This sample of C#eckmate: Learning C# and Programming Chess is provided free of charge. I hope the contents will demonstrate my writing style and general direction of the book, and perhaps convince you to support this independent author and make a purchase.

You will see some apparent errors in this text, notably as pairs of question marks ("??"). These indicate a broken reference, where a sample chapter is referring to a page, section, or subject from the full book that is not in the sample. Rest assured that those references are complete and accurate in the full version of the book.

You can find a full copy of this text at `https://leanpub.com/checkmate-csharp`.

Introduction

Welcome to *C#eckmate: Learning C# and Programming Chess*!

The inspiration for this book is twofold. In 2003 I wanted to learn ASP.NET, and was fortunate to find a book called *ASP.NET Website Programming: Problem, Design, Solution* by Marco Bellinaso and Kevin Hoffman. Instead of simply walking through ASP.NET's APIs and repackaging official documentation, Bellinaso and Hoffman's book showed how to build an interactive website from scratch, introducing ASP.NET topics as they became relevant to the implementation of a content-driven dynamic website. Their chapters followed a "problem, design, solution" pattern, in which they introduced a problem to be solved ("the site needs user accounts"), explored design topics related to the problem (authentication, permissions, profiling, administration, etc.), and then developed a full-source-code solution to the problem (including C# server code, SQL database queries, and HTML/JavaScript for the frontend). Their approach taught me ASP.NET much faster than a more traditional technical book would have, and the lessons I learned from their examples helped me develop my own successful ASP.NET-powered websites. When I first thought about writing a programming book, I knew it would need to follow in Bellinaso and Hoffman's footsteps, implementing a *real project* in the course of teaching some more formal topic.

But what would that project be? My second inspiration comes from projects given to me as a computer science undergraduate at California Polytechnic State University, San Luis Obispo. One project in particular – implementing Othello in C++ – has inspired many of the assignments I give my students, and now serves as a foundation for this book. I love involving board games in the programming courses that I teach, as their implementations typically combine a variety of programming topics into a single assignment that many students find motivating because they enjoy playing games in their social time. I recently taught a class about C# and .NET programming in which students were tasked with implementing chess; at the end of that course, I knew that I had found the right project for my book idea.

As an educator, I believe in in the *learn by doing* philosophy. The world doesn't need another C# reference text with 47 chapters and arbitrary, disjointed examples that do little more than prove the author got their code to compile. Passive learning by reading dry texts works for some people, but most benefit from a more active learning approach. When I teach a programming course, I find that students learn best when assigned challenging projects that incorporate not only lessons from lecture, but also learning opportunities from other disciplines. So while this book has the same goal as many other texts – learning the C# programming language – we'll approach it in an atypical way: by applying lessons from our study of the language to develop a sophisticated application throughout the length of the book. Along the way we will confront challenges in software engineering and architecture, algorithm design, computer security, cross-platform development, and much more. And by applying solutions to these challenges to a larger context, drawing on the reader's background experience and the writer's guiding hand, it is my

hope that those lessons will stick more successfully than if they were presented as end-of-chapter throw-away exercises.

Goals and Topics

I want this text to serve two purposes: learning the C# programming language, and programming a sophisticated implementation of chess.

Why chess? The game is complicated yet familiar. Piece movements are easy to program, but with enough patterns and complexity to make effective demonstrations of various software engineering and architecture techniques. There will be many tricks to explore when deciding how to represent a chess game's state, and many of C#'s features will simplify our designs and improve the cohesion and coupling of our code. We'll learn about design patterns when fitting chess into a generic board game-playing application, and find a visceral pleasure when we program a user interface for the game, complete with an artificial intelligence opponent. In short, chess (and board games in general) ties together a great number of topics in computer science, and the sophisticated implementation we'll cover in this book should provide a number of learning opportunities beyond simple proficiency in the C# programming language.

Learning C#

Part I of this book is aimed at intermediate programmers who are experience in an object-oriented programming language like Java or Python. After introducing the .NET Framework and the Visual Studio IDE, we will dive into the C# programming language from an experienced learner's perspective. We won't waste time with what a variable is, or what an `if` statement is for, or what a class is... rather, we will fast-forward to the important parts of the language, frequently contrasting it with Java and Python. You can skip Part I if you already know C#, but a quick skim might be warranted.

Programming chess

Parts II through **??** – while containing lessons on C# and many of of the .NET libraries – are focused on implementing two board games: Othello[1] and chess[2].

Part II proposes a software framework for a board game-playing console application. The application will support (theoretically) any two-player board game, but we will focus on one example game (Othello) in particular. Othello is a moderately difficult game to program, and studying its design in software will hopefully yield insights that will help us design an implementation of the much more difficult game of chess. After concluding Othello's design, we'll introduce a rigorous testing framework for validating our Othello code and the chess code that we've yet to write. Part

[1] https://en.wikipedia.org/wiki/Reversi
[2] https://en.wikipedia.org/wiki/Chess

II then concludes with a lengthy chapter on designing our chess implementation, with extensions to the testing framework to validate our newest work.

Part **??** introduces the goal of a graphical user interface application. We will learn enough of Windows Presentation Foundation – Microsoft's modern user interface library for .NET on Windows – to program an interface for Othello, and then once again turn our attention to chess.

