O'REILLY®

Fourth Edition

Head First C#

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

Andrew Stellman
& Jennifer Greene

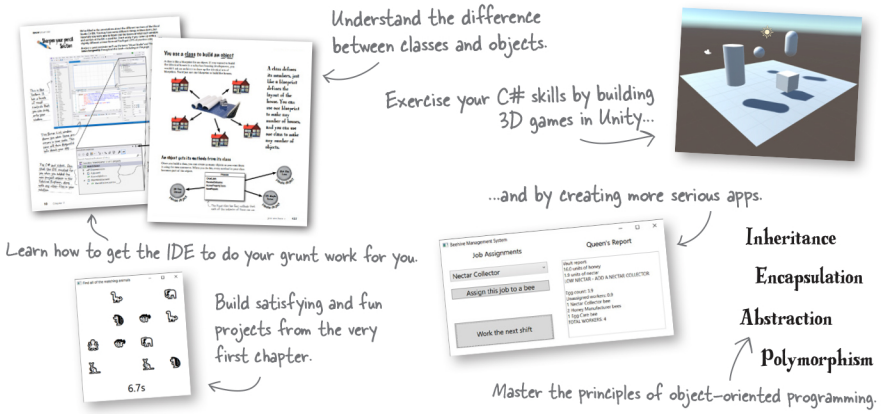
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

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



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

—Ryan White
Game Developer

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

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

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

—Andy Parker
Fledgling C# programmer

What's so special about this book?

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

.NET

US $64.99 **CAN $85.99**

ISBN: 978-1-491-97670-8

9 781491 976708 56499

O'REILLY®

Head First C#

Fourth Edition

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

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

Printed in the United States of America.

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

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

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

Printing History:

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

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

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

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

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

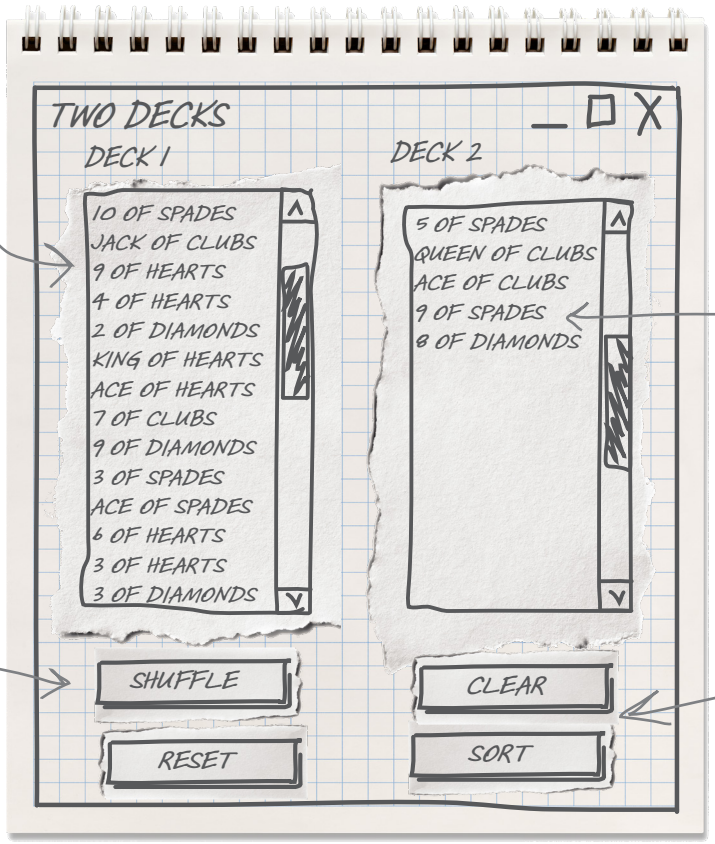
No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI] [2020-11-13]

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.



When you start the app, the left box contains a complete deck of cards. The right box is empty.

Double-clicking on a card in one deck transfers it to the other. So clicking on 9 of Spades will remove it from Deck 2 and add it to Deck 1.

The Shuffle button shuffles the cards in Deck 1, and the Reset button resets it to a sorted 52-card deck.

The Clear button removes all cards from Deck 2, and the Sort button sorts the cards in it so they're in order.

