O'REILLY®

~~Fourth~~ Fifth Edition

Head First C#

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

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.

Andrew Stellman
& Jennifer Greene

This is the .NET MAUI project from Chapter 3. You'll create an app that picks random cards, using a CardPicker class that you created earlier in the chapter (which we'll include in this PDF).

A Brain-Friendly Guide

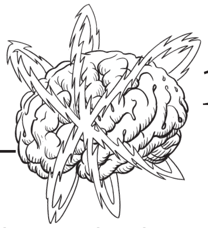
Your finished CardPicker class

Here's the code for your finished CardPicker class. It lives inside the PickRandomCards namespace (which matches the name of the project) and has the methods we just added:

If you're using Visual Studio for Mac, your CardPicker class will have the public access modifier instead of internal. Also, we deleted the constructor method that Visual Studio for Mac added. It's okay if you didn't delete it.

```
namespace PickRandomCards
{
    internal class CardPicker
    {

        public static string[] PickSomeCards(int numberOfCards)
        {
            string[] pickedCards = new string[numberOfCards];
            for (int i = 0; i < numberOfCards; i++)
            {
                pickedCards[i] = RandomValue() + " of " + RandomSuit();
            }
            return pickedCards;
        }

        private static string RandomSuit()
        {
            // get a random number from 1 to 4
            int value = Random.Shared.Next(1, 5);
            // if it's 1 return the string Spades
            if (value == 1) return "Spades";
            // if it's 2 return the string Hearts
            if (value == 2) return "Hearts";
            // if it's 3 return the string Clubs
            if (value == 3) return "Clubs";
            // if we haven't returned yet, return the string Diamonds
            return "Diamonds";
        }

        private static string RandomValue()
        {
            int value = Random.Shared.Next(1, 14);
            if (value == 1) return "Ace";
            if (value == 11) return "Jack";
            if (value == 12) return "Queen";
            if (value == 13) return "King";
            return value.ToString();
        }
    }
}
```

We added these comments to help you understand how the RandomSuit method works. Try adding similar comments to the RandomValue method that explain how it works.



Brain Power

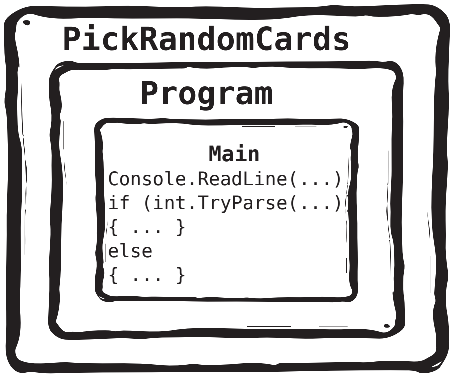
You used the `public` and `static` keywords when you added PickSomeCards. Visual Studio kept the `static` keyword when it generated the methods, and declared them as `private`, not `public`. What do you think these keywords do?

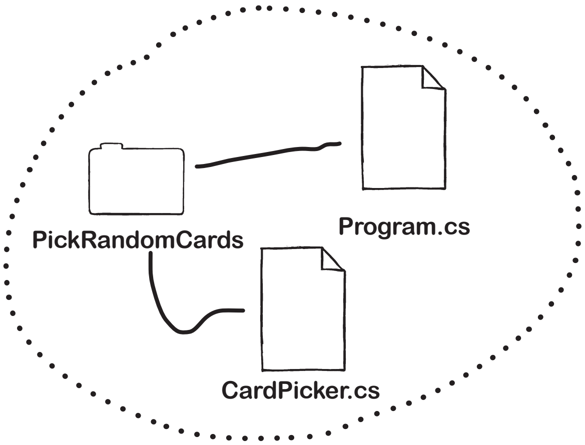
Let's build an app that picks random cards

In the first project in this chapter, you're going to build a console app called PickRandomCards that lets you pick random playing cards.

