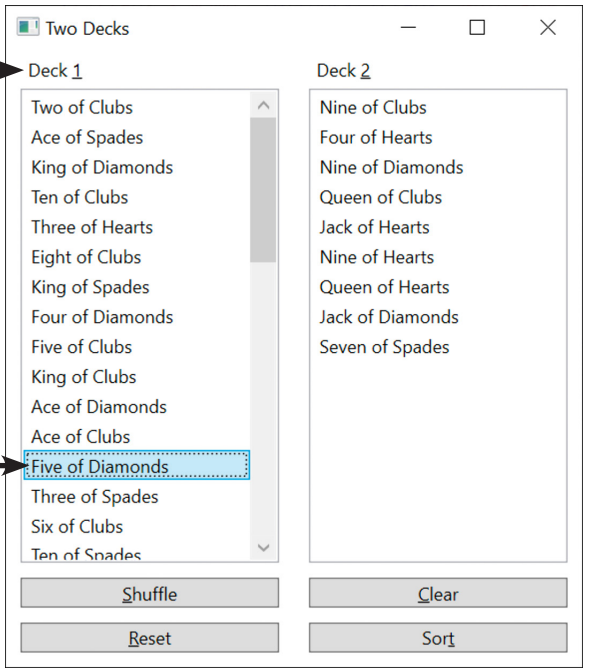
Let's build an app to work with decks of cards

In this exercise, you'll build an app that lets you move cards between two decks. It will have two ListBox controls that show the cards in each deck. The left deck has buttons that let you shuffle it and reset it to 52 cards, and the right deck has buttons that let you clear and sort it. You'll use a Grid with two columns and four rows that contain two ListBoxes, four buttons—and two new controls you haven't seen yet, Label controls, which we'll use to give your ListBoxes **access keys**.

Pop up most Windows apps—including Visual Studio—and tap the Alt key. You'll see underscores display under various buttons, menu items, and other controls. Those are access keys. Hold down alt and tap the key and you jump straight to the control that has it underlined. We'll use access keys to make your app fully keyboard accessible.

This is a Label control. It's used to label other controls and provides access keys that you can use to jump straight to them. You set its access key by putting an underscore _ to the left of a character in the content, and set use its Target to set the control that it jumps to—and the IDE helps makes it easy to set the Label's target.

The app lets the user move cards between the two decks in one of two ways: double-clicking on the card that you want to move, or selecting it (using mouse or arrow keys) and pressing enter. Both actions remove the card and add it to the other deck.



Row 0 has its height set to "Auto" so it grows or shrinks to fit its Label controls.

Row 1 has its height set to the default 1* so it fills up the rest of the window.

Rows 2 and 3 also have their row height set to "Auto" so they grow to fit their Buttons.

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

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

Here are the grid row and column definitions. The top row and bottom two rows have their height set to Auto, which causes them to expand to fit the contents of their cells. The second row has the default height (the same as 1*) so it fills up the rest of the window. Each ListBox has its vertical alignment set to Stretch so it fills up the row. That combination causes the ListBoxes to expand to fill up the space not taken up by the other controls.

Long Exercise

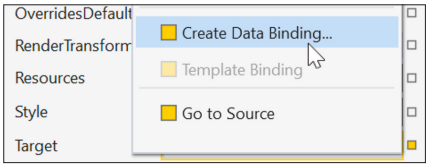
We're going to ***break this project into several parts***. In this part, you'll lay out the window for the app. We want this app to be fully **keyboard accessible**. You'll add **access keys** to the buttons, and use **Label controls** to add access keys to the ListBoxes. Start by creating a new WPF app called *TwoDecksWPF*.

1. Set the **title and width** of the <Window>: `Title="Two Decks" Height="450" Width="400"` – then **add the row and column definitions** that we just gave you. Now you're ready to **add the controls**. Here's what you'll add:

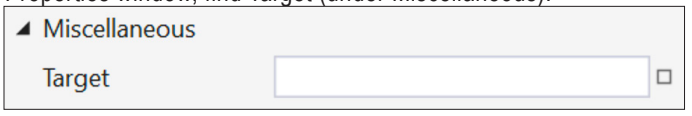
- Add a Label control to each of the two cells in the top row of the Grid. Set the values for their Content properties to `"Deck 1"` and `"Deck 2"` and use the Properties window to name them `deck1Label` and `deck2Label`.
- Add a ListBox control to the two cells in the second row (`Grid.Row="1"`). Use the Properties window to name the two ListBox controls `leftDeckListBox` and `rightDeckListBox`.
- Add a Button control to each cell in the bottom two rows and set their Contents to match the screenshot (`"Shuffle"` and `"Reset"` in the left column, `"Clear"` and `"Sort"` in the right column). Name the four buttons `shuffleLeftDeck`, `resetLeftDeck`, `clearRightDeck`, and `sortRightDeck`.
- Give each Label control a left margin of 10 (`Margin="10,0,0,0"`) and the other controls left, right, and bottom margins of 10 (`Margin="10,0,10,10"`). Reset all other properties to their defaults.