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 ASP.NET Blazor Web Application version of the project. You can find the WPF 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.*

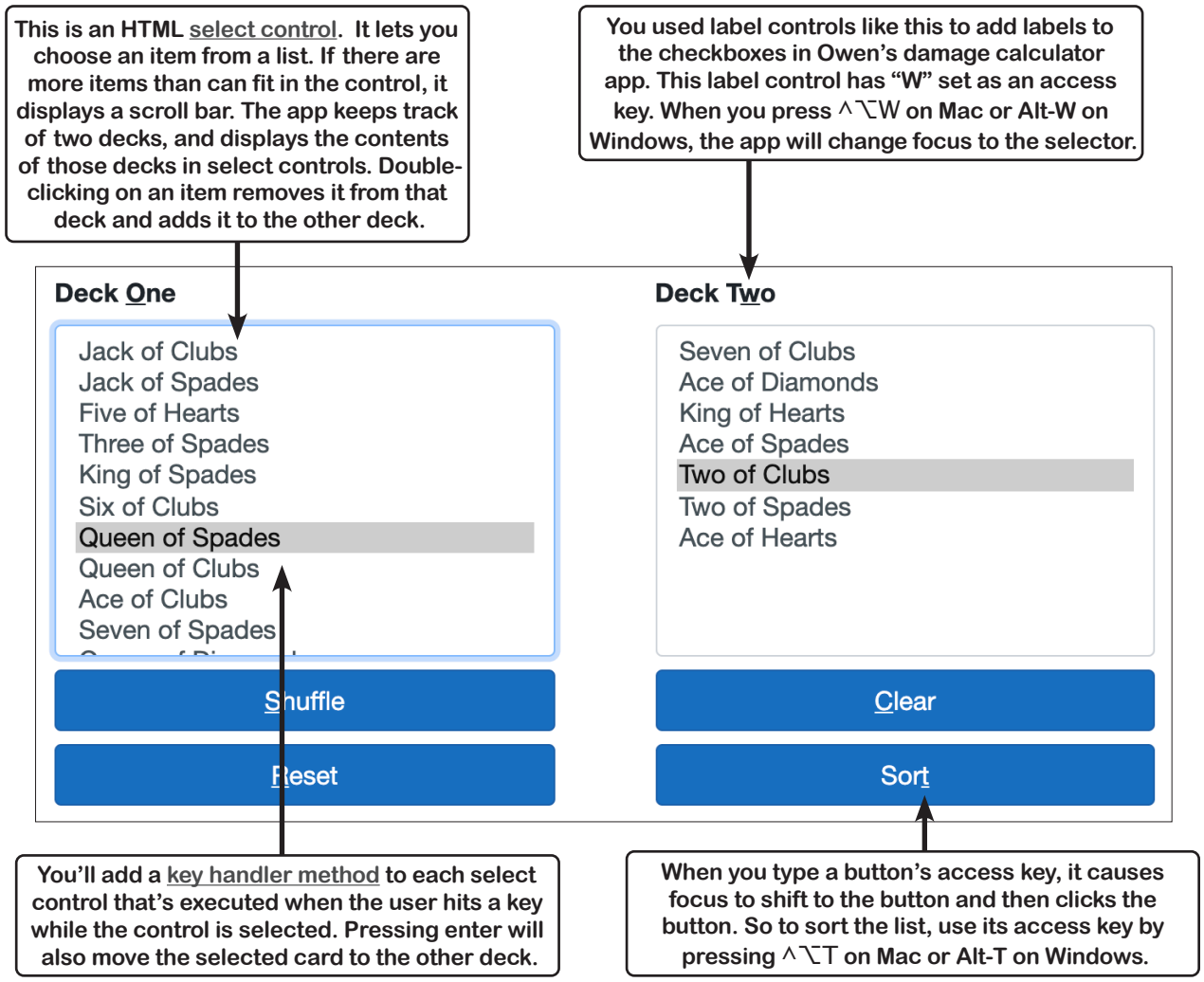
You can see her full advice here, including specific keyboard commands that she recommends you learn to get a great start: https://bit.ly/Keyboard_Tips

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 the next exercise you'll build an app that lets you move cards between two decks. It will have two **select 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 page with two columns (and an empty column in the middle for spacing), each of which contains a label, a select control, and two buttons.

