

O'REILLY®

~~Fourth~~ Fifth
Edition

Head First

C#

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

Andrew Stellman
& Jennifer Greene

These are the .NET MAUI projects from Chapter 8. You'll create a simple app to work display a deck of cards and shuffle, sort, and reset it, then you'll use that knowledge to create a more complex app that moves cards between two different decks.

© 2023 Andrew Stellman & Jennifer Greene, all rights reserved

This is part of an early release preview of the 5th edition of Head First C# by Andrew Stellman and Jenny Greene. We'll release the final version of this PDF when the book is published in late 2023.



A Brain-Friendly Guide

Listview is a MAUI control built for displaying collections

When you're building visual apps, it's very common to display a list of things. That's why .NET MAUI has a really convenient control called **Listview**, which displays the contents of a collection in a vertical list, adding scrollbars to let your user scroll through the list if there are too many items to fit. Let's create an app to learn more.

① Create a new .NET MAUI app called MauiCards.

For now we'll display a familiar array of birds. But later on we'll replace them with cards, and later on we'll be reusing this code, so **make sure you use the name MauiCards** for your app.

← Do this!

② Replace everything inside the <VerticalStackLayout> with a Listview control.

Open MainPage.xaml and delete the contents of the VerticalStackLayout. Replace it with this Listview inside a Border. The Listview has a HeightRequest, which sets its height:

```
<Border>
    <ListView x:Name="MyListView" HeightRequest="125" />
</Border>
```

③ Modify the MainPage constructor to add items to the Listview.

Open MainPage.xaml.cs and delete everything inside of the MainPage class except its constructor.

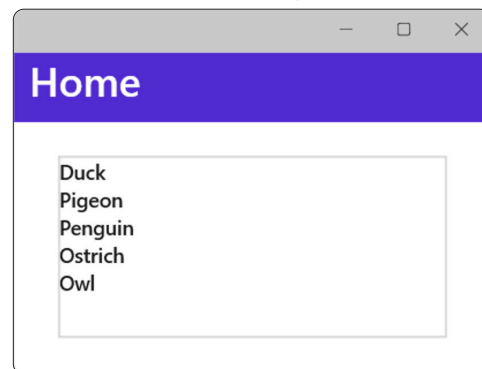
Then flip back to the end of Chapter 2 and find the code where you added birds to a Picker control. You'll do exactly the same thing, except with this Listview instead. Add this code to the MainPage constructor:

```
public partial class MainPage : ContentPage
{
    private string[] birds = new string[] {
        "Duck",
        "Pigeon",
        "Penguin",
        "Ostrich",
        "Owl"
    };

    public MainPage()
    {
        InitializeComponent();

        MyListView.ItemsSource = birds;
    }
}
```

← Back in Chapter 2 you did something really similar to this to set the items in a Picker control. You can use a Listview in exactly the same way.



③ Run your app.

Congratulations, you're displaying the contents of an array in a Listview! Technically, an array is not a .NET collection, but a Listview control can render the contents of an array just as easily as it can render the contents of a List.

ObservableCollection is a collection made for data binding

The `ObservableCollection<T>` class is a collection that's specially built for data binding, especially for displaying items in a `ListView`. It implements the familiar `IEnumerable<T>` and `ICollection<T>` interfaces (just like `List<T>`). It also implements the interfaces that let MAUI data binding know when its contents have changed.

1 Add a Button to your page to add a bird to the ListView.

Modify `MainPage.xaml` to add a `Button` control just above the closing `</VerticalStackLayout>` tag:

```
<Button Text="Add an item" Clicked="Button_Clicked" />
```

2 Create a new ObservableCollection and use it as the ListView items source.

Modify `MainPage.xaml` to add a `Button` control just above the closing `</VerticalStackLayout>` tag:

```
private ObservableCollection<string> myItems = new ObservableCollection<string>();

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

The `ObservableCollection` class is in the `System.Collections.ObjectModel` namespace. Use the Quick Actions menu to add a using directive to the top of your `MainPage.xaml.cs` file:

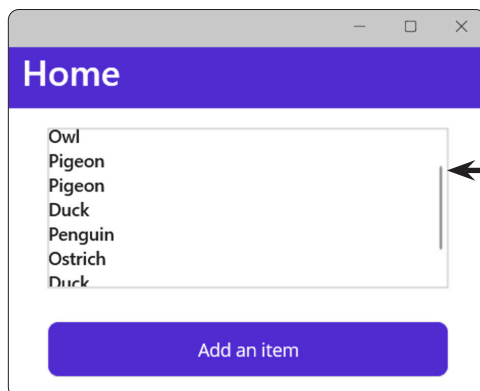
```
using System.Collections.ObjectModel;
```

3 Make the button's event handler method add random birds to the collection.

The `ObservableCollection` class has many of the same methods as the `List` class, including `Add`, `Sort`, and `Clear`. Modify the button's event handler method so it calls the `Add` method to add a random bird:

```
private void Button_Clicked(object sender, EventArgs e)
{
    myItems.Add(birds[Random.Shared.Next(0, birds.Length)]);
}
```

Now run your app and click the button a few times. Every time it adds a bird, the `ListView` updates itself automatically.



You used the HeightRequest property to set the height of the `ListView` control, so it adds a scroll bar if there are more items in the collection than fit in its height. Without this property, the `ListView` will automatically expand to fit the contents of the collection.

Try removing the `HeightRequest` property and running your app to see how the `ListView` expands when you add items.

Add your Card class to the project

Let's make your ListView control work with objects, not just strings—specifically, card objects. You created a Card class, Suits and Values enums, and a CardComparerByValue class. Now you'll reuse them in a MAUI app.



1 Add your Card class, CardComparerByValue class, Suits enum, and Values enum.

The ListView control will automatically call the ToString method for any item that it displays, so make sure you use the version of the Card class **that has the ToString method**. If you use Visual Studio's *Add Existing Item...* / *Add Existing File...* feature to add them, make sure they're all **in the MauiCards namespace**.

2 Enable Nullable in your project settings.

When you added your Card ComparerByValue class, did you notice that it had these warnings:

These are warnings, not errors,
so your code will still run.

 CS8632 The annotation for nullable reference types should only be used in code within a '#nullable' annotations context.
 CS8632 The annotation for nullable reference types should only be used in code within a '#nullable' annotations context.

C# has a lot of really useful features to detect null values, which you'll learn more about in Chapter 12. The .NET MAUI template doesn't have them turned on automatically, so you'll need to turn them on.

Right-click on the MauiCards project in the Solution Explorer window (make sure you're right-clicking on the **project** and not the solution that contains it) and **choose Edit ProjectFile** from the context menu.

This will open a file called MauiCards.csproj. Find the opening <PropertyGroup> tag and add this below it:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <Nullable>enable</Nullable>
```

If Nullable is not enabled, C# will give you warnings when you use ? to mark object references as nullable.

As soon as you add that line, Visual Studio will reload your project and your code will now compile. If it doesn't, make sure your code doesn't have any syntax errors, then close Visual Studio and reopen it.

3 Replace the Birds field with an ObservableCollection.

Delete the birds field from your MainPage class and **replace it** with this declaration:

```
private ObservableCollection<Card> myItems = new ObservableCollection<Card>();
```

4 Modify the Button_Clicked event handler to add a random card.

Here's the code for the event handler:

```
private void Button_Clicked(object sender, EventArgs e)
{
    myItems.Add(
        new Card((Values)Random.Shared.Next(1, 14), (Suits)Random.Shared.Next(4)));
}
```

Now run your app. When you click the button it adds a random card to the ListView.



Exercise

Create a Deck class that extends ObservableCollection.

You learned all about inheritance in Chapter 6. Now it's time to apply that knowledge to create a class that represents a deck of cards. **Add a Deck class** to your project. Make sure that it includes this using directive:

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

Make the Deck class **extend ObservableCollection<Card>** so it inherits all of the collection-related methods, including the Clear and Add methods.

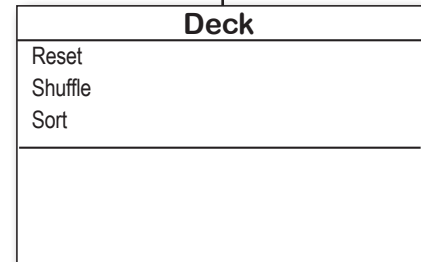
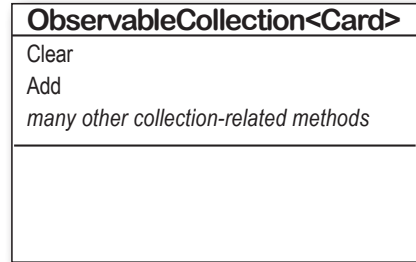
```
class Deck : ObservableCollection<Card>
{
    /// <summary>
    /// The constructor resets the 52-card deck
    /// </summary>
    public Deck() {
        Reset();
    }

    /// <summary>
    /// Clears the deck, then loops through suits and
    /// values to adds each card to the 52-card deck
    /// </summary>
    public void Reset() { ... }

    /// <summary>
    /// Creates a copy of the deck, clears the deck, and
    /// uses a while loop to move a random card from
    /// the copy to the deck and remove it from the copy
    /// </summary>
    public void Shuffle() { ... }

    /// <summary>
    /// Creates a copy of the deck, sorts it using
    /// CardComparerByValue, then moves each card from
    /// the copy back to the deck
    /// </summary>
    public void Sort() { ... }
}
```

← The Deck class is a subclass of ObservableCollection<Card>.



You can create a copy of the deck by using this List<T> constructor:

```
List<Card> copy = new
    List<Card>(this);
```

Then call Deck.Clear(), and use a loop to move cards from the copy back to the Deck.