2. **Add an access key to each button** by adding an underscore in the Content just before the letter the app should use as an access key. For example, we want the focus to jump to the Shuffle button when the user presses Alt-S, so its Content will be: `Content="_Shuffle"` — and remember, the access keys are normally hidden when you run the app, but you can test them by tapping the alt key to make it reveal the access keys. The other Button contents will be `"_Reset"`, `"_Clear"`, and `"Sor_t"` so they can be accessed with Alt-R, Alt-C, and Alt-T (but pressing them won't do anything yet, because they're clicking buttons that don't do anything).

3. **Add access keys to the two Label controls** by setting their contents to `"Deck _1"` and `"Deck _2"`. That's not all you need to do to get the labels to work. When the user presses Alt-1 or Alt-2, we want the focus to jump to the ListBox controls, not the Labels. We can do this by **adding a target**—and Visual Studio makes it really easy to do that.

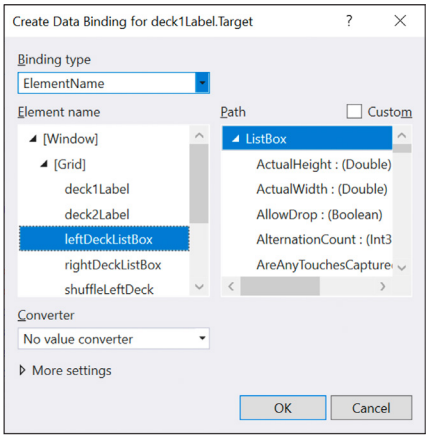


1. Click on the Label the Designer or XAML editor to select it. Then in the Properties window, find Target (under Miscellaneous):



2. Click on the box next to Target and choose **Create Data Binding...**

3. The IDE will pop up its Create Data Binding window, and you should see the names of the controls that you set in Step 1 above. The Element Name box shows the controls in your window in a nested hierarchy, with Window at the top, Grid under that, and the controls you added to the Grid. Click on the control that you want to focus on when the user uses the Label's access key: for the Deck 1 label choose leftDeckListBox, and for the Deck 2 label choose rightDeckListBox.



Once you've set the Target properties, run your app and test access keys by running the app. You should see the focus jump to the left ListBox when you press Alt-1, and it should jump to the right ListBox when you press Alt-2.

Long Exercise Solution

This is the solution to the Part I. We'll get to the rest of the long exercise soon.

Here's the XAML that you created in the first part of the exercise. You laid out a window with two columns and four rows, taking advantage of `Height="Auto"` in the row definitions to let three of the rows expand to fit their contents. You two ListBoxes, two Labels with access keys and targets set to each ListBox, and four Buttons with their own access keys.

```
<Window x:Class="TwoDecksWPF.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:TwoDecksWPF"
        mc:Ignorable="d"
        Title="Two Decks" Height="450" Width="400" >
    <Grid>

        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

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

        <Label x:Name="deck1Label" Content="Deck _1"
               Margin="10,0,0,0" Target="{Binding ElementName=leftDeckListBox, Mode=OneWay}"/>
        <Label x:Name="deck2Label" Content="Deck _2" Grid.Column="1" Margin="10,0,0,0"
               Target="{Binding ElementName=rightDeckListBox, Mode=OneWay}"/>

        <ListBox x:Name="leftDeckListBox" Grid.Row="1" Margin="10,0,10,10" />
        <ListBox x:Name="rightDeckListBox" Grid.Row="1" Grid.Column="1"
                 Margin="10,0,10,10" />

        <Button x:Name="shuffleLeftDeck" Content="_Shuffle" Grid.Row="2"
                Margin="10,0,10,10" />
        <Button x:Name="clearRightDeck" Content="_Clear" Grid.Row="2" Grid.Column="1"
                Margin="10,0,10,10" />
        <Button x:Name="resetLeftDeck" Content="_Reset" Grid.Row="3"
                Margin="10,0,10,10" />
        <Button x:Name="sortRightDeck" Content="Sor_t" Grid.Row="3" Grid.Column="1"
                Margin="10,0,10,10" />
    </Grid>
</Window>
```

Here are the grid row and column definitions that we gave you.