Let's use it as a way to **start using classes**. Here's what its structure will look like:

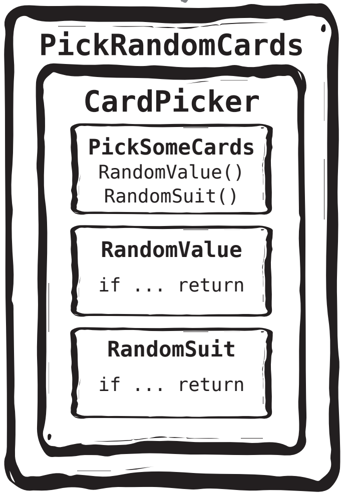
You're going to create a Console App that has a Main method instead of top-level statements (which we'll talk more about). Your Program.cs file will contain a class. That class will have a method called Main, which is the first thing that gets run when you start the app.





When you create your app you'll call it PickRandomCards, so Visual Studio will create a namespace for you that matches the name of the app.

Your Main method will have all the code that communictes with the user, displaying text and getting input. The code that has to do with picking random cards will be in a class called CardPicker.



The CardPicker class doesn't have any fields, and that's okay! We'll talk more about fields later in the chapter.

Build a paper prototype for a classic game

Paper prototypes are really useful for helping you figure out how a game will work before you start building it, which can save you a lot of time. There's a fast way to get started building them—all you need is some paper and a pen or pencil. Start by choosing your favorite classic game. Platform games work especially well, so we chose one of the **most popular, most recognizable** classic video games ever made... but you can choose any game you'd like! Here's what to do next.

Draw this!

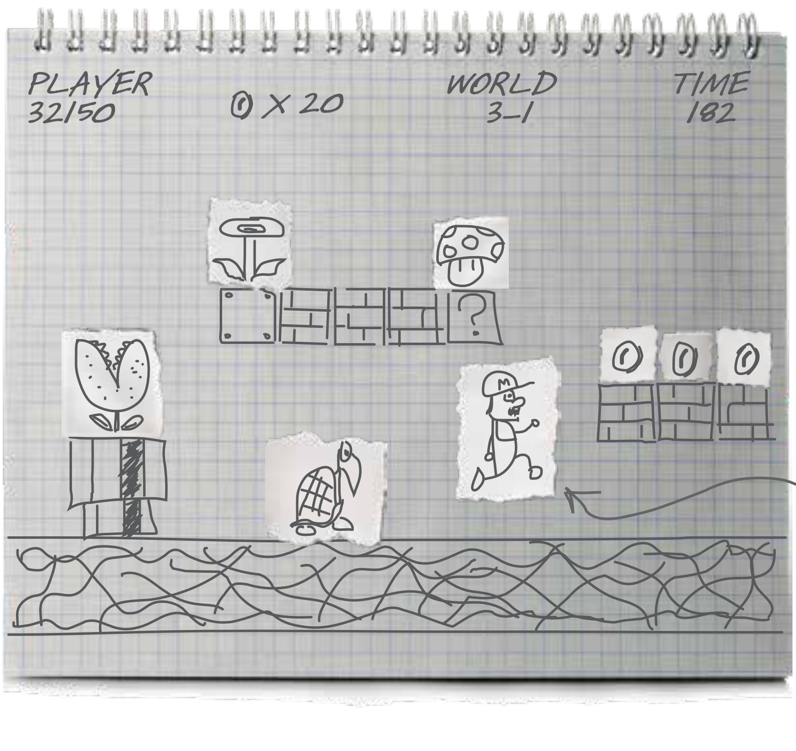
1. **Draw the background on a piece of paper.** Start your prototype by creating the background. In our prototype, the ground, bricks, and pipe don't move, so we drew them on the paper. We also added the score, time, and other text at the top.

2. **Tear small scraps of paper and draw the moving parts.** In our prototype, we drew the characters, the piranha plant, the mushroom, the fire flower, and the coins on separate scraps. If you're not an artist, that's absolutely fine! Just draw stick figures and rough shapes. Nobody else ever has to see this!

3. **"Play" the game.** This is the fun part! Try to simulate player movement. Drag the player around the page. Make the non-player characters move too. It helps to spend a few minutes playing the game, then go back to your prototype and see if you can really reproduce the motion as closely as possible. (It will feel a little weird at first, but that's OK!)