Modify your MainPage.xaml and MainPage.xaml.cs

Replace the myItems declaration so it creates a Deck object instead of an array of strings with birds:

```
private Deck myItems = new Deck();
```

← This replaces the myItems field in MainPage.xaml.cs

Modify your XAML to add buttons to call the Deck methods:

```
<Button Text="Shuffle the deck" Clicked="Shuffle_Clicked"/>
<Button Text="Sort the deck" Clicked="Sort_Clicked"/>
<Button Text="Reset the deck" Clicked="Reset_Clicked"/>
<Button Text="Clear the deck" Clicked="Clear_Clicked"/>
```

Make of the event handlers for these buttons call the method in the Deck.

You'll add these new buttons to your page to call the Shuffle, Sort, Reset, and Clear methods on the Deck object.



Exercise Solution

Here's the Deck class, including the using directive to use the namespace ObservableCollection<T> is in:

```
using System.Collections.ObjectModel;

class Deck : ObservableCollection<Card>
{
    public Deck()
    {
        Reset();
    }

    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.Shared.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);
        }
    }
}
```

Does your ListView show “MauiCard.Card” instead of a card name? Make sure your Card class has a ToString method that returns Name.

This nested for loop goes through each of the suits, and for each suit it loops through all of the cards and adds them to the Deck.

This while loop picks a random card from the copy, adds it to the Deck, and then removes it from the copy, repeating until the copy is empty.

ObservableCollection doesn't have a Sort method—that's why you had to copy the cards to a List, sort it, and copy the cards back to the deck.

Here are the event handler methods for the four buttons you added to the XAML:

```
private void Shuffle_Clicked(object sender, EventArgs e) { myItems.Shuffle(); }

private void Sort_Clicked(object sender, EventArgs e) { myItems.Sort(); }

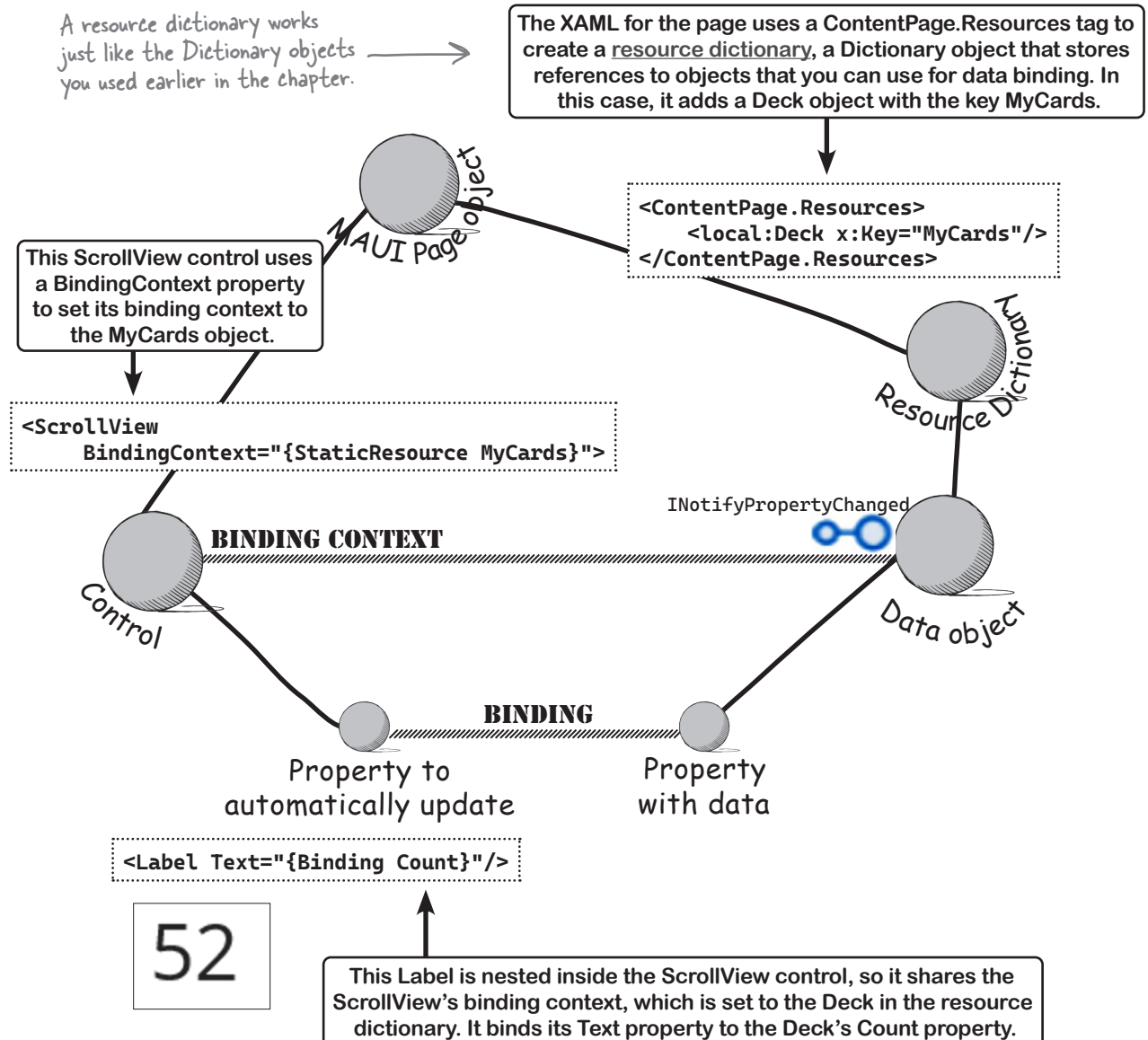
private void Reset_Clicked(object sender, EventArgs e) { myItems.Reset(); }

private void Clear_Clicked(object sender, EventArgs e) { myItems.Clear(); }
```