When you set the row's Height to "Auto" it expands to fit its contents.

When you used the Create Data Binding option in the Properties window to set the target for each Label control, it added these `{Binding...}` values. This is just like the data binding that you added to the Beehive Management System in Chapter 7.

We had to use T as the access key for the Sort button because S was already taken by the Shuffle button.

When you use a Button control's access key, it clicks the button but doesn't change focus.

Part 2: Add a Deck class to hold the cards

Do this!

We want each ListBox to show us a different deck of cards. In this next part of the project, you'll do that:

1. You'll create a Deck class—it will be a collection that holds Card objects
2. You'll add two instances of the Deck class to your window's resources (just like when did with an instance of the Queen class when you added data binding to your Beehive Management System app)
3. You'll use data binding to automatically populate each ListBox with the contents of one of the Decks

Create a Deck that extends ObservableCollection

Start by **adding the Suits and Values enums and the Card and CardComparerByValue classes** that you created earlier in Chapter 8 to your project. Make sure you use the version of Card that overrides the ToString object. Open any of these files and **rename the namespace to TwoDecksWPF** to match your app.

Next, **add this Deck class**:

```
using System.Collections.ObjectModel;

class Deck : ObservableCollection<Card>
{
   // You'll fill in this class soon...
}
```

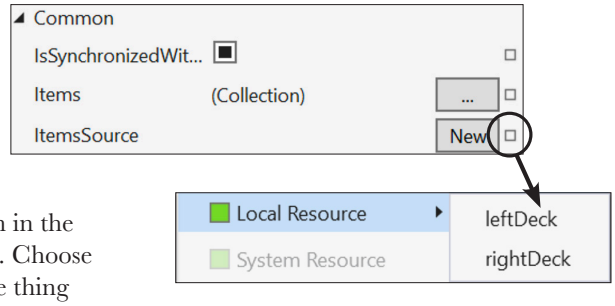
Your Deck class will extend ObservableCollection. That's a collection class that automatically notifies your WPF app any time its items have changed. You'll bind each ListBox to a Deck object. Make sure you include the using directive at the top.

You've worked with several different collections throughout this chapter: Lists, Dictionaries, Queues, and Stacks. Now you'll use the ***ObservableCollection class***, which is built for data binding. An ObservableCollection is a collection that has that same kind of notification built in, letting the app know whenever items get added, removed, or refreshed (just like the notification you added to your Queen class).

Add a Window.Resources with two instances of the Deck class

In Chapter 7 you added a Window.Resources tag to create an instance of the Queen class. Now you'll do somethig very similar. **Add this Window.Resources tag** to your xaml, just above the <Grid> tag:

```
<Window.Resources>
    <local:Deck x:Key="leftDeck"/>
    <local:Deck x:Key="rightDeck"/>
</Window.Resources>
```



Use Visual Studio to set up data binding

Click on the left ListBox in the designer, then expand Common in the Properties window and **click the box next to ItemsSource**. Choose Local Resource >> leftDeck from the menu. Then do the same thing for the right ListBox, setting its ItemsSource to rightDeck. Visual Studio will update the XAML to bind each ItemsSource to an instance of Deck:

```
<ListBox x:Name="leftDeckListBox" Grid.Row="1" Margin="10,0,10,10"
         ItemsSource="{DynamicResource leftDeck}" />
<ListBox x:Name="rightDeckListBox" Grid.Row="1" Grid.Column="1"
         Margin="10,0,10,10" ItemsSource="{DynamicResource rightDeck}" />
```

Now the ListBox items are bound to the Deck instance, so they'll update any time the collection changes.

Long Exercise

Part 3: Finish the Two Decks app. You'll finish the app by adding the code-behind for the main window and implementing the Deck class. Each of the four buttons will need a Click event handler that calls a method on the correct instance of Deck. Each ListBox will get two event handlers: you can move cards by either double-clicking on the item in the ListBox or selecting an item in the ListBox and pressing enter while the ListBox is focused, so each ListBox will need a MouseDoubleClick event handler and a KeyDown event handler.

