

O'REILLY®

Fourth  
Edition

# Head First

# C#

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

---

Andrew Stellman  
& Jennifer Greene



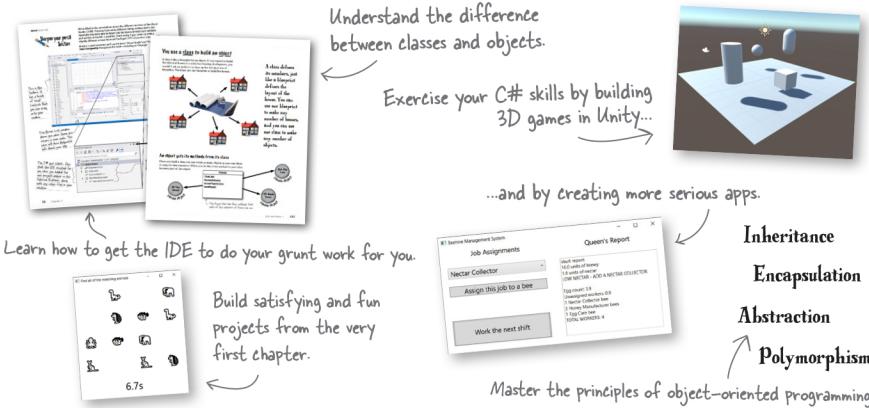
A Brain-Friendly Guide

# 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!



## 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



5 6 4 9 9  
9 781491 976708

"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

O'REILLY®

# Head First C#

Fourth Edition



WOULDN'T IT BE DREAMY IF  
THERE WAS A C# BOOK THAT WAS  
MORE FUN THAN MEMORIZING  
A DICTIONARY? IT'S PROBABLY  
NOTHING BUT A FANTASY...

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](mailto: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

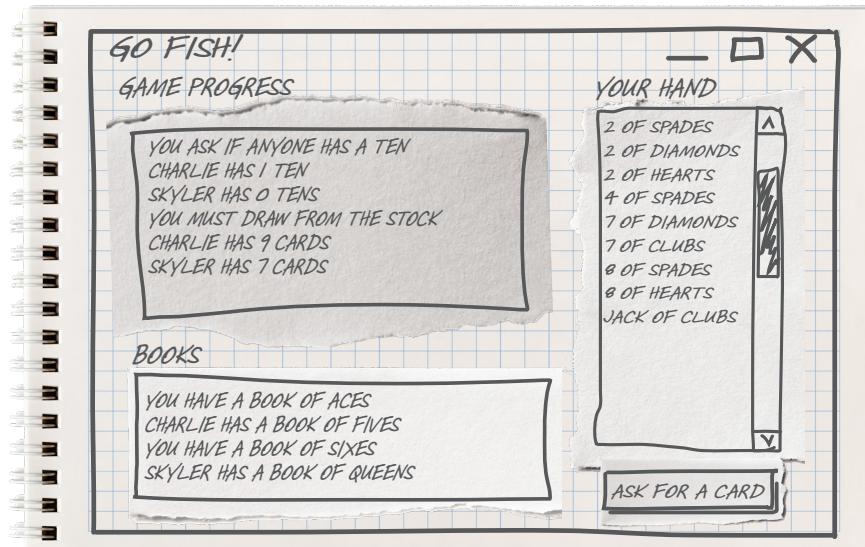
[LSI]

[2020-11-13]

## Downloadable exercise: Go Fish



In this next exercise you'll build a Go Fish card game where you play against computer players. Unit testing will play an important part, because you'll be doing **test-driven development**, a technique where you write your unit tests before you write the code that they test.



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.

**You can download the latest version of this PDF  
from the *Head First C# GitHub* page:**

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

# Build a card game where you play against the computer

In this project, you'll build a card game where a person plays a *Go Fish!* against a number of computer players. Like previous projects, you'll do it in parts

## The rules of Go Fish!

*Go Fish!* is a game played by two to five players. There are a few variations—here are the rules that you'll use for your version:

- ★ The game is played with one **human player** and up to four **computer players**.
- ★ The game starts with a **shuffled deck** of 52 playing cards.
- ★ Each player is dealt a **hand** of 5 cards from the deck. The remaining cards are called the **stock**.
- ★ The players play **rounds**, taking turns playing the round. The human player starts each round, followed by each computer player. They go in the same order during each round. During the round, each player:
  - Chooses a **value** from their hand. The value must match one of the cards in the player's hand.
  - Chooses **another player** and asks if they have any cards of that value.
  - If the other player *has any cards with that value*, those cards are **moved** from the other player's hand to the hand of the player who asked for them.
  - If the other player *does not have any cards with that value*, the player asking for the card must “Go fish!” and **draw a card** from the stock. If the stock is out, the player does not draw a card.
  - The player checks their hand for **books** of cards. A book is a set of all four cards in each suit that have the same value. They remove any complete books from their hand and set them aside. After a book is set aside, that book's value is no longer part of the game.
- ★ The game ends when all players are out of cards. The winner of the game is the player with the most books. The game can end in a tie.

## Playing a sample round

Let's walk through one player's actions during a sample round, just to make sure the rules are clear.

### 1 The player checks their hand to see what values they have.

We'll start with a player that currently has a hand of six cards: Ten of Hearts, Six of Spades, Seven of Diamonds, Eight of Spades, Ten of Diamonds, and Ten of Clubs. It's that player's turn to ask for a card.



**2** The player chooses a value and asks another player for that value.

The current player decides to ask for tens and selects another player to ask. They then ask that other player, “Do you have any 10s?”

DO YOU HAVE ANY  
TENS?



**3** The other player hands over any cards with that value.

In this case, the other player has one ten, the Ten of Spades. They hand that card over to the player who asked for the card.

**4** The card is added to the player's hand.

The player who asked for the ten adds the Ten of Spades to their hand. Their hand now has all four tens.



**5** Pull out books and move to the next player.

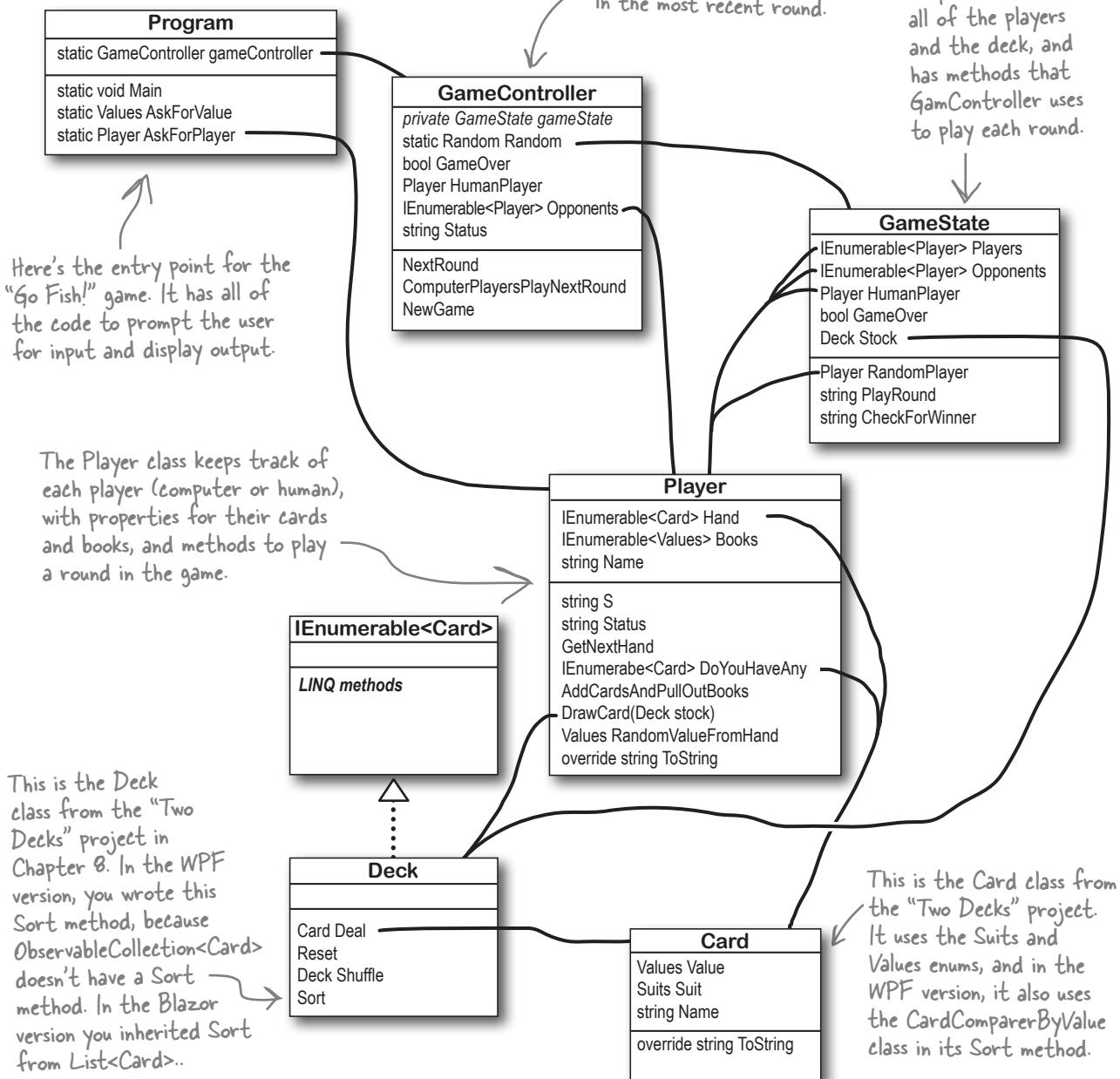
The player pulls out the book of tens and sets them aside. The player is now done playing the round, and gameplay moves to the next player in the game.