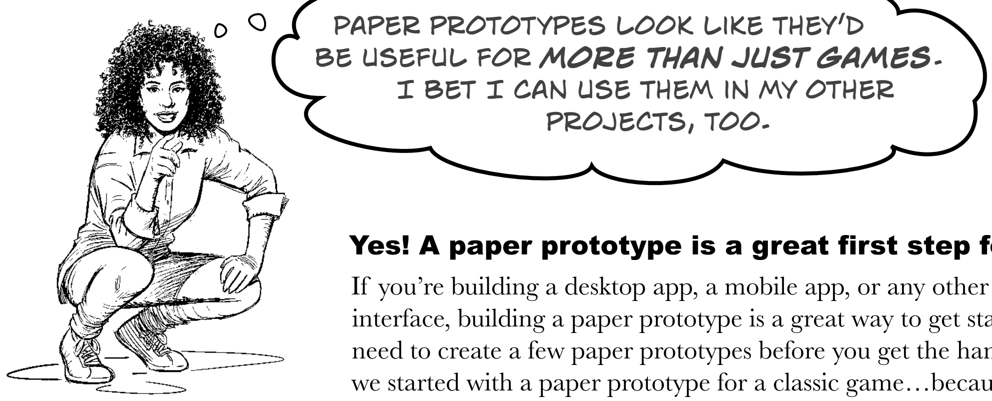
The text at the top of the screen is called the HUD, or head-up display. It's usually drawn on the background in a paper prototype.

When the player catches a mushroom he grows to double his size, so we also drew a small character on a separate scrap of paper.



The ground, bricks, and pipe don't move, so we drew them on the background paper. There's no rule about what goes on the background and what moves around.

The mechanics of how the player jumps were really carefully designed. Simulating them in a paper prototype is a valuable learning exercise.


PAPER PROTOTYPES LOOK LIKE THEY'D BE USEFUL FOR *MORE THAN JUST GAMES*. I BET I CAN USE THEM IN MY OTHER PROJECTS, TOO.

All of the tools and ideas in "Game design... and beyond" sections are important skills that go way beyond just game development—but we've found that they're easier to learn when you try them with games first.

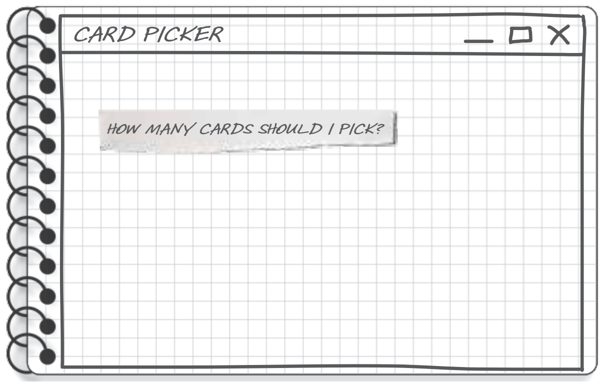
Yes! A paper prototype is a great first step for any project.

If you're building a desktop app, a mobile app, or any other project that has a user interface, building a paper prototype is a great way to get started. Sometimes you need to create a few paper prototypes before you get the hang of it. That's why we started with a paper prototype for a classic game…because that's a great way to learn how to build paper prototypes. **Prototyping is a really valuable skill for any kind of developer**, not just a game developer.

Sharpen your pencil

In the next project, you'll create a MAUI app that uses your CardPicker class to generate a set of random cards. In this paper-and-pencil exercise, you'll build a paper prototype of your app to try out various design options.

Start by drawing the window frame on a large piece of paper and a label on a smaller scrap of paper.



Your app needs to include a Button control with the text "Pick some cards" and a Label control to display the cards somewhere in the window.

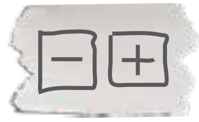
PICK SOME CARDS

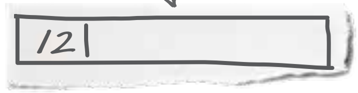
4 OF HEARTS
2 OF DIAMONDS
KING OF SPADES
ACE OF HEARTS
7 OF CLUBS
10 OF SPADES
JACK OF CLUBS
9 OF HEARTS
9 OF DIAMONDS
3 OF CLUBS
ACE OF SPADES

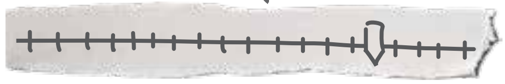
Next, draw a bunch of different types of controls on more small scraps of paper. Drag them around the window and experiment with ways to fit them together. What design do you think works best? There's no single right answer—there are lots of ways to design any app.