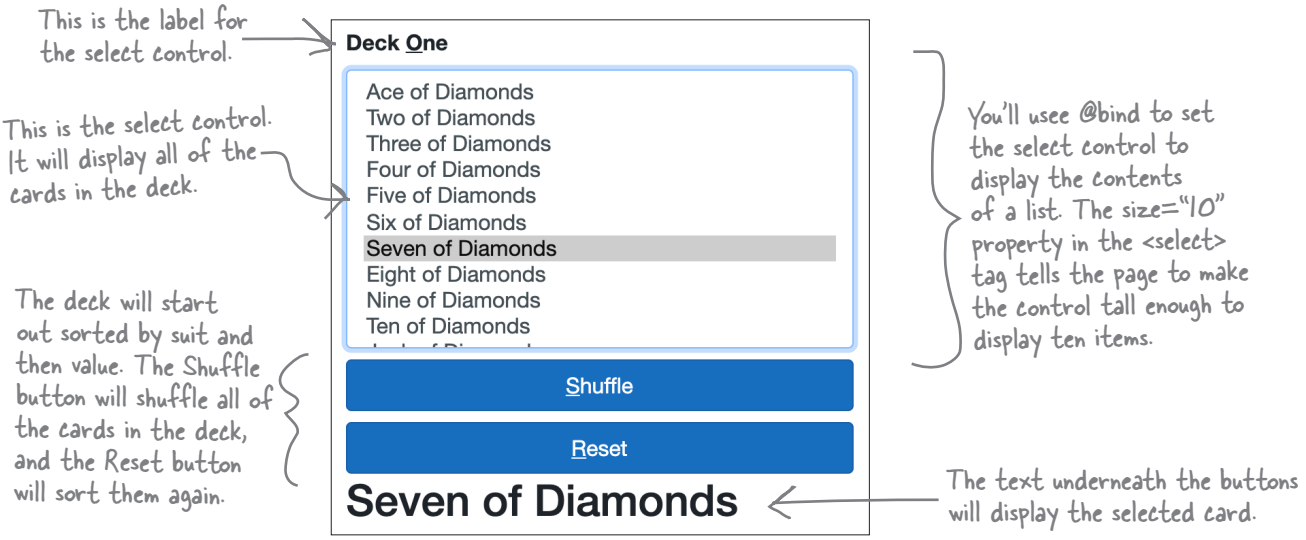
To make your app usable via the keyboard, you'll give your controls **access keys**. An access key is a keyboard shortcut that lets you navigate to a control. For buttons, the access key also clicks the button. By using access keys, you'll give your users a fast and convenient way to navigate your app.



You'll build this app in several parts

When professional developers start projects, they don't just dive in and build everything all at once (at least, not most of the time!). They take an **incremental** approach: they'll tackle the project piece by piece, starting with a small part of the project, getting that part done, and then adding more onto it.

Here's the first part of the project that you'll build:



The HTML markup for the two buttons will be really similar to the buttons in the previous projects. Here's what the HTML markup for the label and select control will look like:

The for="deck1" property says that this is the label for the control with the corresponding id="deck1" property.

The accesskey="o" property tells the page that the access key for this control is O.

The tag causes the text to display in boldface.

```
<label for="deck1" accesskey="o">
    <strong>Deck <span style="text-decoration: underline">O</span>ne</strong>
</label>
<select @bind="twoDecks.LeftCardSelected"
        class="custom-select" size="10" id="deck1">
    @for (int i = 0; i < twoDecks.LeftDeckCount; i++)
    {
        <option value="@i">@twoDecks.LeftDeckCardName(i)</option>
    }
</select>
```

This tag causes the "O" in the label to be underlined. That will let our users know that they can use the access key.

You used a @for loop very similar to this when you sleuthed out the bug in Chapter 1.

Long Exercise

This is a big project! We'll break this project into a few "Long Exercise" parts. In this first part, you'll create a class called TwoDecks, which you'll instantiate in the `@code` section in your Razor page.

1. **Create a new Blazor Web application project named *TwoDecksBlazor*.**

- You'll be working with cards, so **right-click on the project name and choose Add >> Existing Files... (⌥⌘A)**. Navigate to the folder with the solution to the exercise where you shuffled and then sorted the cards (use ⌘-click to multi-select the files), and add **the Card and CardComparerByValue classes** and the **Suits and Values enums** that you created earlier in Chapter 8.
- You'll want these new classes and enums to be in the same namespace as the rest of the project, so open any of these files, right-click on the namespace, and **choose Rename (⌘R) to rename the namespace to *TwoDecksBlazor***.

2. Here's the HTML markup and `@code` section. **Replace your *Index.razor* page** with it:

```
@page "/"

<div class="container">
   <div class="row">
      <div class="col-5">
         <div class="row">
            <label for="deck1" accesskey="o">
               <strong>Deck <span style="text-decoration: underline">O</span>ne</strong>
            </label>
            <select @bind="twoDecks.LeftCardSelected"
                  class="custom-select" size="10" id="deck1">
               @for (int i = 0; i < twoDecks.LeftDeckCount; i++)
               {
                  <option value="@i">@twoDecks.LeftDeckCardName(i)</option>
               }
            </select>
         </div>

         <div class="row">
            <button class="btn btn-primary col mt-2" accesskey="s" @onclick="twoDecks.Shuffle">
               <span style="text-decoration: underline">S</span>huffle
            </button>
         </div>

         <div class="row">
            <button class="btn btn-primary col mt-2" accesskey="r" @onclick="twoDecks.Reset">
               <span style="text-decoration: underline">R</span>eset
            </button>
         </div>

         <div class="row">
            <h2>@twoDecks.LeftDeckCardName(twoDecks.LeftCardSelected)</h2>
         </div>

      </div>
   </div>
</div>

@code {
   TwoDecks twoDecks = new TwoDecks();
}
```

The @code section of your Index.razor page only contains a single field with a reference to a new TwoDecks instance. Keep your eye out for how you'll use separation of concerns to put only code related to the actual behavior of the page in the @code section, and code related to working with the Deck objects in the TwoDeks class.