How would you start building your own *Go Fish!* game? If you wanted to break the project into smaller parts, what part would you start with?

# Here's the class diagram

Your game will use the same Deck and Card classes you used in your Chapter 8 “Two Decks” project. You’ll build a Program class with a Main method and three additional classes: GameController, GameState, and Player.



# You'll build this project in multiple parts starting with the Player class

You've done several "Long Exercise" projects now, and each time you broke the project down into steps that you could build one at a time. You'll do the same for this project. Here's how it will work:

1. First, you'll build the Player class with members that control a player and keeps track of its hand and books.
2. Then you'll build the GameState class that keeps track of the players and deck in the game, with methods to play a round.
3. After that, you'll build the GameController class that manages the game, calling methods to play round after round until the game is over.
4. Finally, you'll build the Program class, with a Main method that takes input from the user and displays the state of the game.



HOW AM I SUPPOSED TO BUILD THOSE CLASSES IN THAT ORDER? I CAN'T RUN THE CODE IN THE PLAYER CLASS UNTIL I HAVE A GAMESTATE, AND I CAN'T GET TO THAT CODE WITHOUT A GAMECONTROLLER, AND THAT WON'T WORK UNLESS IT'S CALLED FROM THE ENTRY POINT.

**You'll use unit tests to build and test each class, starting from the bottom of the class diagram and working your way up.**

The class diagrams that you've seen in the book so far showed you the class members (fields, properties, and methods) and hierarchy (base classes and interfaces they extended). We added additional information to the class diagram for this project: lines that connect the classes and **show how each class uses the other classes**.

First, we arranged the classes in the diagram so that when a class uses another class is above it—for example, the Player class stores Deck and Card references, so the box for the Player class is higher on the page than boxes for Deck Card. Then we drew each line from the Player class member that uses a Deck or Card reference to the top of the Deck or Card box. The Player.DoYouHaveAny method, for example, returns an `IEnumerable<Card>` so there's a line from the DoYouHaveAny method in the Player box to the top of the Card box.

It's possible to do the project starting at the top of the diagram, but that would be complicated: to build the Program methods you'd need to create temporary "fake" methods in GameController, then you'd have to do the same for GameState, and then Player—all before you even wrote a single line of code for the Main method.

Luckily, there's a better way. You already have the Card and Deck classes, so you can start with the Player class—and you can use **unit tests** to make sure the Player class works before you move on to the members of the GameState class that use it.

# Create the solution and add a unit test project

Do this!

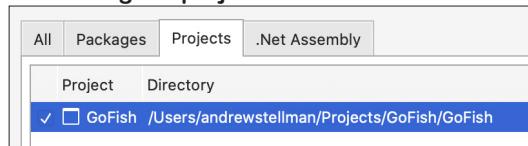
The first step in the project is to **create a new .NET Core Console App project called GoFish** and add an **MSTest unit test project called GoFishTests** to the solution, just like you did earlier in Chapter 9. Here's a refresher with all of the steps you need to follow to create the project.

- ★ Create a **new Console App (.NET Core) project** called *GoFish*.
- ★ Right-click on the solution and **add a new MSTest Unit Test project** called *GoFishTests*, just like you did earlier in Chapter 9. On Windows enter “MSTest” in the search box to search for the unit test template, on Mac choose MSTest Project from the *Tests* section under *Web and Console*. Make sure you choose the **.NET Core MSTest project** (otherwise you'll get a build error).

## Adding the project reference in Windows

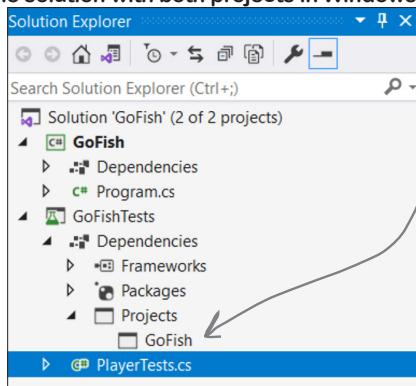


## Adding the project reference in macOS



- ★ Modify the *GoFishTests* project references to add a reference to the *GoFish* project. Expand the *GoFishTests* project. On Windows **right-click on References**, on Mac **right-click on Dependencies**. In both cases, **choose Add Reference...**, select Projects, and **check GoFish**.
- ★ Visual Studio created the unit test project with a file called *UnitTest1.cs* that contains a test class class called *UnitTest1*. Rename the class to *PlayerTests* and the file to *PlayerTests.cs*.

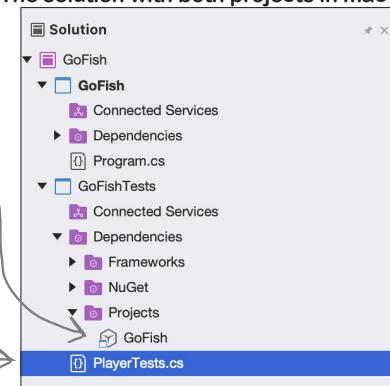
## The solution with both projects in Windows



You modified the references in the unit test project *GoFishTests* to add a reference to the *GoFish* console app project.

You renamed the class with the unit tests to *PlayerTests*.

## The solution with both projects in macOS



- ★ Run the unit test in your *PlayerTests* test class. On Windows, choose **Run all Tests (Ctrl+R, A)** from the Tests menu, and the results will be displayed in the Test Explorer window. On Mac, choose **Run Unit Tests** from the Run menu, and the results will be displayed in the Tests window.
- ★ Add the **Card, Deck, and CardComparerByValue classes** and the **Suits and Values enums** from the Two Decks project in Chapter 8 to the *Go Fish* project. **Rename the namespace** to *GoFish*.

Make sure you add the version of the *Deck* class that you modified in Chapter 9 to support method chaining by changing the return type to *Deck* and returning “this”.

We learned all about LINQ in Chapter 9. Here's a quick recap of some useful LINQ syntax and methods that you'll use in this project.

# HERE'S SOME LINQ THAT YOU'LL FIND USEFUL!

Add the `using System.Linq;` directive to add LINQ methods to any **sequence**, or object that implements the `IEnumerable<T>` interface. It also adds the `Enumerable` class, with useful static methods:

- `Enumerable.Empty` returns an empty sequence
- `Enumerable.Range(5, 8)` returns a sequence of 8 ints starting at 5: { 5, 6, 7, 8, 9, 10, 11, 12 }
- `Enumerable.Repeat("Hi", 3)` returns a sequence with the string "Hi" repeated four times: { "Hi", "Hi", "Hi" }

LINQ methods to take the **count**, **minimum**, **maximum**, **sum**, or **average** of a sequence of numbers:

```
var values = Enumerable.Range(5, 8);
Console.WriteLine(values.Count()); // 8
Console.WriteLine(values.Min()); // 3
Console.WriteLine(values.Max()); // 12
Console.WriteLine(values.Sum()); // 68
Console.WriteLine(values.Average()); // 8.5
```

LINQ methods to take the **first** or **last** elements in a sequence and **concatenate** sequences together:

```
var first3 = values.Take(3);
var last2 = values.TakeLast(2);
var joined = first3.Concat(last2);
Console.WriteLine(string.Join(", ", joined));
// writes 5, 6, 7, 11, 12
```