Use XAML to instantiate your objects for data binding

Your app uses C# code to create an instance of your Deck class and sets the ListView control's ItemsSource to set up data binding. And in previous projects, you set the binding context for your data binding by writing C# code to set the page's BindingContext property.

There's another way to do **data binding in MAUI directly in XAML** without setting properties in C# code. Seeing all of the binding directly the XAML code can be really useful when you're binding a lot of objects, so you don't have to jump back and forth between the XAML and C# code. Here's how it works:



Modify your app to use a resource dictionary

Let's get some practice using a resource dictionary by removing the Deck field from the C# code in your MainPage.xaml.cs file and instantiating it in your XAML instead.



Do this!

① Remove the Deck field from your code-behind.

Start by removing the myItems field and modifying the constructor to remove the line that sets the ListView control's ItemsSource property.

```
public partial class MainPage : ContentPage
{
    private Deck myItems = new Deck();

    public MainPage()
    {
        InitializeComponent();

        MyListView.ItemsSource = myItems;
    }
}
```

② Add the resource dictionary to your XAML.

Add this **xmlns:local** property to your ContentPage tag so it knows about the MauiCards namespace:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MauiCards"
    x:Class="MauiCards.MainPage">
```

Then add this ContentPage.Resources tag right underneath the ContentPage tag above the opening ScrollView tag:

```
    <ContentPage.Resources>
        <local:Deck x:Key="MyCards"/>
    </ContentPage.Resources>
```

Now your app has a resource dictionary that contains a Deck object with the key MyCards.

Declare a namespace in your <ContentPage> tag

Take a closer look at the XAML you just added. You added an xmlns:local property to the ContentPage tag that had the value "clr-namespace:MauiCards" – this defined "local" in your XAML to mean the MauiCards namespace. When you added a local:Deck resource tag, you told MAUI to instantiate a class called Deck that's in the "local" namespace, and your Deck class is in the MauiCards namespace. When you use a class from the same namespace as your app, you'll typically name it "local" – but you can put any name after "xmlns:" to look for classes in other namespaces.

③ Add a Label that binds to the Deck's Count property.

Since Deck extends the ObservableCollection class, it inherits the Count property—and you can use that with data binding, just like you did in Chapter 7. Add a HorizontalStackLayout with Label controls just below the opening <VerticalStackLayout> tag and above the Border and ListView tags:

In Chapter 7 you modified the code-behind in the MainPage.xaml.cs file to set the BindingContext property in the MainPage constructor. Now that you have a resource dictionary, you can use the **StaticResource markup** to set the binding context directly in XAML.

You're going to add a Label that displays this text: **52 cards in the deck**—except instead of the number 52, you'll use data binding. You can only bind an entire property, not part of one, so you'll actually need **two labels**: one to bind to the Count property, and one for the rest of the text—and notice how the second label has a space before the word “cards.”

You'll use a HorizontalStackLayout to put the two Label controls together so they're next to each other. Use the StaticResource markup to the first Label control's BindingContext property to bind it to the MyCards resource, then bind its Text property to its Count property:

```
<HorizontalStackLayout>
  <Label BindingContext="{StaticResource MyCards}" Text="{Binding Count}"/>
  <Label Text=" cards in the deck"/>
</HorizontalStackLayout>
```

Here's where you set the BindingContext using XAML to the object in the resource dictionary with “MyCards” as its key.

④ Set the ListView control's ItemsSource directly in the XAML.

Find the XAML for your ListView control and add ItemsSource property. Use the StaticResource markup to tell it that the source for the items in the list is your Deck resource.