3. **Add this TwoDecks class** to your project:

```
class TwoDecks
{
    private Deck leftDeck = new Deck();

    public int LeftDeckCount { get { return leftDeck.Count; } }

    public int LeftCardSelected { get; set; }

    public string LeftDeckCardName(int i)
    {
        return leftDeck[i].ToString();
    }

    public void Shuffle()
    {
        leftDeck.Shuffle();
    }

    public void Reset()
    {
        leftDeck = new Deck();
    }
}
```

4. **Add a class called Deck** to your project. The Deck class will hold the deck of cards, just like a List. In fact, it will *work exactly like a list* because you'll use inheritance to extend the List class. Here's the Deck class declaration:

```
using System.Collections.Generic;

class Deck : List<Card>
{
```

You'll need this using statement because your Deck class will inherit from the List class you learned about earlier in Chapter 8.

Add these members to your new Deck class:

- A private static instance of Random, which you'll use to shuffle the cards.
- A method called Reset that clears the list (using the Clear method that it inherited from List), then adds cards in order by suit then value. *Hint: 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.*
- A constructor that calls the Reset method.
- A method called Shuffle that shuffles the deck. Here's how it works. First it creates a copy of the deck like this:

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

  It clears the deck, then uses a while loop to add random cards from the copy to the deck, deleting them from the copy after they're added. The while loop ends when the copy is out of cards.
- A method called Deal takes an int parameter called index, uses base[index] to get the Card reference at the index, calls RemoveAt to remove the Card, then returns that Card.

If you get an error about inconsistent accessibility, make sure that none of your classes or enums are declared with the `public` access modifier.