There are LINQ methods to **skip** values in a sequence or take the **first** value in a sequence:

```
var sk = values.Skip(3).Take(4);
var f = sk.First() // 8
Console.WriteLine(string.Join(", ", sk));
// writes 8, 9, 10, 11
```

Use **lambda expressions** with any LINQ method that takes a `Func` parameter. The **Where** method can filter a sequence so it contains only specific values:

```
var d = new Deck();
var a = d.Where(c => c.Value == Values.Ace);
// a contains the four Ace cards in the deck
```

The **Select** method modifies all elements in a sequence:

```
var evens = Enumerable.Range(1, 5)
    .Select(n => n * 2);
// evens contains { 2, 4, 6, 8, 10 }
```

Select works really well with **string interpolation**:

```
var message = evens.Select(n =>
    $"#{n + 1}");
Console.WriteLine(
    string.Join(Environment.NewLine, message))
// Writes 5 lines: #3, #5, #7, #9, #11
```

Use **LINQ query syntax** to manipulate a sequence:

```
var result =
    from v in values // range variable v
    where v < 9 // choose only values > 9
    orderby v descending // sort
    select v * 10; // multiply each by 10
// result = 80, 70, 60, 50
```

Or **chained LINQ methods** to make the same query:

```
var result = values
    .Where(v => v < 9)
    .OrderByDescending(v => v)
    .Select(v => v * 10);
// result = 80, 70, 60, 50
```

Use LINQ query syntax for creating **groups**:

```
var groups = from card in new Deck()
    group card by card.Suit
    into suitGroup
    orderby suitGroup.Key descending
    select suitGroup;
// groups contains four groups, one per suit
```

The same query using LINQ methods and lambdas:

```
var groups = new Deck()
    .GroupBy(card => card.Suit)
    .OrderByDescending(
        suitGroup => suitGroup.Key);
```

Each group is an object with a `Key` property.



## Long Exercise

Part 1: Create the Player class. Here are all of the members of the Player class. This is a **skeleton**, or an outline of a class that has placeholders for (most of) its members but doesn't include the code. We also included XML documentation for all of the public members to help you understand what they need to do.

This Player class skeleton is your starting point. Some methods throw `NotImplementedException` exceptions. Your job is to replace them with working code that makes the Player class do what it's supposed to do.

```
using System.Collections.Generic;
using System.Linq;

public class Player
{
    public static Random Random = new Random();

    private List<Card> hand = new List<Card>();
    private List<Values> books = new List<Values>();

    /// <summary>
    /// The cards in the player's hand
    /// </summary>
    public IEnumerable<Card> Hand => hand;

    /// <summary>
    /// The books that the player has pulled out
    /// </summary>
    public IEnumerable<Values> Books => books;

    public readonly string Name;

    /// <summary>
    /// Pluralize a word, adding "s" if a value isn't equal to 1
    /// </summary>
    public static string S(int s) => s == 1 ? "" : "s";

    /// <summary>
    /// Returns the current status of the player: the number of cards and books
    /// </summary>
    public string Status => throw new NotImplementedException();

    /// <summary>
    /// Constructor to create a player
    /// </summary>
    /// <param name="name">Player's name</param>
    public Player(string name) => Name = name;

    /// <summary>
    /// Alternate constructor (used for unit testing)
    /// </summary>
    /// <param name="name">Player's name</param>
    /// <param name="cards">Initial set of cards</param>
    public Player(string name, IEnumerable<Card> cards)
    {
        Name = name;
        hand.AddRange(cards);
    }
}
```

We saw earlier in Chapter 9 that you need to make your classes public to use them in the the unit test project. Make sure you **modify** the Card, Deck, and CardComparerByValue classes and Suits and Values enums to **add the public access modifier**, otherwise you'll get compiler errors about inconsistent accessibility.

We implemented a few of the members—like the Hand and Books properties and their backing fields, the readonly Name field, and a useful S method to pluralize an English word, so  `$"card{S(hand.Count())}"` interpolates to “card” if there's one card in the hand, and “cards” if there are either zero or multiple cards.

We added two constructors. The second one is mainly used for unit testing.

We learned in Chapter 4 that you should only have a **single instance** of Random, so modify the Deck class to use the same static Random instance as Player. **Change** this line:

private static Random random = new Random();

To this: private static Random random = Player.Random;



```

    /// <summary>
    /// Gets up to five cards from the stock
    /// </summary>
    /// <param name="stock">Stock to get the next hand from</param>
    public void GetNextHand(Deck stock)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// If I have any cards that match the value, return them. If I run out of cards, get
    /// the next hand from the deck.
    /// </summary>
    /// <param name="value">Value I'm asked for</param>
    /// <param name="deck">Deck to draw my next hand from</param>
    /// <returns>The cards that were pulled out of the other player's hand</returns>
    public IEnumerable<Card> DoYouHaveAny(Values value, Deck deck)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// When the player receives cards from another player, adds them to the hand
    /// and pulls out any matching books
    /// </summary>
    /// <param name="cards">Cards from the other player to add</param>
    public void AddCardsAndPullOutBooks(IEnumerable<Card> cards)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Draws a card from the stock and add it to the player's hand
    /// </summary>
    /// <param name="stock">Stock to draw a card from</param>
    public void DrawCard(Deck stock)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Gets a random value from the player's hand
    /// </summary>
    /// <returns>The value of a randomly selected card in the player's hand</returns>
    public Values RandomValueFromHand() => throw new NotImplementedException();

    public override string ToString() => Name;
}

```

Use LINQ to implement RandomValueFromHand:  
first order the list by card value, then select the  
value of each card, skip a random number of  
cards, and choose the first element in the result.

We gave you XML comments as a starting point to help you figure out what the Player class needs to do. But you'll need more information than that to figure out what the Player class is supposed to do! What do you think we'll give you to help with that?



## Long Exercise

**Part 1 (continued): Add the Player class unit tests.** Here's the complete `PlayerTests` class, along with a `MockRandom` class used by one of the tests. Add this code to `PlayerTests.cs`, and add the `MockRandom` object to the `GoFishTests` project. Your job is to modify the `Player` class so all of these tests pass.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;
using System.Linq;
using GoFish;
```

```
[TestClass]
public class PlayerTests
```

```
{ [TestMethod]
public void TestGetNextHand()
```

```
{ var player = new Player("Owen", new List<Card>());
player.GetNextHand(new Deck());
CollectionAssert.AreEqual(
    new Deck().Take(5).Select(card => card.ToString()).ToList(),
    player.Hand.Select(card => card.ToString()).ToList()); }
```

We saw `CollectionAssert` in Chapter 9 – it compares an expected collection with an actual result.

```
[TestMethod]
public void TestDoYouHaveAny()
```

```
{ IEnumerable<Card> cards = new List<Card>()
{
    new Card(Values.Jack, Suits.Spades),
    new Card(Values.Three, Suits.Clubs),
    new Card(Values.Three, Suits.Hearts),
    new Card(Values.Four, Suits.Diamonds),
    new Card(Values.Three, Suits.Diamonds),
    new Card(Values.Jack, Suits.Clubs),
};
```

```
var player = new Player("Owen", cards);
```

```
var threes = player.DoYouHaveAny(Values.Three, new Deck())
    .Select(Card => Card.ToString())
    .ToList();
```

```
CollectionAssert.AreEqual(new List<string>()
{
    "Three of Diamonds",
    "Three of Clubs",
    "Three of Hearts",
}, threes);
```

```
Assert.AreEqual(3, player.Hand.Count());
```

```
var jacks = player.DoYouHaveAny(Values.Jack, new Deck())
    .Select(Card => Card.ToString())
    .ToList();
```

```
CollectionAssert.AreEqual(new List<string>()
{
    "Jack of Clubs",
    "Jack of Spades",
}, jacks);
```

```
var hand = player.Hand.Select(Card => Card.ToString()).ToList();
CollectionAssert.AreEqual(new List<string>() { "Four of Diamonds" }, hand);
```

```
Assert.AreEqual("Owen has 1 card and 0 books", player.Status); }
```

Unit tests aren't just useful for making sure your code works. They're also a great way to understand what your code is supposed to do. Part of your job is to read through these unit tests to figure out what the `Player` class should do. You'll know your class is working when all of the unit tests pass.

`GetNewHand` returns up to 5 cards from the deck. `CollectionAssert` can't compare cards, so we used the `Select LINQ` method to convert them to lists of card names to compare.

The test sets up an instance of `Player` with a set of cards. We used the constructor that takes a name and a sequence of cards to start with a hand that has two jacks, three threes, and a four.

The `DoYouHaveAny` method removes the matching cards from the player's hand and returns them—in this case, the three threes.

This second call to `DoYouHaveAny` returns the two jacks and removes them from the player's hand. Make sure your method sorts the cards before you return them so they match the test.

The end of the test checks the cards in the Player's hand and verifies the Status property.