```
<Border>
  <ListView x:Name="MyListView" HeightRequest="125"
    ItemsSource="{StaticResource MyCards}" />
</Border>
```

Using a Resource Directory Up Close



When you added the <ContentPage.Resources> tag to your ContentPage, you told MAUI to create a **resource dictionary** for the page. That's a dictionary to hold objects that the XAML elements on your page can access, and it works just like the Dictionary class you've been using—including the **TryGetValue method**.

You'll often need your code-behind to work with objects in the resource dictionary. For example, your app has buttons that need to call methods on the Deck resource stored in the dictionary. You can access the resource dictionary using the ContentPage's Resources property. You'll use this code to fetch the Deck from the dictionary:

```
if (Resources.TryGetValue("MyCards", out object myCards) && myCards is Deck deck)
    // now the Deck variable contains a reference to the MyDeck resource
```

Take a close look at what's happening in this line of code. It uses the Resources property to access the resource dictionary, then calls its TryGetValue method to get the Deck reference. If it finds the deck reference, it uses it to cast it to a Deck. Take a little time to understand what's going on here.

Modify the event handlers to use the resource dictionary

We just showed you a line of code that gets a reference to the Deck object in the page's resource dictionary. Let's use that code to get your app's button event handlers to work with the Deck object in the page's resource dictionary.

- ★ Remove the `myItems` field from your `MainPage` class and **add a method called `getDeckFromResources`** that uses `Resources.TryGetValue` to get the reference to the Deck object from the Resources dictionary.
- ★ Modify the four Button event handlers to **replace `myItem` with `getDeckFromResources()`**.

Once you've done that, the code-behind for your `MainPage.xaml.cs` file should look like this:

```
namespace MauiCards;
```

```
public partial class MainPage : ContentPage  
{
```

```
    public MainPage()  
    {
```

```
        InitializeComponent();  
    }
```

```
    private Deck getDeckFromResources()  
    {
```

```
        if (Resources.TryGetValue("MyCards", out object myCards) && myCards is Deck deck)  
            return deck;
```

```
        else
```

```
            return new Deck();  
    }
```

Resources.TryGetValue returns an Object reference, so we need to use `is` to safely cast it to a Deck reference. This method should never reach this statement—but we included it so the app doesn't throw an exception just in case it does. Did we make the right choice?

```
    private void Button_Clicked(object sender, EventArgs e)  
    {
```

```
        getDeckFromResources().Add(  
            new Card((Values)Random.Shared.Next(1, 14), (Suits)Random.Shared.Next(4));  
        )  
    }
```