Start with this method to move a card from one deck to the other

Add this MoveCard method to the code-behind in MainWindow.xaml.cs. If you call MoveCard(true) it moves the card in that's currently selected in the left deck to the right deck. If you call MoveCard(false) it moves the selected card in the right deck to the left deck. Look carefully at the code—every piece of it is something you've learned so far.

```
private void MoveCard(bool leftToRight) {
    if ((Resources["rightDeck"] is Deck rightDeck) && (Resources["leftDeck"] is Deck leftDeck)) {
        if (leftToRight) {
            if (leftDeckListBox.SelectedItem is Card card) {
                leftDeck.Remove(card);
                rightDeck.Add(card);
            }
        } else {
            if (rightDeckListBox.SelectedItem is Card card) {
                rightDeck.Remove(card);
                leftDeck.Add(card);
            }
        }
    }
}
```

The Window.Resources tag that you added to the XAML created a resource dictionary with two entries: references to two instances of the Deck class with keys leftDeck and rightDeck. Look closely at this code—it's how you safely get a reference to the left deck.

The ListBox.SelectedItem property returns a reference to the item that's currently selected. You used data binding to set the ItemsSource to a collection of Cards, so the selected item will be a Card reference.

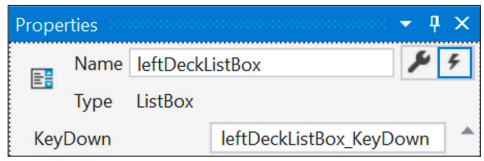
Modify the MainWindow constructor to safely get a reference to the right deck and call its Clear method.

Add a Click event handler to each Button and KeyDown and MouseDoubleClick event handlers to each ListBox

Double-click on each button to **add a Click event handler** for each button. Just like before, the IDE will automatically generate names like shuffleLeftDeck_Click because you set the Name properly. Use the `is` keyword to get a reference to either the right or left deck (like in the MoveCard method above) and call the appropriate method on the Deck class.

Next, use the Properties window to **add a KeyDown event handler and a MouseDoubleClick event handler** to each ListBox. The MouseDoubleClick event handlers will get called any time the user double-clicks on a ListBox. The first click selects the item, so the event handler just needs to call either MoveCard(true) or MoveCard(false).

The KeyDown event handler is called any time the ListBox is focused and the user presses a key. We want the user to be able to use the up and down arrows to navigate the cards, so we only want the event handler to respond when the user presses the Enter key. Take a close look at the arguments for the KeyDown event handler:




```
private void leftDeckListBox_KeyDown(object sender, KeyEventArgs e)
```

You should only call MoveCard if (e.Key == Key.Enter) – the KeyEventArgs class has a Key property, and its type is an enum called System.Windows.Input.Key. ***Use "Go to Definition / Declaration" to explore the Key enum.***

Long Exercise

Your Deck class extends ObservableCollection<Card> so you can bind each ListBox to an instance of Deck. You'll call its Add method to add a Card, RemoveAt to remove a Card from a specific index, and Clear to remove all cards. We've given you the method declarations and some of the code, and comments to help you finish the Deck class.

```
using System.Collections.ObjectModel;

class Deck : ObservableCollection<Card>
{
    private static Random random = new Random();

    public Deck() {
        Reset();
    }

    public void Reset() {
        /* Call Clear() to remove all cards from the deck, then use two for loops to add
         * all combinations of suit and value, calling Add(new Card(...)) to add each card */
        throw new NotImplementedException("The Reset method restes the 52-card deck")
    }

    public Card Deal(int index) {
        // Use base[index] to pull out the specific card and RemoveAt(index) to remove it
        throw new NotImplementedException("The Deal method will deal a card from the deck");
    }

    public void Shuffle() {
        /* Use new List<Card>(this) to create a copy of the deck, then pick a random card
         * from copy, call copy.RemoveAt to remove it, and Add(card) to add it */
       throw new NotImplementedException("The Shuffle method will randomize the cards")
    }

    public void Sort() {
        List<Card> sortedCards = new List<Card>(this);
        sortedCards.Sort(new CardComparerByValue());
        // Use a foreach loop to call Add for each card in sortedCards
        throw new NotImplementedException("The Sort method sorts the
cards.")
    }
}
```

You'll need this using directive because ObservableCollection is in this namespace.

You saw code near the beginning of Chapter 8 that used nested for loops to add cards to an array. Your Reset method will be very similar to that code.