```
[TestMethod]
public void TestAddCardsAndPullOutBooks()
{
    IEnumerable<Card> cards = new List<Card>()
    {
        new Card(Values.Jack, Suits.Spades),
        new Card(Values.Three, Suits.Clubs),
        new Card(Values.Jack, Suits.Hearts),
        new Card(Values.Three, Suits.Hearts),
        new Card(Values.Four, Suits.Diamonds),
        new Card(Values.Jack, Suits.Diamonds),
        new Card(Values.Jack, Suits.Clubs),
    };
    var player = new Player("Owen", cards);
    Assert.AreEqual(0, player.Books.Count());
    var cardsToAdd = new List<Card>()
    {
        new Card(Values.Three, Suits.Diamonds),
        new Card(Values.Three, Suits.Spades),
    };
    player.AddCardsAndPullOutBooks(cardsToAdd);
    var books = player.Books.ToList();
    CollectionAssert.AreEqual(new List<Values>() { Values.Three, Values.Jack }, books);
    var hand = player.Hand.Select(Card => Card.ToString()).ToList();
    CollectionAssert.AreEqual(new List<string>() { "Four of Diamonds" }, hand);
    Assert.AreEqual("Owen has 1 card and 2 books", player.Status);
}

[TestMethod]
public void TestDrawCard()
{
    var player = new Player("Owen", new List<Card>());
    player.DrawCard(new Deck());
    Assert.AreEqual(1, player.Hand.Count());
    Assert.AreEqual("Ace of Diamonds", player.Hand.First().ToString());
}

[TestMethod]
public void TestRandomValueFromHand()
{
    var player = new Player("Owen", new Deck());
    Player.Random = new MockRandom() { ValueToReturn = 0 };
    Assert.AreEqual("Ace", player.RandomValueFromHand().ToString());
    Player.Random = new MockRandom() { ValueToReturn = 4 };
    Assert.AreEqual("Two", player.RandomValueFromHand().ToString());
    Player.Random = new MockRandom() { ValueToReturn = 8 };
    Assert.AreEqual("Three", player.RandomValueFromHand().ToString());
}
}

/// <summary>
/// Mock Random for testing that always returns a specific value
/// </summary>
public class MockRandom : System.Random
{
    public int ValueToReturn { get; set; } = 0;
    public override int Next() => ValueToReturn;
    public override int Next(int maxValue) => ValueToReturn;
    public override int Next(int minValue, int maxValue) => ValueToReturn;
}
```

The `Player.RandomValueFromHand` method uses the `Random` class to generate random values. How do you test a method that relies on a random number? We used a mock object, or a simulated `Random` object that mimics the behavior of the actual .NET `Random` class.

Lucky for us, the `Next` and `NextInt` methods in the `.NET Random` class are virtual, so we created a `MockRandom` class that extends `System.Random` but overrides those methods. We added a `ValueToReturn` property to tell the mock object what int value its `Next` and `NextInt` methods should return. That lets us test methods that rely on random numbers.

Carefully read through the code in this test method—between the test and the XML comments, you can figure out what the `AddCardsAndPullOutBooks` method does.

The `DrawCard` method pulls the next card out of the deck and adds it to the player's hand. What happens if the deck is empty? How would you test that?

We replaced the `Player.Random` reference with a reference to a new `MockRandom` object with `ValueToReturn` set to return a specific value.

Here's our `MockRandom` object that overrides its `int` methods to return a specific value.



## Long Exercise Solution

Here's our code for the Player class. Remember, it's **not cheating** to peek at our solution when you're working on your code! Just make sure you take the time to understand it all.

```
using System.Collections.Generic;
using System.Linq;

public class Player
{
    public static Random Random = new Random();

    private List<Card> hand = new List<Card>();
    private List<Values> books = new List<Values>();

    /// <summary>
    /// The cards in the player's hand
    /// </summary>
    public IEnumerable<Card> Hand => hand;

    /// <summary>
    /// The books that the player has pulled out
    /// </summary>
    public IEnumerable<Values> Books => books;

    public readonly string Name;

    /// <summary>
    /// Pluralize a word, adding "s" if a value isn't equal to 1
    /// </summary>
    public static string S(int s) => s == 1 ? "" : "s";

    /// <summary>
    /// Returns the current status of the player: the number of cards and books
    /// </summary>
    public string Status =>
        $"{Name} has {hand.Count()} card{S(hand.Count())} and {books.Count()} book{S(books.Count())}";

    /// <summary>
    /// Constructor to create a player
    /// </summary>
    /// <param name="name">Player's name</param>
    public Player(string name) => Name = name;

    /// <summary>
    /// Alternate constructor (used for unit testing)
    /// </summary>
    /// <param name="name">Player's name</param>
    /// <param name="cards">Initial set of cards</param>
    public Player(string name, IEnumerable<Card> cards)
    {
        Name = name;
        hand.AddRange(cards);
    }

    /// <summary>
    /// Gets up to five cards from the stock
    /// </summary>
    /// <param name="stock">Stock to get the next hand from</param>
    public void GetNextHand(Deck stock)
    {
        while ((stock.Count() > 0) && (hand.Count < 5))
        {
            hand.Add(stock.Deal(0));
        }
    }
}
```

**There are many ways to solve any programming problem. It's okay if your code looks different than ours, as long as the unit tests pass! For example, we used LINQ methods, but it's absolutely valid to use LINQ query syntax. You don't even have to use LINQ at all! But make sure you take the time to understand our solution, even if you came up with a different (and possibly better!) way to solve the same problem.**

You can use the unit tests figure out exactly what the Status method should return. We used the S method to pluralize "card" and "book" in the status message.

There are lots of ways to get up to 5 cards from the deck. We decided to use a while loop. What did you come up with?

Did you change the private Deck.random field to use Player.Random? If not, do it now:

```
private static Random random = Player.Random;
```

```
/// <summary>
/// If I have any cards that match the value, return them. If I run out of cards, get
/// the next hand from the deck.
/// </summary>
/// <param name="value">Value I'm asked for</param>
/// <param name="deck">Deck to draw my next hand from</param>
/// <returns>The cards that were pulled out of the other player's hand</returns>
public IEnumerable<Card> DoYouHaveAny(Values value, Deck deck)
{
    var matchingCards = hand.Where(card => card.Value == value)
        .OrderBy(Card => Card.Suit);
    hand = hand.Where(card => card.Value != value).ToList();
    if (hand.Count() == 0)
        GetNextHand(deck); ← The rules say that when a player
                           runs out of cards, they need to
                           draw a new hand from the stock.
    return matchingCards;
}
```

We used Where and OrderBy to pull the matching cards out of the hand to return them, then used Where to remove those same cards.

```
/// <summary>
/// When the player receives cards from another player, adds them to the hand
/// and pulls out any matching books
/// </summary>
/// <param name="cards">Cards from the other player to add</param>
public void AddCardsAndPullOutBooks(IEnumerable<Card> cards)
{
    hand.AddRange(cards); ← The first thing the method does is add the cards to the hand.

    var foundBooks = hand
        .GroupBy(card => card.Value)
        .Where(group => group.Count() == 4) } We used GroupBy to group the hand by
        .Select(group => group.Key); } value, then Where to include only the groups
books.AddRange(foundBooks); } that have all four suits, and finally Select
books.Sort(); } to convert each group to its key, the suit.

    hand = hand
        .Where(card => !books.Contains(card.Value))
        .ToList(); }
```

Once the method finds the books, it adds them to its private books field, and then updates its private hand field to remove any cards that match a found book.

```
/// <summary>
/// Draws a card from the stock and add it to the player's hand
/// </summary>
/// <param name="stock">Stock to draw a card from</param>
public void DrawCard(Deck stock)
{
    if (stock.Count > 0)
        AddCardsAndPullOutBooks(new List<Card>() { stock.Deal(0) });
}
```

DrawCard needs to pull out the books after it deals a card. Can you figure out how to add a unit test to make sure that works?

We sorted the hand by value so the test will always start with the hand in the same order.