Your app needs a way for the user to choose the number of cards to pick. Try drawing an Entry control that they can use to type numbers into your app.

Try drawing Slider and Stepper controls, too. Can you think of other controls that you've used to input numbers into apps before? Maybe a Picker? Get creative!



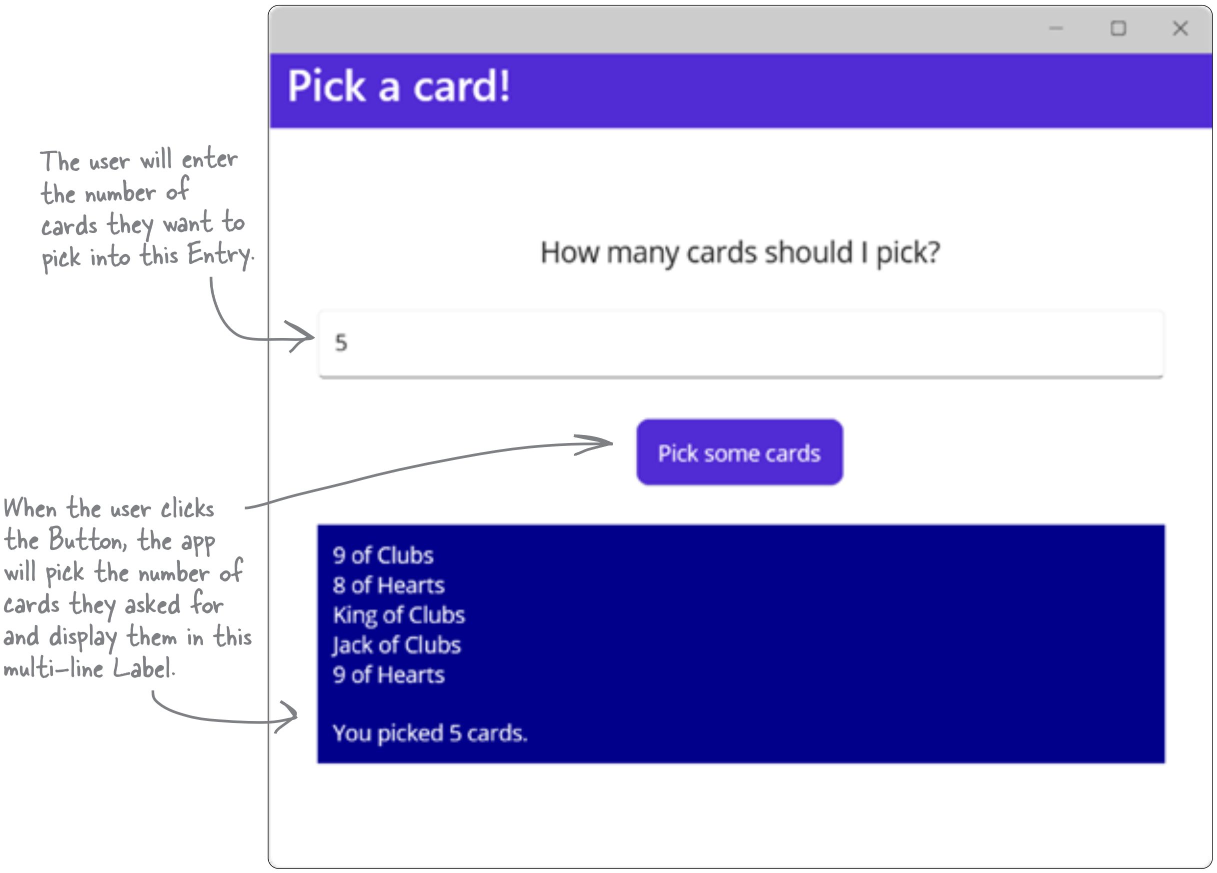




Build a MAUI version of your random card app

All of the code for picking random cards is conveniently organized into a class called CardPicker. Now you'll **reuse that class** in a .NET MAUI app.

Here's how the app will work.


The user will enter the number of cards they want to pick into this Entry.
When the user clicks the Button, the app will pick the number of cards they asked for and display them in this multi-line Label.
Pick a card!
How many cards should I pick?
5
Pick some cards

```
9 of Clubs
8 of Hearts
King of Clubs
Jack of Clubs
9 of Hearts

You picked 5 cards.
```

Make your app accessible!