Long Exercise Solution

This is the first part of a big project. We gave you the TwoDecks class and the markup for the *Index.razor* page. Your job was to create a Deck class that extends List<Card> and adds methods to reset and shuffle the deck.

```
using System.Collections.Generic;

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

    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.Next(copy.Count);
            Card card = copy[index];
            copy.RemoveAt(index);
            Add(card);
        }
    }

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

The List class is in this namespace, so you'll need this using statement.

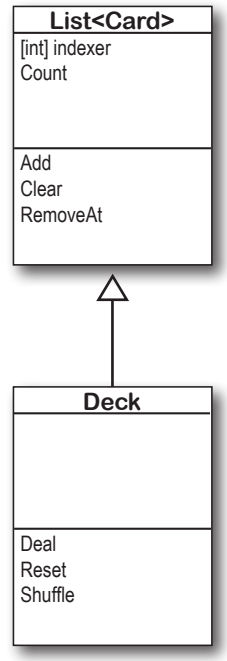
The deck is reset to a sorted deck when it's first instantiated.

We saw code similar to this to create a sorted list of cards at the beginning of Chapter 8. Can you find it?

Since the Deck class extends List<Card>, it inherits RemoveAt, Add, and Clear from that base class..

The Shuffle method creates a new List of cards with a copy of the cards in the deck, then adds random cards from that list to the deck, removing each card as it's added. When the copy is empty, the deck contains all of the original cards, but now they're in a random order.

The Deal method deals a card from a specific index in the deck by getting its reference, removing it from the deck, and then returning the reference.

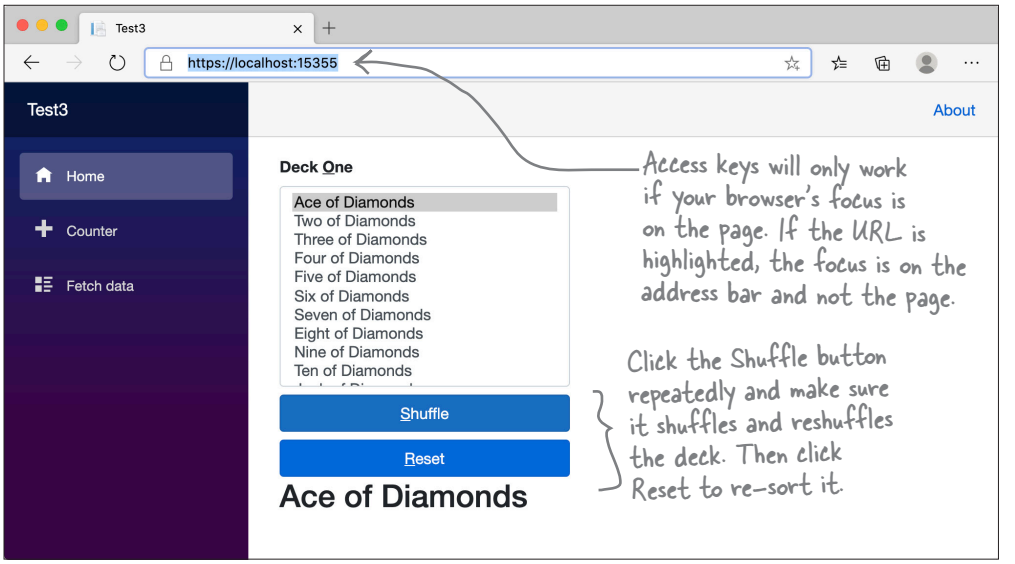


Did you have to remove the `public` access modifier from a class or enum to get rid of an inconsistent accessibility error? You'll learn more about that in Chapter 9.

Test your app so far

Go ahead and run your app and make sure that it works. Use the access keys:

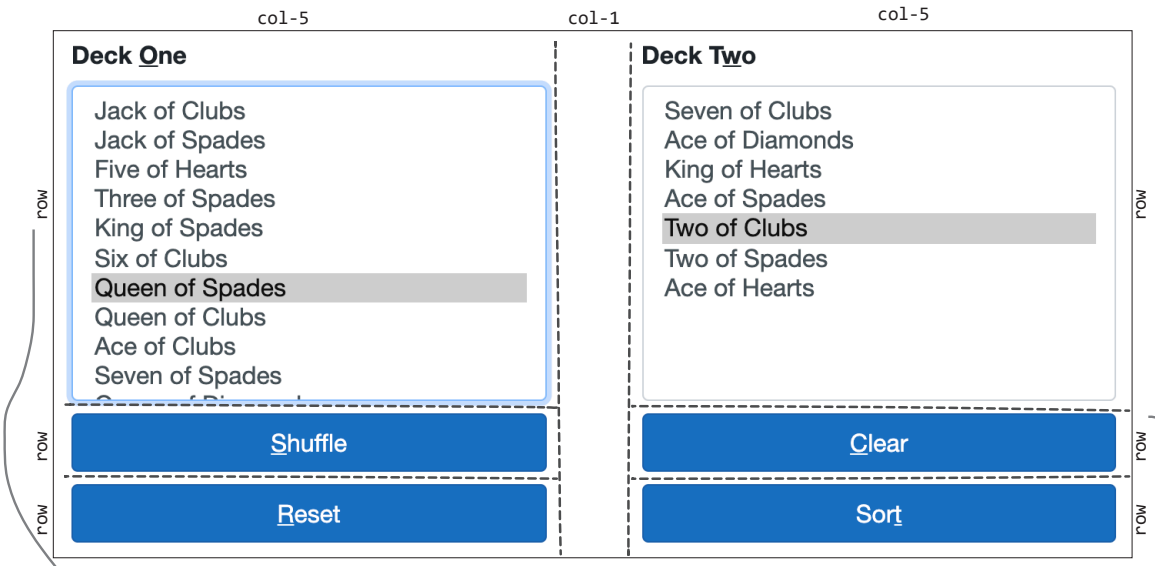
- If your browser is focused on the address bar and not the page, press Tab or click inside the page to focus on it—otherwise the access keys won't work.
- Press ^⌥O to navigate to the Deck One select control. Once focus has switched to it, use the up and down arrows to change the selected card.
- Press ^⌥S to click the Shuffle button and focus on it. Once it's focused, press Enter to click it again.
- Press ^⌥R to click the Reset button and focus on it.



When you take an incremental approach to building a project, it's really useful to thoroughly test your code after each increment. Not only will that help you find bugs, but it's also a really effective way to learn from your progress and figure out if you need to change direction.

Next, you'll add a second deck with two more buttons

In the next part of the project, you'll add a second deck. You'll update the HTML markup to add a second column with a select control and two buttons, and add another deck to the TwoDecks class.




```
<div class="container">
    <div class="row">

        <div class="col-5">
            <div class="row"> ... </div>

            <div class="row">
                <button class="btn btn-primary col mt-2"> ... </button>
            </div>

            <div class="row">
                <button class="btn btn-primary col mt-2"> ... </button>
            </div>
        </div>

        <div class="col-1"/>

        <div class="col-5">

            <div class="row"> ... </div>

            <div class="row">
                <button class="btn btn-primary col mt-2"> ... </button>
            </div>

            <div class="row">
                <button class="btn btn-primary col mt-2"> ... </button>
            </div>
        </div>

    </div>
</div>
```

Compare the rows and columns in the screenshot above with the HTML markup to the right. We replaced some of the markup with ellipses (...) to make it easier to see the structure. Before you move on to the next part of the exercise, try matching up the parts of the HTML to the rows and columns in the screenshot.

Before you move onto the exercise on the next page, think about how you would modify the TwoDecks class to add a second deck and methods to clear and sort it.

Long Exercise

In the next part of the project, you'll modify the HTML page to add a selector for a second deck, and you'll modify the TwoDecks class to add methods to sort and clear that second deck.

1. **Modify the TwoDecks class to a second Deck instance.** Add the following members:

- A private Deck field called rightDeck initialized with a new Deck instance.
- A read-only property called RightDeckCount that returns the number of cards in rightDeck.
- An automatic int property called RightCardSelected.
- A method called RightDeckCardName that takes the index of a card as the parameter and returns its name.
- A method called Sort that uses CardComparerByValue to sort the right deck.
- A method called Clear, but ***don't*** **make it clear the right deck**. Make the Clear method **shuffle** the right deck. You'll use this to test the page and make sure it works. Don't worry, you'll modify it in the last part of the project.

2. **Modify the HTML markup to add a second deck.** You'll add a second column with a selector and two buttons:

- Add an empty column: `<div class="col-1" />`
  It should go right above the second-to-last `</div>` at the end of the HTML markup, so the last three lines of the HTML markup above the @code section look like this:

```
        <div class="col-1" />
    </div>
</div>
```

- Copy the markup for the entire left column, starting with `<div class="col-5" />` and ending with its closing `</div>` just above the empty column that you added. Paste the copied HTML after the empty column you added.
- Change every occurrence of LeftDeckCount, LeftCardSelected, and LeftDeckCardName in the right column's HTML markup to RightDeckCount, RightCardSelected and RightDeckCardName.
- Modify the select control's access key, and change its `id` property to change it from `deck1` to `deck2`:

```
<select @bind="twoDecks.RightCardSelected"
        class="custom-select" size="10" id="deck2">
```

- Modify the label control's access key, and change its text to "Deck Two" and `for` property from `deck1` to `deck2`:

```
<label for="deck2" accesskey="w">
    <strong>Deck T<span style="text-decoration: underline">w</span>o</strong>
</label>
```

- Change the <label> control's for property and the <select> control's id property from deck1 to deck2.
- Modify the two buttons so they call the Clear and Sort methods:

```
<div class="row">
    <button class="btn btn-primary col mt-2" accesskey="c" @onclick="twoDecks.Clear">
        <span style="text-decoration: underline">C</span>lear
    </button>
</div>

<div class="row">
    <button class="btn btn-primary col mt-2" accesskey="t" @onclick="twoDecks.Sort">
        Sor<span style="text-decoration: underline">t</span>
    </button>
</div>
```

Long Exercise Solution

Here's the updated TwoDecks class. You added a private Deck field called rightDeck with a reference to a second Deck object, properties to get its count and track the selected card, and methods to get a card name, clear, and sort it.

```
class TwoDecks
{
    private Deck leftDeck = new Deck();
    private Deck rightDeck = new Deck();

    public int LeftDeckCount { get { return leftDeck.Count; } }
    public int RightDeckCount { get { return rightDeck.Count; } }

    public int LeftCardSelected { get; set; }
    public int RightCardSelected { get; set; }

    public string LeftDeckCardName(int i)
    {
        return leftDeck[i].ToString();
    }

    public string RightDeckCardName(int i)
    {
        return rightDeck[i].ToString();
    }

    public void Shuffle()
    {
        leftDeck.Shuffle();
    }

    public void Reset()
    {
        leftDeck = new Deck();
    }

    public void Clear()
    {
        rightDeck.Shuffle();
    }

    public void Sort()
    {
        rightDeck.Sort(new CardComparerByValue());
    }
}
```

We asked you to make the Clear button shuffle the deck instead of sorting it, because that makes it easier for you to test your code and make sure it works.

Long Exercise Solution

Here's the HTML markup for the second column, as well as an empty middle column to add space between them. We left out the markup for the first column, since it didn't change.

```
<div class="container">
    <div class="row">
        <div class="col-5">

            ...

        </div>

        <div class="col-1" />

        <div class="col-5">
            <div class="row">
                <label for="deck2" accesskey="w">
                    <strong>Deck T<span style="text-decoration: underline">w</span>o</strong>
                </label>
                <select @bind="twoDecks.RightCardSelected"
                        class="custom-select" size="10" id="deck2">
                    @for (int i = 0; i < twoDecks.RightDeckCount; i++)
                    {
                        <option value="@i">@twoDecks.RightDeckCardName(i)</option>
                    }
                </select>
            </div>

            <div class="row">
                <button class="btn btn-primary col mt-2" accesskey="c" @onclick="twoDecks.Clear">
                    <span style="text-decoration: underline">C</span>lear
                </button>
            </div>

            <div class="row">
                <button class="btn btn-primary col mt-2" accesskey="t" @onclick="twoDecks.Sort">
                    Sor<span style="text-decoration: underline">t</span>
                </button>
            </div>

            <div class="row">
                <h2>@twoDecks.RightDeckCardName(twoDecks.RightCardSelected)</h2>
            </div>