```
/// <summary>
/// Gets a random value from the player's hand
/// </summary>
/// <returns>The value of a randomly selected card in the player's hand</returns>
public Values RandomValueFromHand() => hand.OrderBy(card => card.Value)
    .Select(card => card.Value)
    .Skip(Random.Next(hand.Count())))
    .First();

    public override string ToString() => Name;
```

To get a random value from the hand, we used the Select method to convert each card to its value, then skipped a random number of cards and got the next one.



I CAN USE UNIT TESTS TO MAKE SURE ONE CLASS WORKS BEFORE MOVING ON TO THE NEXT ONE, SO I CAN CHOOSE TO IMPLEMENT THE CLASSES IN ANY ORDER I WANT.

### Unit tests let you develop code your own way.

One of the most challenging parts of real-world software development is figuring out how to manage your projects, and unit tests can help you do that. At the beginning of the book, you were doing small projects, so you didn't really need to plan your approach. But now that you're doing much larger projects, you need to take a more systematic approach. Unit tests can help you choose an approach that works well for your project because they let you be **flexible** about the order that you build your classes. They let you choose which part of the code to work on first, and give you a good stopping point—all unit tests for that part of the code pass—so you can be confident moving on to the next part of the project.

### Test-driven development means writing unit tests first

Unit tests help you take on larger projects by giving you the flexibility to choose what part of the code to work on first, and a good stopping point for that part of the code so you can more easily break the project up into parts—which is what we did with this project.

But we did something else that's even more important: we had you *create the unit tests first*. We gave you the skeleton of the Player class, then we gave you a unit test so you could see exactly what it's supposed to do. It was your job to write the code for the Player class to make it pass the tests. When you write unit tests first, it's called **test-driven development** (or TDD).

You'll use test-driven development to build the GameState and GameController classes. We'll give you their unit tests, just like we did with the Player class, and you'll use those tests to figure out exactly what the classes are supposed to do.

You can do test-driven development on any project! It's a great way to make sure you really understand what your classes are supposed to do, and you end up with a lot fewer bugs than you would without it.



## LONG Exercise

**Part 2: Create the GameState class.** Here's a skeleton for the GameState class. Like before, we gave you a **skeleton**—we gave you the fields and properties, and it's up to you to implement the methods that throw NotImplementedExceptions.

```
using System.Collections.Generic;
using System.Linq;

public class GameState
{
    public readonly IEnumerable<Player> Players;
    public readonly IEnumerable<Player> Opponents;
    public readonly Player HumanPlayer;
    public bool GameOver { get; private set; } = false;

    public readonly Deck Stock;

    /// <summary>
    /// Constructor creates the players and deals their first hands
    /// </summary>
    /// <param name="humanPlayerName">Name of the human player</param>
    /// <param name="opponentNames">Names of the computer players</param>
    /// <param name="stock">Shuffled stock of cards to deal from</param>
    public GameState(string humanPlayerName, IEnumerable<string> opponentNames, Deck stock)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Gets a random player that doesn't match the current player
    /// </summary>
    /// <param name="currentPlayer">The current player</param>
    /// <returns>A random player that the current player can ask for a card</returns>
    public Player RandomPlayer(Player currentPlayer) =>
        throw new NotImplementedException();

    /// <summary>
    /// Makes one player play a round
    /// </summary>
    /// <param name="player">The player asking for a card</param>
    /// <param name="playerToAsk">The player being asked for a card</param>
    /// <param name="valueToAskFor">The value to ask the player for</param>
    /// <param name="stock">The stock to draw cards from</param>
    /// <returns>A message that describes what just happened</returns>
    public string PlayRound(Player player, Player playerToAsk,
                           Values valueToAskFor, Deck stock)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Checks for a winner by seeing if any players have any cards left, sets GameOver
    /// if the game is over and there's a winner
    /// </summary>
    /// <returns>A string with the winners, an empty string if there are no winners</returns>
    public string CheckForWinner()
    {
        throw new NotImplementedException();
    }
}
```



## Long Exercise

**Part 2 (continued): Add the GameState class unit tests.** Here's the complete `GameStateTests` class. In addition to tests for each method in the class, it also includes a separate test for the constructor, because it's more complex than the `Player` constructor.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;
using System.Linq;
using GoFish;

[TestClass]
public class GameStateTests
{
    [TestMethod]
    public void TestConstructor()
    {
        var computerPlayerNames = new List<string>()
        {
            "Computer1",
            "Computer2",
            "Computer3",
        };
        var gameState = new GameState("Human", computerPlayerNames, new Deck());
        CollectionAssert.AreEqual(
            new List<string> { "Human", "Computer1", "Computer2", "Computer3" },
            gameState.Players.Select(player => player.Name).ToList());
        Assert.AreEqual(5, gameState.HumanPlayer.Hand.Count()); ← The GameState constructor calls each player's GetNextHand method to deal their initial hand. We already tested that method in PlayerTests, so we didn't include an in-depth test for it here.
    }

    [TestMethod]
    public void TestRandomPlayer()
    {
        var computerPlayerNames = new List<string>()
        {
            "Computer1",
            "Computer2",
            "Computer3",
        }; ← To test the RandomPlayer method, we set up a GameState, then used the MockRandom object to get RandomPlayer to return a specific player.
        var gameState = new GameState("Human", computerPlayerNames, new Deck());
        Player.Random = new MockRandom() { ValueToReturn = 1 };
        Assert.AreEqual("Computer2",
            gameState.RandomPlayer(gameState.Players.ToList()[0]).Name);

        Player.Random = new MockRandom() { ValueToReturn = 0 };
        Assert.AreEqual("Human",
            gameState.RandomPlayer(gameState.Players.ToList()[1]).Name);
        Assert.AreEqual("Computer1",
            gameState.RandomPlayer(gameState.Players.ToList()[0]).Name);
    }

    [TestMethod]
    public void TestPlayRound()
    {
        var deck = new Deck();
        deck.Clear();
        var cardsToAdd = new List<Card>() { ← We test the PlayRound method by setting up a deck to deal to our Owen and Brittany players. Once the deck is set up, we create a GameState with the two players and call PlayRound to play out the rounds.
            // Cards the game will deal to Owen
            new Card(Values.Jack, Suits.Spades),
            new Card(Values.Jack, Suits.Hearts),
            new Card(Values.Six, Suits.Spades),
            new Card(Values.Jack, Suits.Diamonds),
            new Card(Values.Six, Suits.Hearts),
        };
    }
}
```

```

// Cards the game will deal to Brittney
new Card(Values.Six, Suits.Diamonds),
new Card(Values.Six, Suits.Clubs),
new Card(Values.Seven, Suits.Spades),
new Card(Values.Jack, Suits.Clubs),
new Card(Values.Nine, Suits.Spades),

// Two more cards in the deck for Owen to draw when he runs out
new Card(Values.Queen, Suits.Hearts),
new Card(Values.King, Suits.Spades),
};

foreach (var card in cardsToAdd)
    deck.Add(card);

var gameState = new GameState("Owen", new List<string>() { "Brittney" }, deck);

var owen = gameState.HumanPlayer;
var brittney = gameState.Opponents.First();

Assert.AreEqual("Owen", owen.Name);
Assert.AreEqual(5, owen.Hand.Count());
Assert.AreEqual("Brittney", brittney.Name);
Assert.AreEqual(5, brittney.Hand.Count());

var message = gameState.PlayRound(owen, brittney, Values.Jack, deck);
Assert.AreEqual("Owen asked Brittney for Jacks" + Environment.NewLine +
    "Brittney has 1 Jack card", message);
Assert.AreEqual(1, owen.Books.Count());
Assert.AreEqual(2, owen.Hand.Count());
Assert.AreEqual(0, brittney.Books.Count());
Assert.AreEqual(4, brittney.Hand.Count());

message = gameState.PlayRound(brittney, owen, Values.Six, deck);
Assert.AreEqual("Brittney asked Owen for Sixes" + Environment.NewLine +
    "Owen has 2 Six cards", message);
Assert.AreEqual(1, owen.Books.Count());
Assert.AreEqual(2, owen.Hand.Count());
Assert.AreEqual(1, brittney.Books.Count());
Assert.AreEqual(2, brittney.Hand.Count());

message = gameState.PlayRound(owen, brittney, Values.Queen, deck);
Assert.AreEqual("Owen asked Brittney for Queens" + Environment.NewLine +
    "The stock is out of cards", message);
Assert.AreEqual(1, owen.Books.Count());
Assert.AreEqual(2, owen.Hand.Count());
}

[TestMethod]
public void TestCheckForAWinner()
{
    var computerPlayerNames = new List<string>()
    {
        "Computer1",
        "Computer2",
        "Computer3",
    };

    var emptyDeck = new Deck();
    emptyDeck.Clear();
    var gameState = new GameState("Human", computerPlayerNames, emptyDeck);
    Assert.AreEqual("The winners are Human and Computer1 and Computer2 and Computer3",
        gameState.CheckForWinner());
}
}

Can you think of additional ways to test that the CheckForAWinner method works? Try writing another unit test for that method.