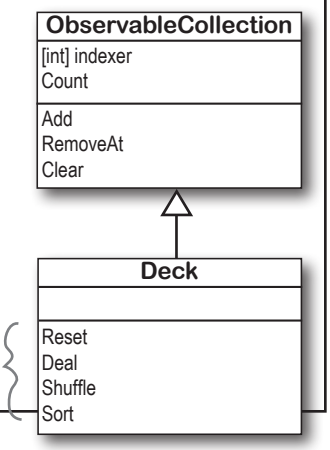
List<T> has an overloaded constructor that takes an IEnumerable<T> and initializes the list with its contents.

ObservableCollection doesn't have a Sort method! You'll need to add one yourself.

Flip back to Chapter 1 – this is really similar to the foreach loop you used to pull random emoji from a List<string>.

Your job is to replace the exceptions and comments with working code.

Your Deck class extends ObservableCollection, which means you can use members that deck inherited from the base class, including the [indexer], the Count property, and the Add, RemoveAt, and Clear methods. For example, we told you to use `base[index]` to get a specific card from the deck—that's using the inherited indexer.



You'll add these methods to the Deck class to reset, deal, shuffle, or sort the cards in the collection.

Long Exercise Solution

Your app is done! This was a larger project, but breaking it down into smaller parts gave us a way to tackle the project in steps. Here's the code-behind for the XAML window.

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

        if (Resources["rightDeck"] is Deck rightDeck)
        {
            rightDeck.Clear();
        }
    }

    private void MoveCard(bool leftToRight)
    {
        if ((Resources["rightDeck"] is Deck rightDeck)
            && (Resources["leftDeck"] is Deck leftDeck))
        {
            if (leftToRight)
            {
                if (leftDeckListBox.SelectedItem is Card card)
                {
                    leftDeck.Remove(card);
                    rightDeck.Add(card);
                }
            }
            else
            {
                if (rightDeckListBox.SelectedItem is Card card)
                {
                    rightDeck.Remove(card);
                    leftDeck.Add(card);
                }
            }
        }
    }

    private void leftDeckListBox_MouseDoubleClick(object sender, MouseButtonEventArgs e)
    {
        MoveCard(true);
    }

    private void rightDeckListBox_MouseDoubleClick(object sender, MouseButtonEventArgs e)
    {
        MoveCard(false);
    }
```

Here's where the MainWindow constructor was modified to safely get a reference to the right deck and call its Clear method.

This code safely gets a reference to the right deck. It's made up of two parts that you're familiar with: using a key to access an item in a Dictionary (in this case, the Window resource dictionary) and using the "is" keyword to safely cast it (in this case to a Deck reference).

Here's the MoveCard method that we gave you in the instructions. Take a few minutes and use the debugger to step through it, using the Locals window to keep an eye on the contents of both decks.

The ListBoxes' MouseDoubleClick event handlers just call the MoveCard method. The left deck ListBox calls MoveCard(true) to move the selected card from left to right, the right deck ListBox calls MoveCard(false) to move the card from right to left.

```
    private void leftDeckListBox_KeyDown(object sender, KeyEventArgs e)
    {
        if (e.Key == Key.Enter)
        {
            MoveCard(true);
        }
    }

    private void rightDeckListBox_KeyDown(object sender, KeyEventArgs e)
    {
        if (e.Key == Key.Enter)
        {
            MoveCard(false);
        }
    }

    private void shuffleLeftDeck_Click(object sender, RoutedEventArgs e)
    {
        if (Resources["leftDeck"] is Deck leftDeck)
        {
            leftDeck.Shuffle();
        }
    }

    private void resetLeftDeck_Click(object sender, RoutedEventArgs e)
    {
        if (Resources["leftDeck"] is Deck leftDeck)
        {
            leftDeck.Reset();
        }
    }

    private void sortRightDeck_Click(object sender, RoutedEventArgs e)
    {
        if (Resources["rightDeck"] is Deck rightDeck)
        {
            rightDeck.Sort();
        }
    }

    private void clearRightDeck_Click(object sender, RoutedEventArgs e)
    {
        if (Resources["rightDeck"] is Deck rightDeck)
        {
            rightDeck.Clear();
        }
    }
}
```

The KeyDown event handlers work just like the MouseDoubleClick event handlers, except they use the KeyEventArgs argument to move the card only if the user pressed the Enter key.

Each of the Button Click event handlers use the is keyword to safely get a reference to either the left or right deck.