Accessibility is really important—and paying attention to accessibility is a great way to focus on important skills, like understanding your users and their needs.

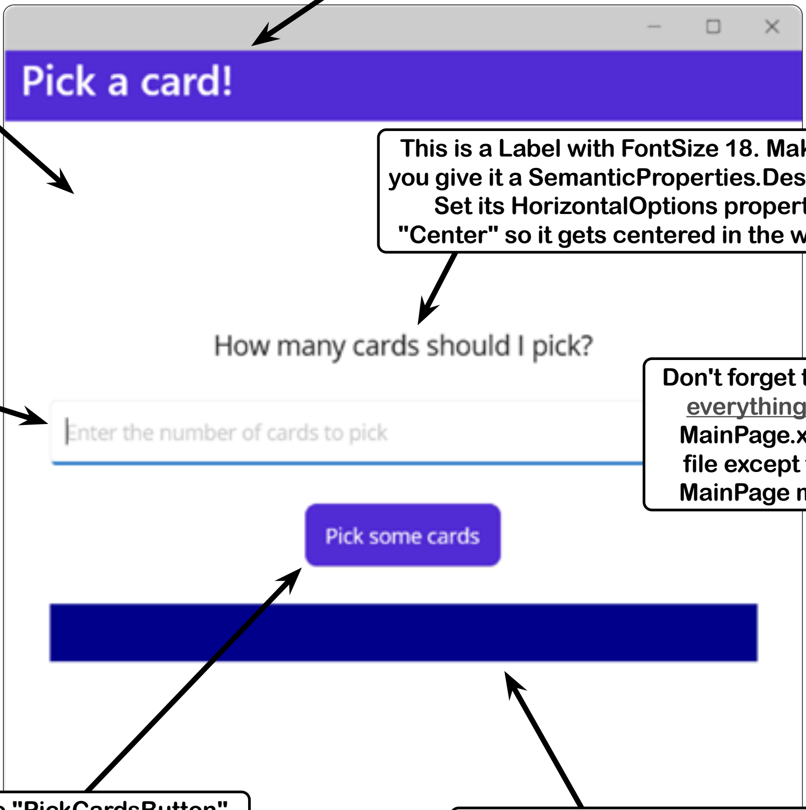
- ★ The Label and Entry controls each have a **SemanticProperties. Description property** so the screen reader will read it out loud.
- ★ The Button control has a **SemanticProperties.Hint property** because the screen reader will read the contents of the button but we still want to give people who use accessibilty tools additional context for the control.

Exercise

You already have the tools you need to create the XAML for the MAUI card picker app! In this exercise, you'll use what you learned about XAML in the first two chapters to create the main page for your app. You may need to go back to the XAML code you wrote in Chapter 2 to see how you added controls to your page.

Create a new .NET MAUI app called **PickRandomCardsMAUI**. Edit the MainPage.xaml.cs file to delete the controls inside the VerticalStackLayout (just like you did in Chapter 2), then add the controls for your card picker app.

Bonus: Edit the AppShell.xaml file to set the page title! We haven't showed you how to do that yet—can you figure it out?


Pick a card!
How many cards should I pick?
Enter the number of cards to pick
Pick some cards

Can you figure out how to set the page title? Open the AppShell.xaml file, look for a <ShellContent> tag, and change its Title property.

The ContentPage contains a ScrollView, which contans a VerticalStackLayout, just like your last MAUI project.

This is a Label with FontSize 18. Make sure you give it a SemanticProperties.Description. Set its HorizontalOptions property to "Center" so it gets centered in the window.

This is an Entry. Give it a placeholder and a semantic description for accessibility, and use the x:Name property to name it "NumberOfCards" so your code can read its value.

Don't forget to delete everything in the MainPage.xaml.cs file except for the MainPage method.

This Button has the name "PickCardsButton" and a Clicked event handler method called PickCardsButton_Clicked. Make sure the event handler method is created in MainPage.xaml.cs. Set its HorizontalOptions property to "Center" and give it a SemanticProperties.Hint property.

This is a multi-line Label with the name "PickedCards" and a Padding property to 20 so it has some space around the text. It has white text on a dark blue background. Make sure it has a SemanticProperties.Description.