```

Here's where we set up the deck, then create the GameState with one human player (Owen) and one computer player (Brittney).

Next we make sure the GameState was set up correctly, with hands of five cards dealt to each of the two players.

In the first round, Owen asks Brittney for Jacks. We set up the deck so that Brittney has one jack.

Look closely at the message that the PlayRound method returns. Your PlayRound method should return a message that looks just like this. Notice how "Sixes" is spelled correctly.

We're using Environment.NewLine to add line breaks (instead of @ verbatim strings) because we want this code to work on both Mac and Windows, and your test will fail if it tries to compare \n against \r\n.

We checked for a winner by setting up a GameState with an empty deck, so all of the players would be dealt empty hands. They all have the same number of books, so they'll all be winners.



## Long Exercise Solution

Here's our code for the GameState class. It has a constructor and methods to pick a random player, play a round, and check for a winner.

```
using System.Collections.Generic;
using System.Linq;

public class GameState
{
    public readonly IEnumerable<Player> Players;
    public readonly IEnumerable<Player> Opponents;
    public readonly Player HumanPlayer;
    public bool GameOver { get; private set; } = false;

    public readonly Deck Stock;

    /// <summary>
    /// Constructor creates the players and deals their first hands
    /// </summary>
    /// <param name="humanPlayerName">Name of the human player</param>
    /// <param name="opponentNames">Names of the computer players</param>
    /// <param name="stock">Shuffled stock of cards to deal from</param>
    public GameState(string humanPlayerName, IEnumerable<string> opponentNames, Deck stock)
    {
        this.Stock = stock;

        HumanPlayer = new Player(humanPlayerName);
        HumanPlayer.GetNextHand(Stock);

        var opponents = new List<Player>();
        foreach (string name in opponentNames)
        {
            var player = new Player(name);
            player.GetNextHand(stock);
            opponents.Add(player);
        }
        Opponents = opponents;
        Players = new List<Player>() { HumanPlayer }.Concat(Opponents);
    }

    /// <summary>
    /// Gets a random player that doesn't match the current player
    /// </summary>
    /// <param name="currentPlayer">The current player</param>
    /// <returns>A random player that the current player can ask for a card</returns>
    public Player RandomPlayer(Player currentPlayer) =>
        Players
            .Where(player => player != currentPlayer)
            .Skip(Player.Random.Next(Players.Count() - 1))
            .First();
    
```

Create the Player object for the human player and draw its next hand from the shuffled stock of cards.

Create the Player object for each computer player and draw their cards.

We used the LINQ Concat method to create the list of all players (human and computer).

We used the LINQ methods to get a random player from the list of players. First we use Where to make sure we're picking a player who isn't the current player, then we skip a random number of players, and pull the first player from the list.

```

/// <summary>
/// Makes one player play a round
/// </summary>
/// <param name="player">The player asking for a card</param>
/// <param name="playerToAsk">The player being asked for a card</param>
/// <param name="valueToAskFor">The value to ask the player for</param>
/// <param name="stock">The stock to draw cards from</param>
/// <returns>A message that describes what just happened</returns>
public string PlayRound(Player player, Player playerToAsk,
    Values valueToAskFor, Deck stock)
{
    var valuePlural = (valueToAskFor == Values.Six) ? "Sixes" : $"{valueToAskFor}s";
    var message = $"{player.Name} asked {playerToAsk.Name}"
        + $" for {valuePlural}{Environment.NewLine}";
    var cards = playerToAsk.DoYouHaveAny(valueToAskFor, stock);
    if (cards.Count() > 0)
    {
        player.AddCardsAndPullOutBooks(cards);
        message += $"{playerToAsk.Name} has {cards.Count()}"
            + $" {valueToAskFor} card{Player.S(cards.Count())}";
    }
    else if (stock.Count == 0)
    {
        message += $"The stock is out of cards"; ↴ The PlayRound method relies on
    }                                         the methods you already added
    else                                         to the Player class to ask another
    {
        player.DrawCard(stock);                ← player for a card, add those
        message += $"{player.Name} drew a card"; cards and pull out books or draw
    }                                         a card from the stock, and get
    if (player.Hand.Count() == 0)             ← the next hand if the player is out.
    {
        player.GetNextHand(stock);
        message += $"{Environment.NewLine}{player.Name} ran out of cards,"
            + $" drew {player.Hand.Count()} from the stock";
    }
    return message;
}

/// <summary>
/// Checks for a winner by seeing if any players have any cards left, sets GameOver
/// if the game is over and there's a winner
/// </summary>
/// <returns>A string with the winners, an empty string if there are no winners</returns>
public string CheckForWinner()
{
    var playerCards = Players.Select(player => player.Hand.Count()).Sum();
    if (playerCards > 0) return "";
    GameOver = true;
    var winningBookCount = Players.Select(player => player.Books.Count()).Max();
    var winners = Players.Where(player => player.Books.Count() == winningBookCount);
    if (winners.Count() == 1) return $"The winner is {winners.First().Name}";
    return $"The winners are {string.Join(" and ", winners)}";
}

```

We used the conditional operator to make the message correctly use the word "Sixes"

The PlayRound method relies on the methods you already added to the Player class to ask another player for a card, add those cards and pull out books or draw a card from the stock, and get the next hand if the player is out.

## Long Exercise

 Part 3: Add the GameController class and unit tests. Here's the skeleton for the GameController class, followed by the GameControllerTests class with unit tests for the constructor and its two methods, NextRound and NewGame. The NextRound method calls a private ComputerPlayersPlayRound method.

```
using System.Collections.Generic;
using System.Linq;

public class GameController
{
    public static Random Random = new Random();

    private GameState gameState;
    public bool GameOver { get { return gameState.GameOver; } }
    public Player HumanPlayer { get { return gameState.HumanPlayer; } }
    public IEnumerable<Player> Opponents { get { return gameState.Opponents; } }

    public string Status { get; private set; } ← The Status property is important. The constructor, NextRound, and NewGame methods update it so the app can use it to write messages for the player.

    /// <summary>
    /// Constructs a new GameController
    /// </summary>
    /// <param name="humanPlayerName">Name of the human player</param>
    /// <param name="computerPlayerNames">Names of the computer players</param>
    public GameController(string humanPlayerName, IEnumerable<string> computerPlayerNames)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Plays the next round, ending the game if everyone ran out of cards
    /// </summary>
    /// <param name="playerToAsk">Which player the human is asking for a card</param>
    /// <param name="valueToAskFor">The value of the card the human is asking for</param>
    public void NextRound(Player playerToAsk, Values valueToAskFor)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// All of the computer players that have cards play the next round. If the human is
    /// out of cards, then the deck is depleted and they play out the rest of the game.
    /// </summary>
    private void ComputerPlayersPlayNextRound()
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Starts a new game with the same player names
    /// </summary>
    public void NewGame()
    {
        throw new NotImplementedException();
    }
}
```

Unit tests only test public class members.  
We included a private method called ComputerPlayersPlayNextRound, which is called by NextRound. You won't test that method directly—but you'll know that it works correctly if the unit test for the NextRound method passes.