        </div>
    </div>
</div>
```

The HTML markup for the left column stays exactly the same as before, so we didn't include it in this solution..

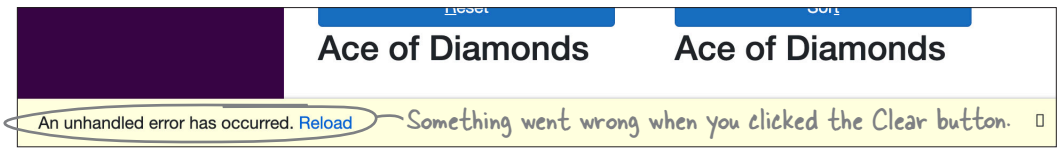
Make your Clear button work

We asked you to make the Clear button shuffle the deck instead of clear it, because we wanted you to be able to test your app—if you didn't have a way to shuffle the deck, you couldn't check that your sort button would work. **Modify your TwoDecks.Clear method** to clear the right deck:

```
public void Clear()
{
    rightDeck.Clear();
}
```

Do this!

Now run your app and **click the Clear button**. Uh-oh—something's wrong:



Time to put on our Sherlock Holmes caps.

Sleuth it out

Let's use the Visual Studio debugger to sleuth out this bug.

1. Choose **Other Windows >> Application Output - TwoDecksBlazor – Debug** from the View menu to open the application output window. You'll see an exception with a stack trace:

```
System.ArgumentOutOfRangeException: Index was out of range. Must be non-negative and
less than the size of the collection.
Parameter name: index
  at System.Collections.Generic.List`1[T].get_Item (System.Int32 index)
  at TwoDecksBlazor.TwoDecks.RightDeckCardName (System.Int32 i) in TwoDecks.cs:23
```

Aha! Some clues to what went wrong.

   So we know the line of code that caused the problem.

2. Let's have a look at that particular line of code:

```
public string RightDeckCardName(int i)
{
    return rightDeck[i].ToString();
}
```

Here's the line that's throwing the 'Index was out of range' exception.

   So what's going on?

3. The problem only happens when you click the Clear button:

   - Run your app
   - After it loads, place a breakpoint on the line that's throwing the exception
   - Use the Locals window to find the values of `i` and `rightDeck`. It turns out `i` is `0`, and `rightDeck` is an empty list. And when you try to get element zero—the first element—from an empty list or array, you'll get an ArgumentOutOfRangeException.

   We know the details of the crime. Now we just need to narrow down the list of suspects.

Sleuth it out

It's time to narrow down the list of suspects, find the culprit, and solve the case.

4. There are only two places in your code that call the TwoDecks.RightDeckCardName method. The first is the for loop in your *Index.razor* page that adds the options to the select control:

```
@for (int i = 0; i < twoDecks.RightDeckCount; i++)
{
    <option value="@i">@twoDecks.RightDeckCardName(i)</option>
}
```

Sometimes the easiest way to check if you've found the problem is to remove it and see if it still happens. So try **changing the for loop so it just prints the card index**:

```
@for (int i = 0; i < twoDecks.RightDeckCount; i++)
{
    <option value="@i">Index @i</option>
}
```

Run your app again. The selector fills with lines like "Index 5" – so now try pressing the Clear button. If you didn't remove your breakpoint, it will still trigger, so remove it and continue.

No, the exception still happens. So **change the loop back** to the way it was so it adds the card names to the select control again.

5. Here's the other place in your *Index.razor* page that calls the TwoDecks.RightDeckCardName method:

```
<div class="row">
    <h2>@twoDecks.RightDeckCardName(twoDecks.RightCardSelected)</h2>
</div>
```

Try **removing that entire `<div>...</div>` block** from the page. Run your code and press the Clear button. It doesn't throw the exception anymore—your button clears the deck. ***Now your app works!***

6. Go ahead and **remove the other `<div>...</div>` block** to make the left column match:

```
<div class="row">
    <h2>@twoDecks.LeftDeckCardName(twoDecks.LeftCardSelected)</h2>
</div>
```

Remove both of these blocks at the bottom of each column.

Now your app matches the screenshot at the beginning of the project.

MINI Sharpen your pencil

Why did removing that ... block from your *Index.razor* page fix the exception?

.....

.....

.....

Move cards between your decks

In last part of the project, you'll modify your app to move cards from one deck to the other. When the user double-clicks on a card in the selector control, the app will move that card to the other deck. You'll use a **mouse event handler** to handle the double-click. The user can also focus on a selector, use the arrow keys to choose a card, and press enter to move the selected card. You'll use a **key event handler** to respond to the keypress when the control is focused.

Do this!

1. **Add a Direction enum.**
Your Blazor code will use this enum to tell the TwoDecks class which direction to move the card.

```
enum Direction
{
    LeftToRight,
    RightToLeft,
}
```

2. **Add a MoveCard method to your TwoDecks class.**
The MoveCard method either moves the currently selected card in the left deck to the right deck, or it moves the card from right to left.