Peeking at the solution is not cheating! It's actually a great way to get these ideas to stick in your brain.

Exercise Solution

Your MainPage.xaml.cs file should have a `public MainPage()` method that calls InitializeComponent and an empty Clicked event handler method and nothing else.

Here's the XAML for the contents of MainPage.xaml (we didn't include the outer <ContentPage> tag):

```
<ScrollView>
    <VerticalStackLayout
        Spacing="25"
        Padding="30,0"
        VerticalOptions="Center">

        <Label
            Text="How many cards should I pick?"
            SemanticProperties.Description="How many cards should I pick?"
            FontSize="18"
            HorizontalOptions="Center" />

        <Entry
            x:Name="NumberOfCards"
            SemanticProperties.Description="Enter the number of cards to pick"
            Placeholder="Enter the number of cards to pick" />

        <Button
            x:Name="PickCardsButton"
            Text="Pick some cards"
            SemanticProperties.Hint="Picks some cards"
            Clicked="PickCardsButton_Clicked"
            HorizontalOptions="Center" />

        <Label x:Name="PickedCards" Padding="20"
                TextColor="White" BackgroundColor="DarkBlue"
                SemanticProperties.Description="Shows the cards that were picked" />

    </VerticalStackLayout>
</ScrollView>
```

These are the same ScrollView and VerticalStackLayout tags that Visual Studio created using the .NET MAUI template.

The HorizontalOptions property centers the label on the page. Try the other options—do you like the way they look better?

You gave the Entry, Button, and Label controls names that you'll use in your C# code.

Make sure Visual Studio added the PickCardsButton_Clicked event handler method that gets called when the button is clicked. You'll use it in the second part of this project.

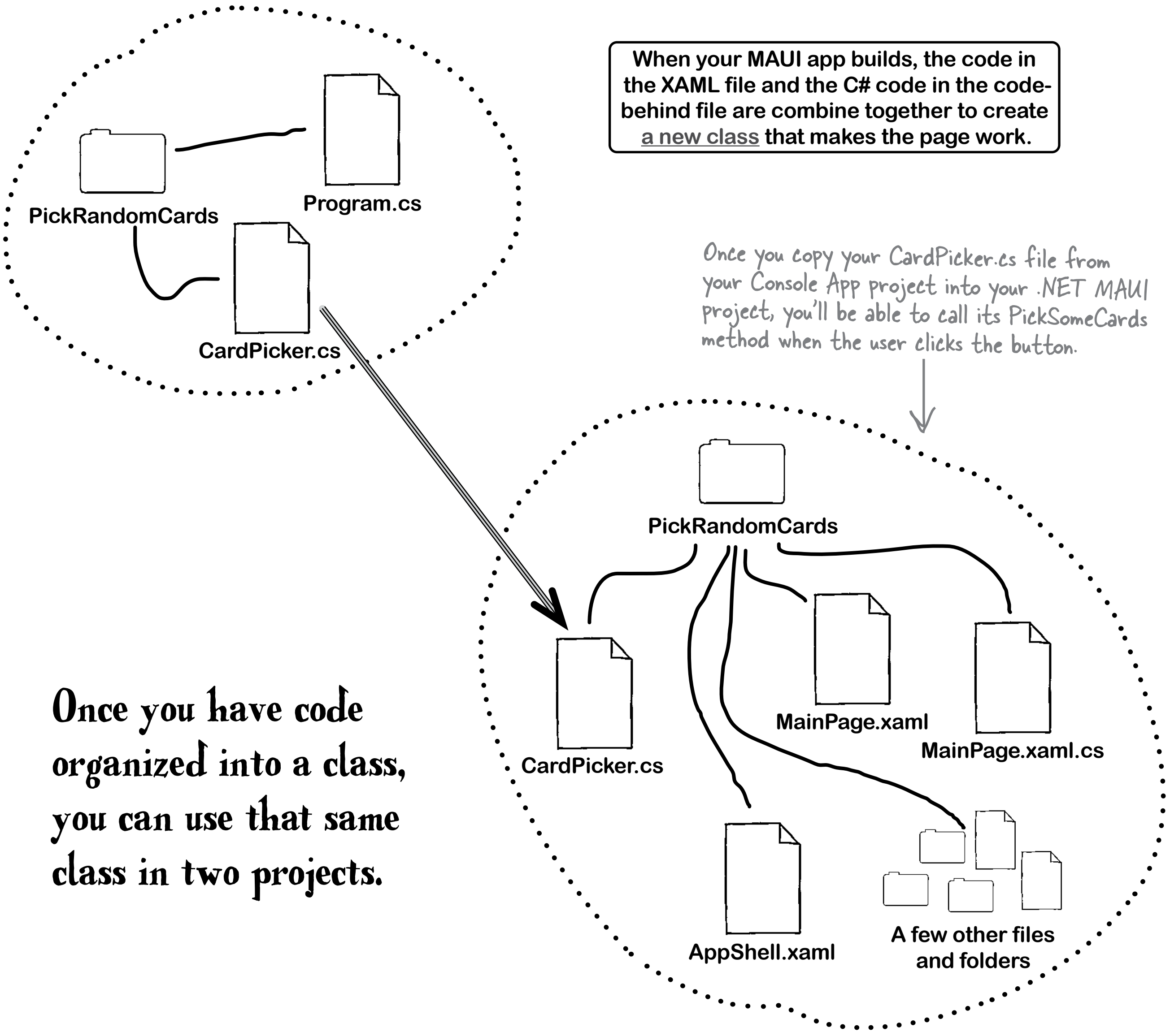
We made this change to AppShell.xaml to set the title of the page to "Pick a card!":