```
    private void Shuffle_Clicked(object sender, EventArgs e) {
```

```
        getDeckFromResources().Shuffle();  
    }
```

```
    private void Sort_Clicked(object sender, EventArgs e) {
```

```
        getDeckFromResources().Sort();  
    }
```

```
    private void Reset_Clicked(object sender, EventArgs e) {
```

```
        getDeckFromResources().Reset();  
    }
```

```
    private void Clear_Clicked(object sender, EventArgs e) {
```

```
        getDeckFromResources().Clear();  
    }
```

```
}
```

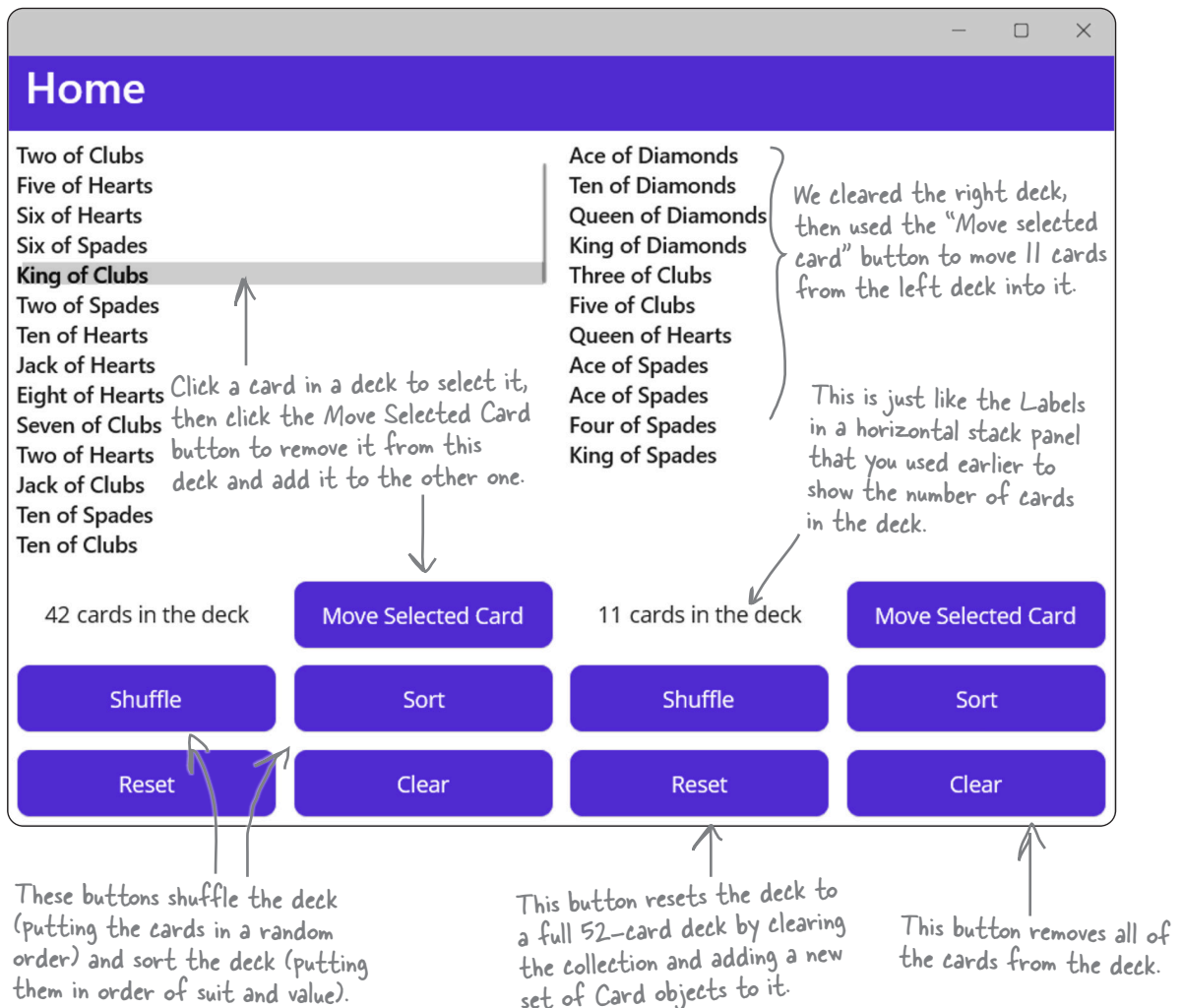
The `getDeckFromResources` method uses `Resources.TryGetValue` to get the Deck object from the page's resource dictionary, then uses `"is"` to safely cast it to a Deck reference.

The `myItems` field that you removed from the `MainPage` class has a reference to a Deck object. The `getDeckFromResources` method returns a reference to the Deck in the page's resource dictionary, so replacing `myItems` with a call to that method makes the page work just like it did before.

User what you've learned to build an app with two decks

You've just used a `ListView` control, an `ObservableCollection`—which you extended to create your `Deck` class—created a resource dictionary with a `Deck` object, done all of the data binding directly in your XAML code, and written C# code to access that object in the resource dictionary.

This next exercise is about ***really getting those new C# concepts and tools into your brain***. You'll create an app that has two decks of cards, with buttons that let you shuffle, sort, reset, and clear the decks. You'll also take advantage of a feature in the `ListView` control that lets you select an item in the list to create buttons to move cards between the decks.



In this next exercise, you'll take the ideas and tools you just used and apply them to a new project. This is a great way to get them to stick in your brain.



Exercise

You've just learned about ListView controls and more about data binding. Combined with what you've learned with what you already know about building .NET MAUI apps to build an app that manages two decks.

Create a .NET MAUI project called *TwoDecks* and add Deck and the other card-related classes and enums

Create a new project. Right-click on the project in the Solution Explorer and choose the "Add >> Existing Item... Shift+Alt+A" (Windows) or "Add >> Existing Files... ⌘+⌘+A" (macOS), then navigate to your MauiCards project folder and add Deck.cs, which holds the Deck class. Then do the same for the Card class, Suits enum, Values enum, and the CardComparerByValue class. Make sure you're using the version of the Card class that has a ToString method.

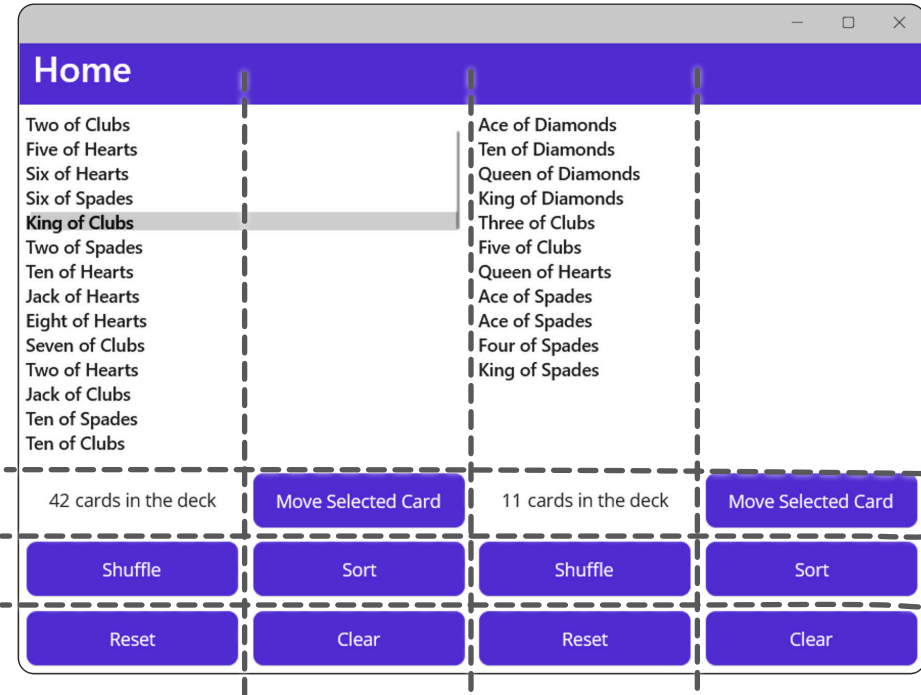
If CardComparerByValue class has warnings that "the annotation for nullable reference types should only be used in code within a '#nullable' annotations context," edit the project file and add this to the <PropertyGroup> section:

```
<Nullable>enable</Nullable>
```

Rebuild your project and make sure that it does not have any warnings.

Use a grid with four columns and four rows to lay out the page

Delete everything inside the <ContentPage> tag and replace it with a <Grid> with four columns and four rows. The four columns are equal widths, so none of the <ColumnDefinition/> tags should have a Width property. Don't add a Height property to the top <RowDefinition/> tag, but give the bottom three a Height="Auto" property.



Adding Height="Auto" to the bottom three rows causes them to automatically adjust to fit whatever controls are in the row. That will cause the top row to automatically expand to fill the rest of the page. If you resize the window, the bottom rows will stay the same height, and the top row will adjust to fit the new height.



Exercise

Add a resource dictionary to your page with two Deck objects

In the last project, you added a resource dictionary to your page with a Deck object. You'll add a resource dictionary to this page too, but this time it will create two Deck objects: one reference will have the key `LeftDeck`, and the other reference will have the key `RightDeck`.

Since you copied `Deck.cs` and the other card-related files from the `MauiCards` project, they should all have the namespace `MauiCards` declaration at the top. **Add a `xmlns:mauiCards` property** to your `ContentPage` tag, which will let you use the **`mauiCards:`** prefix to create objects in the `MauiCards` namespace when you declare your resource dictionary. Replace the XAML in `MainPage.xaml.cs` with this, then add controls nested in the `Grid`:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:mauiCards="clr-namespace:MauiCards"
  x:Class="TwoDecks.MainPage">
```

You need the `mauiCards:` prefix because the `Deck` object is in the `MauiCards` namespace.

```
<ContentPage.Resources>
  <mauiCards:Deck x:Key="LeftDeck"/>
  <mauiCards:Deck x:Key="RightDeck"/>
</ContentPage.Resources>
```

Here's where you'll declare your resource dictionary with two `Deck` references. They have the keys `LeftDeck` and `RightDeck`.

```
<Grid>
```

The contents of the page will go here

```
</Grid>
```

This is where the controls nested in the `Grid` will go. You'll also need to add the row and column definitions here.

```
</ContentPage>
```

Modify your code-behind to get Deck references from the resource dictionary so the buttons can use them

You'll be using the `Deck` type in your code, so make sure you add this using directive at the top of `MainPage.xaml.cs`:

```
using MauiCards;
```

Add this method to your code-behind. You had a really similar method in your last project, and this one is almost identical—the only difference is that it takes the key as a parameter so you can get either deck from the dictionary.

```
private Deck getDeckFromResources(String key)
{
    if (Resources.TryGetValue(key, out object resource) && resource is Deck deck)
        return deck;
    else
        return new Deck();
}
```

There's one more piece of the puzzle: you need a way to figure out what card is selected, so you can make the buttons that move a card from one deck to the other work. Use `x.Name` to **name your `ListView` controls** `LeftListView` and `RightListView`. If a card is selected in the left `ListView`, **`LeftListView.SelectedItem` is `Card` `card`** will return true. If it does, call `getDeckFromResources("RightDeck").Add(card)` to add the card to the right deck, then call `getDeckFromResources("LeftDeck").Remove(card)` to remove it from the left deck. Then do the reverse for the button to move a card from the right deck to the left, adding it to the left deck and removing it from the right.



Exercise Solution

Here's the XAML for the page in MainPage.xaml.cs.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:mauiCards="clr-namespace:MauiCards"
  x:Class="TwoDecks.MainPage">

  <ContentPage.Resources>
    <mauiCards:Deck x:Key="LeftDeck"/>
    <mauiCards:Deck x:Key="RightDeck"/>
  </ContentPage.Resources>

  <Grid>

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

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

    <ListView x:Name="LeftListView" ItemsSource="{StaticResource LeftDeck}"
      Grid.ColumnSpan="2" Margin="5" />

    <ListView x:Name="RightListView" ItemsSource="{StaticResource RightDeck}"
      Grid.Column="2" Grid.ColumnSpan="2" Margin="5"/>

    <HorizontalStackLayout Grid.Row="1"
      VerticalOptions="Center" HorizontalOptions="Center">
      <Label BindingContext="{StaticResource LeftDeck}" Text="{Binding Count}"/>
      <Label Text=" cards in the deck"/>
    </HorizontalStackLayout>

    <Button Text="Move Selected Card" Grid.Row="1" Grid.Column="1"
      SemanticProperties.Hint="Moves the selected card from left to right"
      Margin="5" Clicked="moveLeftDeck_Clicked" />