```
public void MoveCard(Direction direction)
{
    if (direction == Direction.LeftToRight)
    {
        rightDeck.Add(leftDeck[LeftCardSelected]);
        leftDeck.RemoveAt(LeftCardSelected);
    }
    else
    {
        leftDeck.Add(rightDeck[RightCardSelected]);
        rightDeck.RemoveAt(RightCardSelected);
    }
}
```

The Direction enum tells the MoveCard method which direction to move the card.

Moving the selected card means adding the card at the selected index to the destination deck, then removing it from the source deck.

MINI Sharpen your pencil Solution

Why did removing that ... block from your *Index.razor* page fix the exception?

The select control's @bind binding caused it to set twoDecks.RightCardSelected to zero when the deck was cleared. Then the binding in the passed that value as an argument to the twoDecks.RightDeckCardName method, causing it to try to get element #0 from an empty list.

3 Add keyboard and mouse event handlers.

Modify the `@code` section at the bottom of your *Index.razor* page to add four event handler methods: a keyboard and mouse handler for the left deck, and corresponding methods for the right deck.

```
@code {
    TwoDecks twoDecks = new TwoDecks();

    private void LeftKeyHandler(KeyboardEventArgs e)
    {
        if (e.Key == "Enter")
            twoDecks.MoveCard(Direction.LeftToRight);
    }

    private void LeftDblClickHandler(MouseEventArgs e)
    {
        twoDecks.MoveCard(Direction.LeftToRight);
    }

    private void RightKeyHandler(KeyboardEventArgs e)
    {
        if (e.Key == "Enter")
            twoDecks.MoveCard(Direction.RightToLeft);
    }

    private void RightDblClickHandler(MouseEventArgs e)
    {
        twoDecks.MoveCard(Direction.RightToLeft);
    }
}
```

A Blazor keyboard event handler is a method that takes a KeyboardEventArgs parameter. You can use `@onkeypress` to hook a control up to it. When the user focuses on that control and presses a key, Blazor calls the event handler, passing it an argument with e.Key set to the key that was pressed. To detect an enter key, check that e.Key is equal to the string "Enter".

A mouse event handler takes a MouseEventArgs parameter. To handle normal left-clicks, you'd use `@onclick` to bind a control to the handler. But we want to keep clicks for selecting, and instead use double-clicks—and for that we'll need to use `@ondblclick`.

4 Update your HTML markup to call the event handlers.

Modify the HTML markup for the left select control an `@onkeypress` attribute that calls LeftKeyHandler and an `@ondblclick` attribute that calls LeftDblClickHandler:

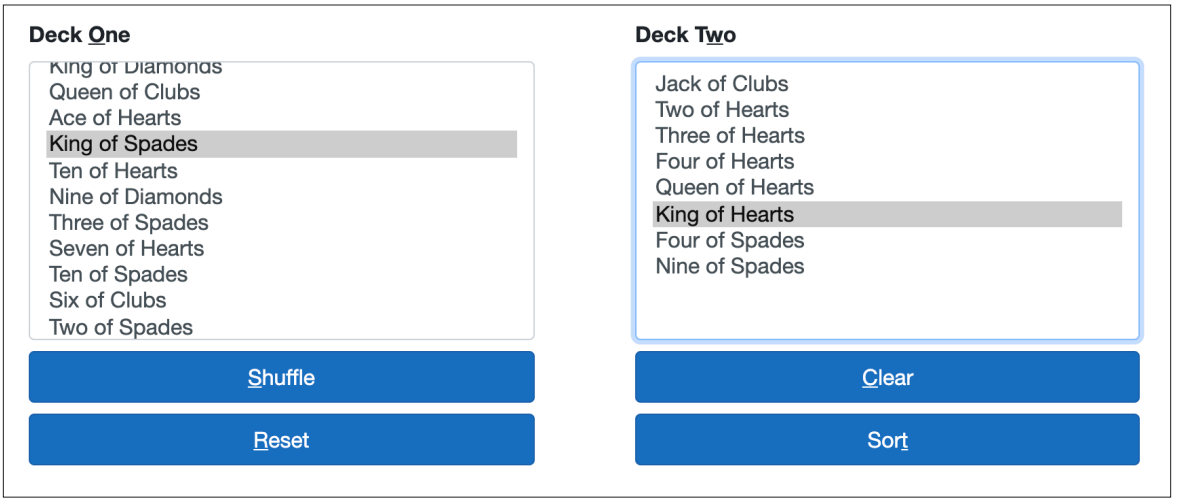
```
<select @bind="twoDecks.LeftCardSelected"
        @onkeypress="LeftKeyHandler" @ondblclick="LeftDblClickHandler"
        class="custom-select" size="10" id="deck1">
```

Now modify the HTML markup for the left select control an `@onkeypress` attribute that calls RightKeyHandler and an `@ondblclick` attribute that calls RightDblClickHandler:

```
<select @bind="twoDecks.RightCardSelected"
        @onkeypress="RightKeyHandler" @ondblclick="RightDblClickHandler"
        class="custom-select" size="10" id="deck2">
```

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. It also gives you extra opportunities to find, sleuth out, and fix bugs.