```
/* Call Clear() to remo
 * combination of suit
```

When we used /* and */ to create multi-line comments in the exercise instructions, we added an extra star at the beginning of each new line. It's not necessary, but it's something that a lot of developers do because it makes multi-line comments easier to read (especially if they're indented).

Long Exercise Solution

Here's the code for the Deck class. It extends ObservableCollection<Card> so you can bind it directly to a ListBox, and adds four additional methods to work with the cards in the collection.

The ObservableCollection<T> class is a collection that's specially built for data binding, especially for displaying items in a ListBox. It implements the familiar IEnumerable<T> and ICollection<T> interfaces (just like List<T>), and also the interfaces that let WPF data binding know when its contents have changed. The Deck class uses inheritance to extend ObservableCollection—its methods use the inherited Add, RemoveAt, and Clear methods to update the cards in the collection.

```
using System.Collections.ObjectModel;

class Deck : ObservableCollection<Card>
{
    private static Random random = new Random();

    public Deck()
    {
        Reset();
    }

    public Card Deal(int index)
    {
        Card cardToDeal = base[index];
        RemoveAt(index);
        return cardToDeal;
    }

    public void Reset()
    {
        Clear();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                Add(new Card((Values)value, (Suits)suit));
    }

    public void Shuffle()
    {
        List<Card> copy = new List<Card>(this);
        Clear();
        while (copy.Count > 0)
        {
            int index = random.Next(copy.Count);
            Card card = copy[index];
            copy.RemoveAt(index);
            Add(card);
        }
    }

    public void Sort()
    {
        List<Card> sortedCards = new List<Card>(this);
        sortedCards.Sort(new CardComparerByValue());
        Clear();
        foreach (Card card in sortedCards)
        {
            Add(card);
        }
    }
}
```

The Deal method deals a card from a specific index in the Deck. It calls base[index] get the card, then RemoveAt(index) to remove it from the collection before returning it.

The Reset method clears the deck, then uses a nested for loop to add all 13 cards for each of the four suits, calling the Add method to add each card.

The Shuffle method randomizes the cards. The first thing it does is use create a copy of the deck using this overloaded List<T> constructor.

This while loop chooses a random card from the copy, removes it, and adds it back to the Deck. This is really similar to code that you wrote in your animal matching game in Chapter 1.

We gave you the first two lines of the Sort method that create a copy of the collection. This foreach loop call Add for each card in sortedCards.

Long Exercise Solution

Here are the changes to the XAML to connect the ListBoxe controls to their KeyDown event handlers and the Button controls to their Click event handlers.

```
<ListBox x:Name="leftDeckListBox" Grid.Row="1" Margin="10,0,10,10"
     ItemsSource="{DynamicResource leftDeck}"
     KeyDown="leftDeckListBox_KeyDown" />

 <ListBox x:Name="rightDeckListBox" Grid.Row="1" Grid.Column="1"
     Margin="10,0,10,10" ItemsSource="{DynamicResource rightDeck}"
     KeyDown="leftDeckListBox_KeyDown" />

 <Button x:Name="shuffleLeftDeck" Content="_Shuffle" Grid.Row="2"
         Margin="10,0,10,10" Click="shuffleLeftDeck_Click" />

 <Button x:Name="clearRightDeck" Content="_Clear" Grid.Row="2" Grid.Column="1"
         Margin="10,0,10,10" Click="clearRightDeck_Click" />

 <Button x:Name="resetLeftDeck" Content="_Reset" Grid.Row="3"
         Margin="10,0,10,10" Click="resetLeftDeck_Click" />

 <Button x:Name="sortRightDeck" Content="Sor_t" Grid.Row="3" Grid.Column="1"
         Margin="10,0,10,10" Click="sortRightDeck_Click" />
```

Test your app and make sure it works

Your app is done! This was a larger project, but breaking it down into smaller parts gave us a chance to test it out along the way—and we caught and fixed a bug in the process.

Now take some time to test it out and make sure it works the way we expect it to. Here are a few things to try:

- Make sure all of the access keys work—use them to click the buttons and focus on the decks.
- Use the access key to clear Deck Two.
- Shuffle and reset Deck One several times.
- Use the access keys to shuffle Deck One, then focus on its select, press the down arrow to move down to the next card, then press enter a bunch of times to copy random cards to Deck Two.
- Click the Sort button to sort the cards that you copied into Deck Two.

Breaking your project down into smaller parts gives you a chance to learn from each part and make changes along the way.