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    ...
    Shell.FlyoutBehavior="Disabled" BackgroundColor="Red">

    <ShellContent
        Title="Pick a card!"
        ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />

</Shell>
```

The XAML in AppShell.xml tells your MAUI app what to do when it first starts up. The ShellContent's Route property tells it to load the page in your MainPage.xaml file. Try setting the BackgroundColor of the outer Shell tag—what does that change in the app?

Make your MAUI app pick random cards

You've got an app that looks like it's supposed to, and that's a great start! In the second part of this project, you'll make it work, so when the user enters a number and clicks the button it picks random cards. That's where your CardPicker class comes in. You've already created a class that picks random cards. Now you just need to **copy that class into your new APP**. Once it's copied, you'll be able to make your button's event handler method call the PickSomeCards method in the CardPicker class.



When your MAUI app builds, the code in the XAML file and the C# code in the code-behind file are combine together to create a new class that makes the page work.

Once you copy your CardPicker.cs file from your Console App project into your .NET MAUI project, you'll be able to call its PickSomeCards method when the user clicks the button.

Once you have code organized into a class, you can use that same class in two projects.

Reuse your the CardPicker class

You took the time to put all of the random card picking code into a convenient class. Now it's time to take advantage of that class by **reusing it in your new MAUI app**.

Do this!

1 **Choose 'Add Existing Item' or 'Add Existing Files' in Visual Studio.**
You created a file called CardPicker.cs in your PickRandomCards console app. Now you'll tell Visual Studio to **add that class file** to your MAUI project, which will cause it to copy the file into your MAUI app's project folder..

- On Windows, right-click on the project in the the Solution Explorer window and choose Add >> Existing Item... (Shift+Alt+A), or choose Add Existing Item from the Project menu.
- On macOS, right-click on the project in the Solution window and choose Add >> Existing Files... (⌥⌘A). Visual Studio will ask whether you want to copy, move, or add a link to the file. Make sure you copy the file so it exists in the MAUI project folder.

2 **Find your CardPicker.cs file and add it to your project.**
Visual Studio will pop up a folder explorer window. Navigate to the folder with your PickACard console app and **double-click on CardPicker.cs**. You should now see CardPicker in the Solution Explorer.

3 **Try to use your CardPicker class in the MainPage.xaml.cs code.**
Open MainPage.xaml.cs. Edit the PickCardsButton_Clicked event handler method and try add a statement that calls your CardPicker.PickSomeCards method.

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void PickCardsButton_Clicked(object sender, EventArgs e)
    {
        CardPicker.
    }
}
```

Here's the event handler method that Visual Studio added to your C# code when you added a Clicked event handler to the XAML for the button.

Hold on—something's wrong!

When you start typing the statement to call CardPicker.PickSomeCards, Visual Studio doesn't pop up its normal IntelliSense window, and there's a squiggly error line under CardPicker.