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;
using GoFish;
using System.Linq;

[TestClass]
public class GameControllerTests
{
    [TestInitialize]
    public void Initialize()
    {
        Player.Random = new MockRandom() { ValueToReturn = 0 };
    }

    [TestMethod]
    public void TestConstructor()
    {
        var gameController = new GameController("Human",
            new List<string>() { "Player1", "Player2", "Player3" });
        Assert.AreEqual("Starting a new game with players Human, Player1, Player2, Player3",
            gameController.Status);
    }

    [TestMethod]
    public void TestNextRound()
    {
        // The constructor shuffles the deck, but MockRandom makes sure it stays in order
        // so Owen should have Ace to 5 of Diamonds, Brittney should have 6 to 10 of Diamonds
        var gameController = new GameController("Owen", new List<string>() { "Brittney" });

        gameController.NextRound(gameController.Opponents.First(), Values.Six);
        Assert.AreEqual("Owen asked Brittney for Sixes" +
            Environment.NewLine + "Brittney has 1 Six card" +
            Environment.NewLine + "Brittney asked Owen for Sevens" +
            Environment.NewLine + "Brittney drew a card" +
            Environment.NewLine + "Owen has 6 cards and 0 books" +
            Environment.NewLine + "Brittney has 5 cards and 0 books" +
            Environment.NewLine + "The stock has 41 cards" +
            Environment.NewLine, gameController.Status);
    }

    [TestMethod]
    public void TestNewGame()
    {
        Player.Random = new MockRandom() { ValueToReturn = 0 };
        var gameController = new GameController("Owen", new List<string>() { "Brittney" });
        gameController.NextRound(gameController.Opponents.First(), Values.Six);
        gameController.NewGame();
        Assert.AreEqual("Owen", gameController.HumanPlayer.Name);
        Assert.AreEqual("Brittney", gameController.Opponents.First().Name);
        Assert.AreEqual("Starting a new game", gameController.Status);
    }
}

```

We need to set `Player.Random` to a new `MockRandom` that always returns 0 to make sure the deck is in order and the players always pick the same “random” value from their hands. We put this in a method marked with the `[TestInitialize]` attribute, which tells MSTest to always run that method before running any of the tests in the class.

The constructor test checks to make sure the `Status` property is updated correctly after the `GameController` is instantiated.

The `NextRound` method uses the `GameState.RandomPlayer` and `Player.RandomValueFromHand` methods to make the computer players choose a random value to ask for and a player to ask, so we'll use `MockRandom` to test it.

The `NextRound` method calls the `GameState` method to make the human player to play the next round, then calls the private `ComputerPlayersPlayNextRound` method to make the computer players play. All the test needs to do is check the `Status` property—if the status matches the expected result of the first round we can be comfortable that the method works.

Starting a new game causes `GameController` to create a new `GameState` with a newly shuffled Deck (which will actually be in order because we're using `MockRandom`).

`NextRound` eventually calls each `Player` object's `RandomValueFromHand` method. We made sure it sorts the hand before picking a random value, so when we use `MockRandom` it will always pick the same “random” values for the test.



Here's our code for the Player class. Remember, it's **not cheating** to peek at our solution when you're working on your code! Just make sure you take the time to understand it all.

```
using System.Collections.Generic;
using System.Linq;

public class GameController
{
    private GameState gameState;
    public bool GameOver { get { return gameState.GameOver; } }
    public Player HumanPlayer { get { return gameState.HumanPlayer; } }
    public IEnumerable<Player> Opponents { get { return gameState.Opponents; } }

    public string Status { get; private set; }

    /// <summary>
    /// Constructs a new GameController
    /// </summary>
    /// <param name="humanPlayerName">Name of the human player</param>
    /// <param name="computerPlayerNames">Names of the computer players</param>
    public GameController(string humanPlayerName, IEnumerable<string> computerPlayerNames)
    {
        gameState = new GameState(humanPlayerName, computerPlayerNames, new Deck().Shuffle());
        Status = $"Starting a new game with players {string.Join(", ", gameState.Players)}";
    }

    /// <summary>
    /// Plays the next round, ending the game if everyone ran out of cards
    /// </summary>
    /// <param name="playerToAsk">Which player the human is asking for a card</param>
    /// <param name="valueToAskFor">The value of the card the human is asking for</param>
    public void NextRound(Player playerToAsk, Values valueToAskFor)
    {
        Status = gameState.PlayRound(gameState.HumanPlayer, playerToAsk,
            valueToAskFor, gameState.Stock) + Environment.NewLine;

        ComputerPlayersPlayNextRound();

        Status += string.Join(Environment.NewLine,
            gameState.Players.Select(player => player.Status));
        Status += $"{Environment.NewLine}The stock has {gameState.Stock.Count()} cards";

        Status += Environment.NewLine + gameState.CheckForWinner();
    }
}
```

```
/// <summary>
/// All of the computer players that have cards play the next round. If the human is
/// out of cards, then the deck is depleted and they play out the rest of the game.
/// </summary>
private void ComputerPlayersPlayNextRound()
{
    IEnumerable<Player> computerPlayersWithCards;
    do
    {
        computerPlayersWithCards =
            gameState
                .Opponents
                .Where(player => player.Hand.Count() > 0);
        foreach (Player player in computerPlayersWithCards)
        {
            var randomPlayer = gameState.RandomPlayer(player);
            var randomValue = player.RandomValueFromHand();
            Status += gameState
                .PlayRound(player, randomPlayer, randomValue, gameState.Stock)
                + Environment.NewLine;
        }
    } while ((gameState.HumanPlayer.Hand.Count() == 0)
        && (computerPlayersWithCards.Count() > 0));
}

/// <summary>
/// Starts a new game with the same player names
/// </summary>
public void NewGame()
{
    Status = "Starting a new game";
    gameState = new GameState(gameState.HumanPlayer.Name,
        gameState.Opponents.Select(player => player.Name),
        new Deck().Shuffle());
}
```

Here's a great opportunity to get some practice writing unit tests. Can you come up with more tests for your Player, GameState, and GameController classes?



## Long Exercise

**Part 4: Add the Program class.** Now that the “guts” of the game are done, it’s time to finish the project. Here’s a sample run of the game. It starts by prompting the user for their name and the number of computer opponents (which must be between 1 and 5). Then it plays each round, writing the cards in the player’s hand to the console, then prompting for a card to ask for (which must be in the player’s hand) and an opponent to ask for a card. To finish the round, it calls GameController.NextRound and displays GameController.Status. When the game is over, it asks the player to press Q to quit, or any other key for a new game.

There aren’t unit tests for the Program class. Look closely at the output and create the Program class with a Main method that generates matching output. We’ve given you a skeleton for the Program class as a starting point.

```
Enter your name: Andrew
Enter the number of computer opponents: 4
Welcome to the game, Andrew
Starting a new game with players Human,
Computer #1, Computer #2, Computer #3,
Computer #4
Your hand:
Ace of Clubs
Three of Hearts
Six of Diamonds
Six of Spades
Ten of Hearts
What card value do you want to ask for? Six
1. Computer #1
2. Computer #2
3. Computer #3
4. Computer #4
Who do you want to ask for a card? 2
Human asked Computer #2 for Sixes
Human drew a card
Computer #1 asked Computer #3 for Threes
Computer #1 drew a card
Computer #2 asked Human for Queens
Computer #2 drew a card
Computer #3 asked Computer #1 for Twos
Computer #3 drew a card
Computer #4 asked Computer #2 for Tens
Computer #4 drew a card
Human has 6 cards and 0 books
Computer #1 has 6 cards and 0 books
Computer #2 has 6 cards and 0 books
Computer #3 has 6 cards and 0 books
Computer #4 has 6 cards and 0 books
The stock has 22 cards
```

```
class Program
{
    /// <summary>
    /// Play a game of Go Fish!
    /// </summary>
    static void Main(string[] args)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// The GameController to manage the game
    /// </summary>
    static GameController gameController;