  </Grid>
</ContentPage>
```

This xmlns property in the ContentPage tag sets up the "mauiCards" namespace, which contains Deck and the other card-related classes.

Your page has a resource dictionary with two Deck objects, with the keys LeftDeck and RightDeck.

The page has four equal sized columns.

The ListView is in the top row, and the buttons are in the bottom four rows. Setting the button row heights to "Auto" causes them to shrink so they're exactly the size of the buttons, leaving the top row to expand to fill up the rest of the grid.

The two ListView controls span two columns, so they're twice as wide as each button.

This label uses the BindingContext property to bind to the left Deck object, and binds its Text property to its Count property.



Exercise Solution

```

<Button Text="Shuffle" Grid.Row="2"
        SemanticProperties.Hint="Shuffles the left deck"
        Margin="5" Clicked="shuffleLeftDeck_Clicked" />

<Button Text="Sort" Grid.Row="2" Grid.Column="1"
        SemanticProperties.Hint="Sorts the left deck"
        Margin="5" Clicked="sortLeftDeck_Clicked" />

<Button Text="Reset" Grid.Row="3"
        SemanticProperties.Hint="Resets the left deck"
        Margin="5" Clicked="resetLeftDeck_Clicked" />

<Button Text="Clear" Grid.Row="3" Grid.Column="1"
        SemanticProperties.Hint="Clears the left deck"
        Margin="5" Clicked="clearLeftDeck_Clicked" />

<HorizontalStackLayout Grid.Row="1" Grid.Column="2"
        VerticalOptions="Center" HorizontalOptions="Center">
    <Label BindingContext="{StaticResource RightDeck}" Text="{Binding Count}"/>
    <Label Text=" cards in the deck"/>
</HorizontalStackLayout>

<Button Text="Move Selected Card" Grid.Row="1" Grid.Column="3"
        SemanticProperties.Hint="Moves the selected card from right to left"
        Margin="5" Clicked="moveRightDeck_Clicked" />

<Button Text="Shuffle" Grid.Row="2" Grid.Column="2"
        SemanticProperties.Hint="Shuffles the right deck"
        Margin="5" Clicked="shuffleRightDeck_Clicked" />

<Button Text="Sort" Grid.Row="2" Grid.Column="3" Margin="5"
        SemanticProperties.Hint="Sorts the right deck"
        Clicked="sortRightDeck_Clicked" />

<Button Text="Reset" Grid.Row="3" Grid.Column="2" Margin="5"
        SemanticProperties.Hint="Resets the right deck"
        Clicked="resetRightDeck_Clicked" />

<Button Text="Clear" Grid.Row="3" Grid.Column="3" Margin="5"
        SemanticProperties.Hint="Clears the right deck"
        Clicked="clearRightDeck_Clicked" />

</Grid>

```

```
</ContentPage>
```

These These
are bottom four
buttons on the left
side of the page.
They're in the left
two columns.

Did you remember
to include semantic
properties? Accessibility
is really important, and
you should get in the
habit of making your
apps accessible.

The labels and buttons on the right side of the
page are almost the same as the ones on the left.



Exercise Solution

Here's the C# from the code-behind for the page. Make sure to include the class declaration, constructor, and using MauiCards; directive.

```
private Deck getDeckFromResources(String key)
{
    if (Resources.TryGetValue(key, out object resource) && resource is Deck deck)
        return deck;
    else
        return new Deck();
}
```

We gave you this method to get a Deck from the resources.

```
private void shuffleLeftDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("LeftDeck").Shuffle();
}
```

```
private void sortLeftDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("LeftDeck").Sort();
}
```

```
private void clearLeftDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("LeftDeck").Clear();
}
```

```
private void resetLeftDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("LeftDeck").Reset();
}
```

```
private void moveLeftDeck_Clicked(object sender, EventArgs e)
{
    if (LeftListView.SelectedItem is Card card)
    {
        getDeckFromResources("RightDeck").Add(card);
        getDeckFromResources("LeftDeck").Remove(card);
    }
}
```

These Clicked event handlers work just like the ones in your previous app.

Here's the Clicked event handler for the button that moves the selected card from the left deck to the right one. It works exactly the way we described in the instructions.

```
private void shuffleRightDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("RightDeck").Shuffle();
}
```

```
private void sortRightDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("RightDeck").Sort();
}
```

```
private void clearRightDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("RightDeck").Clear();
}
```

```
private void resetRightDeck_Clicked(object sender, EventArgs e)
{
    getDeckFromResources("RightDeck").Reset();
}
```

```
private void moveRightDeck_Clicked(object sender, EventArgs e)
{
    if (RightListView.SelectedItem is Card card)
    {
        getDeckFromResources("LeftDeck").Add(card);
        getDeckFromResources("RightDeck").Remove(card);
    }
}
```

The event handlers for the buttons on the right work just like the ones on the left.