Why do you think Visual Studio is treating CardPicker like that?

Add a using directive to use code in another namespace

When you created your CardPicker class, it added a **namespace declaration** at the top that matches the name you chose for the the project, followed by a pair of curly brackets that contain the entire class:

```
namespace PickRandomCards
{
   ... your class is in the PickRandomCards namespace ...
}
```

Compare that to the code at the top of your MainPage.xaml.cs file in your MAUI app:

```
namespace PickRandomCardsMAUI;

public partial class MainPage : ContentPage
{
  ... your MAUI app's code is in the PickRandomCardsMAUI namespace ...
}
```

The reason your MainPage class can't access the methods in your CardPicker class is because they're in different namespaces. Luckily, C# has an easy way to deal with this. You'll add a **using directive** in your code that calls the methods in CardPicker—that's a special line that you put at the top of a class file to tell it to use code in another namespace.

Add this!

Add this line to the top of your MainPage.xaml.file. If you chose a different name for your console app, replace PickRandomCards with then namespace in your CardPicker.cs file.

```
using PickRandomCards;
```

This using directive will let you add code to your MainPage.xaml.cs file that uses classes in the PickRandomCards namespace—so now you can can write code that calls methods in your CardPicker class. You might see other using directives at the top of the file, too.

Now go back to the event handler method for your button. Start typing `CardPicker.` like you did before. Now Visual Studio will pop up its IntelliSense window, just like you'd expect it to.

Exercise

Here's a C# coding challenge for you! Now that you added the using directive to the top of your MainPage.xaml.cs file, you can call the your CardPicker class. Can you finish the event handler method to make your app work?

- The first thing the method does is call int.TryParse to convert NumberOfCards.Text to a number.
- If the number is valid, it calls CardPicker.PickSomeCards just like in your console app. If it isn't, it makes the PickedCards label display a message: `PickedCards.Text = "Please enter a valid number.";`
- Instead of writing to the console, it sets PickedCards.Text to a string value to make text appear in the PickedCards Label control. You can clear the text in PickedCards like this: `PickedCards.Text = String.Empty;`
- After it clears the PickedCards label, it uses a foreach loop that works just like the one in your console app.
- Add this statement after the foreach loop to tell the user how many cards they picked:
  `PickedCards.Text += Environment.NewLine + "You picked " + numberOfCards + " cards.";`

Exercise Solution

Heres the finished event handler method.

```
private void PickCardsButton_Clicked(object sender, EventArgs e)
{
  if (int.TryParse(NumberOfCards.Text, out int numberOfCards))
  {
    string[] cards = CardPicker.PickSomeCards(numberOfCards);
    PickedCards.Text = String.Empty;
    foreach (string card in cards)
    {
      PickedCards.Text += card + Environment.NewLine;
    }
    PickedCards.Text += Environment.NewLine + "You picked " + numberOfCards + " cards.";
  }
  else
  {
    PickedCards.Text = "Please enter a valid number.";
  }
}
```

Now that you have a using directive at the top of your MainPage.xaml.cs file, you can use the CardPicker class.

The foreach loop works just like the one in the console app, except instead of writing a line of text to the console it adds a line to the multi-line PickedCards Label control.

What happens if you don't add this last line to your PickedCards Label? Does it look weird? Can you sleuth out how to fix it?

Bullet Points

- Classes have methods that contain statements that perform actions. Well-designed classes have sensible method names.
- Some methods have a **return type**. You set a method's return type in its declaration. A method with a declaration that starts with the `int` keyword returns an int value. Here's a statement that returns an int value: `return 37;`
- When a method has a return type, it **must** have a `return` statement that returns a value that matches a return type. So if a method declaration has the string return type then you need a `return` statement that returns a string.
- As soon as a `return` statement in a method executes, your program jumps back to the statement that called the method.
- Not all methods have a return type. A method with a declaration that starts `public void` doesn't return anything at all. You can still use a `return` statement to exit a void method, as in this example: `if (finishedEarly) { return; }`
- Developers often **reuse** the same code in multiple programs. Classes can help you make your code more reusable.
- When you **select a control** in the XAML code editor, you can edit its properties in the Properties window.
- The XAML code combines with the C# code in the code-behind file to **create a new class**.