    /// <summary>
    /// Prompt the human player for a card value
    /// in their hand
    /// </summary>
    /// <returns>The value to ask for</returns>
    static Values PromptForAValue()
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Prompt the human player for an opponent
    /// to ask for a card
    /// </summary>
    /// <returns>The opponent to ask</returns>
    static Player PromptForAnOpponent()
    {
        throw new NotImplementedException();
    }
}
```

Your hand:  
 Ace of Clubs  
 Three of Hearts  
 Six of Diamonds  
 Six of Spades  
 Seven of Hearts  
 Ten of Hearts

What card value do you want to ask for? **Ten**

1. Computer #1
2. Computer #2
3. Computer #3
4. Computer #4

Who do you want to ask for a card? **4**

Human asked Computer #4 for Tens

Computer #4 has 1 Ten card

Computer #1 asked Computer #3 for Jacks

Computer #3 has 1 Jack card

Computer #2 asked Computer #1 for Twos

Computer #2 drew a card

Computer #3 asked Computer #1 for Twos

Computer #3 drew a card

Computer #4 asked Computer #2 for Sevens

Computer #2 has 1 Seven card

Human has 7 cards and 0 books

Computer #1 has 7 cards and 0 books

Computer #2 has 6 cards and 0 books

Computer #3 has 6 cards and 0 books

Computer #4 has 6 cards and 0 books

The stock has 20 cards

Your hand:  
 Ace of Clubs  
 Three of Hearts  
 Six of Diamonds  
 Six of Spades  
 Seven of Hearts  
 Ten of Hearts  
 Ten of Spades

What card value do you want to ask for? **Seven**

1. Computer #1
2. Computer #2
3. Computer #3
4. Computer #4

Who do you want to ask for a card? **4**

Human asked Computer #4 for Sevens

Computer #4 has 2 Seven cards

Computer #1 asked Computer #2 for Three

Computer #1 drew a card

Computer #2 asked Computer #1 for Queens

Computer #2 drew a card

Computer #3 asked Computer #4 for Eight

Computer #4 has 1 Eight card

Computer #4 asked Computer #2 for Jacks  
 Computer #4 drew a card  
 Human has 9 cards and 0 books  
 Computer #1 has 8 cards and 0 books  
 Computer #2 has 7 cards and 0 books  
 Computer #3 has 7 cards and 0 books  
 Computer #4 has 4 cards and 0 books  
 The stock has 17 cards

Your hand:  
 Ace of Clubs  
 Three of Hearts  
 Six of Diamonds  
 Six of Spades  
 Seven of Diamonds  
 Seven of Hearts  
 Seven of Spades  
 Ten of Hearts  
 Ten of Spades

What card value do you want to ask for? **Three**

1. Computer #1
2. Computer #2
3. Computer #3
4. Computer #4

Who do you want to ask for a card? **1**

Human asked Computer #1 for Threes

Computer #1 has 3 Three cards

Computer #1 asked Computer #3 for Eight

Computer #1 drew a card

Computer #2 asked Computer #3 for Twos

Computer #3 has 1 Two card

Computer #3 asked Human for Kings

Computer #3 drew a card

Computer #4 asked Computer #2 for Sixes

Computer #4 drew a card

Human has 8 cards and 1 book

Computer #1 has 6 cards and 0 books

Computer #2 has 8 cards and 0 books

Computer #3 has 7 cards and 0 books

Computer #4 has 5 cards and 0 books

The stock has 14 cards

Your hand:  
 Ace of Clubs  
 Six of Diamonds  
 Six of Spades  
 Seven of Diamonds  
 Seven of Hearts  
 Seven of Spades  
 Ten of Hearts  
 Ten of Spades

What card value do you want to ask for?



```

class Program
{
    /// <summary>
    /// Play a game of Go Fish!
    /// </summary>
    static void Main(string[] args)
    {
        Console.Write("Enter your name: ");
        var humanName = Console.ReadLine();

        Console.Write("Enter the number of computer opponents: ");
        int opponentCount;
        while (!int.TryParse(Console.ReadKey().KeyChar.ToString(), out opponentCount)
            || opponentCount < 1 || opponentCount > 4)
        {
            Console.WriteLine("Please enter a number from 1 to 4");
        }
        Console.WriteLine($"{Environment.NewLine}Welcome to the game, {humanName}");

        gameController = new GameController(humanName,
            Enumerable.Range(1, opponentCount).Select(i => $"Computer #{i}"));
        Console.WriteLine(gameController.Status);

        while (!gameController.GameOver)
        {
            while (!gameController.GameOver)
            {
                Console.WriteLine($"Your hand:");
                foreach (var card in gameController.HumanPlayer.Hand
                    .OrderBy(card => card.Suit)
                    .OrderBy(card => card.Value))
                    Console.WriteLine(card);

                var value = PromptForAValue();

                var player = PromptForAnOpponent();
                gameController.NextRound(player, value);
                Console.WriteLine(gameController.Status);
            }

            Console.WriteLine("Press Q to quit, or any other key for a new game.");
            if (Console.ReadKey(true).KeyChar.ToString().ToUpper() == "N")
                gameController.NewGame();
        }
    }
}

```

Here's our Program class with the entry point. Here's a good example of **separation of concerns**: the Program class takes care of the input and output, while the GameController handles the actual gameplay, and it, in turn, depends on GameState and Player to do their parts.

This foreach loop uses LINQ to put the cards in suit and value order, then writes them to the console.

After the program gets the input from the player, it tells GameController to play the next round.

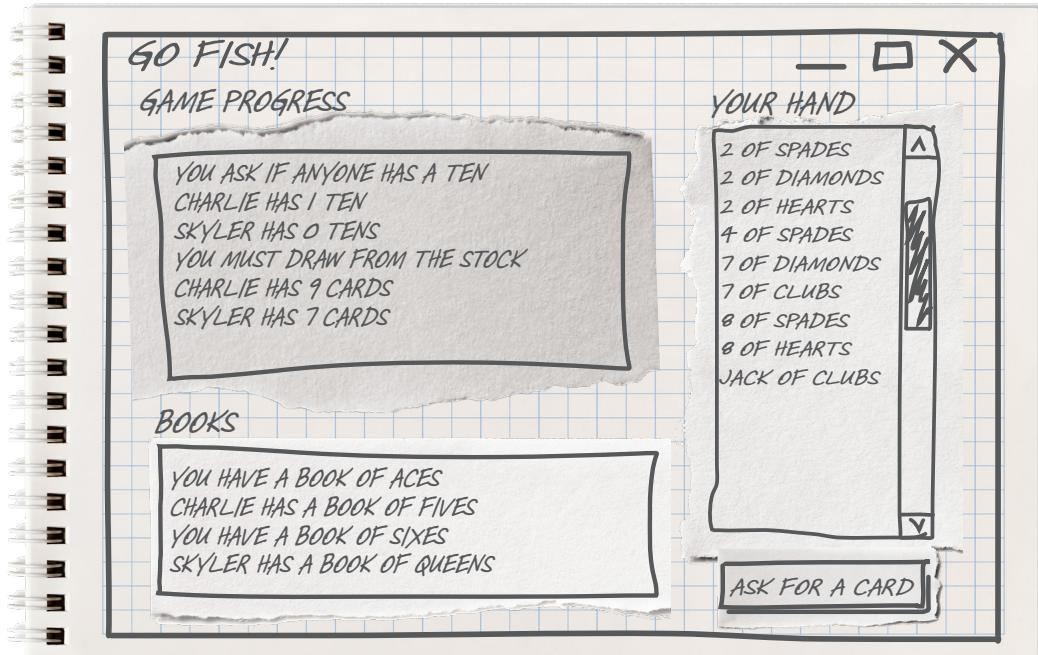
```
/// <summary>
/// The GameController to manage the game
/// </summary>
static GameController gameController;

/// <summary>
/// Prompt the human player for a card value in their hand
/// </summary>
/// <returns>The value to ask for</returns>
static Values PromptForAValue()
{
    var handValues = gameController.HumanPlayer.Hand.Select(card => card.Value).ToList();
    Console.WriteLine("What card value do you want to ask for? ");
    while (true)
    {
        if (Enum.TryParse(typeof(Values), Console.ReadLine(), out var value) &&
            handValues.Contains((Values)value))
            return (Values)value;
        else
            Console.WriteLine("Please enter a value in your hand.");
    }
}

/// <summary>
/// Prompt the human player for an opponent to ask for a card
/// </summary>
/// <returns>The opponent to ask</returns>
static Player PromptForAnOpponent()
{
    var opponents = gameController.Opponents.ToList();
    for (int i = 1; i <= opponents.Count(); i++)
        Console.WriteLine($"{i}. {opponents[i - 1]}");
    Console.WriteLine("Who do you want to ask for a card? ");
    while (true)
    {
        if (int.TryParse(Console.ReadLine(), out int selection)
            && selection >= 1 && selection <= opponents.Count())
            return opponents[selection - 1];
        else
            Console.WriteLine($"Please enter a number from 1 to {opponents.Count()}: ");
    }
}
```

## Can you use the same classes to build a UI?

We put together a simple paper prototype for a UI. But we didn't finish it—it has a button to ask for a card, but it still needs a way for the player to choose which opponent to ask. So here's a **code challenge** for you! Can you use this project as a starting point to build a WPF or Blazor version of the *Go Fish!* game?



### Here's what you'll need to do...

- ★ Create your own paper prototype, and figure out how you want to prompt the player for an opponent to ask.
- ★ Add either a WPF or Blazor WebAssembly project to your GoFish solution.
- ★ Modify its project dependencies to add a dependency on the GoFish project so it can see GameController, Card, Deck, and the other classes and enums.
- ★ Create the XAML window or HTML page that has an instance of GameController and bind the game progress to its Status property.
- ★ Create event handlers to get the input and play the next round.
- ★ When the game is over, prompt the user to reset GameController and start a new game.

**Did you come up with a creative and interesting solution to this code challenge?  
We'd love to see it! Tell us about it on Twitter [@HeadFirstCSharp](#)**