Part **??** aims to integrate a worthy artificial intelligence opponent into our game-playing application. We will introduce the Minimax algorithm for decision-making, then apply it to our general board game framework and integrate the resulting AI opponent into our console and GUI applications. If we find the resulting opponent lacking, we will go back and consider ways of optimizing our chess implementation without changing the public interface developed in Part II.

Problem, Design, Implementation

Parts II and **??** of the book will be organized in a *problem, design, implementation* pattern. Each chapter will begin by identifying a problem that must be solved to implement a generic board game-playing application. We will then apply lessons from software architecture, engineering, and testing to design a solution to our problems. An implementation of the design will follow.

Target Audience

This is not a book for beginning programmers. The chapters on learning C# will assume a general familiarity with imperative and object-oriented programming, equivalent to 2-3 years of programming courses in a university computer science curriculum; an ideal reader would have experience with languages like Java, C++, or Python, and be ready to learn a similar language. We'll be skipping most of the basics of programming – "what are variables?", "what are functions?" – in favor of a faster tour through C# that focuses more on applications and patterns thanks to the prior knowledge of the reader.

Conventions

The text of this book is formatted in different styles and layouts to help disseminate the many lessons and side topics within.

- Most of the book is written in prose, using this font.
- Code segments are written in typewriter font, as in `int x = 10`. Short expressions and statements will appear inline within paragraphs, but longer examples will be emphasized with full syntax highlighting:

```
public static void Main(string[] args) {
  Console.WriteLine("Hello, world!");
}
```

- **Bold** font typically denotes the introduction of new terms, or to highlight a summary statement, particularly within a numbered or bulleted list.
- *Italicized* font is used for general emphasis. It also highlights the names of menus within applications (*File*, *Tools*, etc.).
- Many types of highlighted boxes will set aside text that isn't strictly necessary to the content being discussed:

Discussion

Discussions involve lengthier and informal explanations of a recent topic, often involving history, theory, or other background knowledge that help highlight a lesson.

Info

Info blocks will typically contain formal definitions and documentation for language features or program design decisions.

Warning

Warnings will highlight common mistakes or potential pitfalls that are easy to overlook.

Tip

Tips include miscellaneous thoughts and advice from the writer's personal experiences.

Avocado Facts

I like avocados. I give them to students who get high scores on exams, and use them as silly examples in exercises. I'd like to share some avocado facts during this book. You can ignore these blocks if you don't care for my silly distractions.

Code style

I try to use a consistent coding style when writing C# code. Most of the details are irrelevant, but a few are noteworthy:

- I name the member variables of a class starting with a lowercase "m", followed by PascalCasing for the rest of the variable name. For example, if a class has a member variable for "title text", I would name it `mTitleText`. The `m` emphasizes that the variable is a member of the class, not a temporary value. All other variables are named using camelCasing.
- I indent with tabs and align with spaces. Normally I set tab width to 3 spaces, but the book's formatting works better with shorter indents, so we'll use 2 spaces for tabs in the text. (The beauty of tabs, of course, being that you can set your editor to whatever tab size you prefer and then we can stop arguing about it.)

- I use implicit typing with the `var` keyword whenever the type of the variable I'm declaring is unambiguous from context and the variable's name.

Source Code

Most of the book's source code is available for free on GitHub, at `https://github.com/nealterrell/checkmate-source`. The code is organized by book Part, and has projects you can compile and run using Visual Studio 2017. I encourage you to fork my repository as you work through the book, making your own changes and implementations as you feel the inspiration.

About the Author

Hi, I'm Neal! I am a lecturer in the Computer Engineering and Computer Science department at California State University Long Beach. What's a lecturer? We are faculty members who only teach, as opposed to *professors* who do research, publish, and administer a department. All the time that professors spend advancing the state of the art of computer science, I get to spend learning and practicing to be a better teacher.

I teach many different courses: theory and mathematics classes like discrete math, algorithms, and data structures; programming-focused classes like introductory Python and intermediate Java and C++; and senior electives in programming language theory and design, search engine technology and information theory, and .NET development. My students know me (I think) as a challenging but fair instructor who demands a lot but provides the resources necessary for success to those who seek them.

I have been programming with C# since 2002, and it remains my favorite programming language. I am also proficient in C++0x, Java, Python, JavaScript, Clojure, F#, and know enough Erlang and Prolog to get in trouble.

This is the first book I've ever written, though I've published hundreds of pages of lecture notes and lab manuals for the courses I teach. I hope you enjoy it!

Part I

C# and the .NET Framework

Chapter 6:
LINQ

Language Integrated Query (LINQ[1]) is a .NET library that extends the .NET collection classes – `IEnumerable<T>` in particular – to incorporate lessons from functional programming (FP) languages regarding the filtering, transformation, and aggregation of collections of data. It consists of a set of extension classes that add new member methods to the collection classes, as well as a SQL-like sub-language for working with those methods.

6.1 Extension Methods

Ada is obsessed with palindromes. She writes a static C# method to test if a given string is a palindrome, ignoring capitalization:

```
public class StringUtil {
  public static bool IsPalindrome(string s) {
    s = s.ToLower();
    for (int i = 0, j = s.Length - 1; i < s.Length / 2; i++, j--) {
      if (s[i] != s[j]) {
        return false;
      }
    }
    return true;
  }
}
```

She checks her code by calling it with her own name and the name of her best friend as arguments:

```
bool palindrome1 = StringUtils.IsPalindrome("Ada");
bool palindrome2 = StringUtils.IsPalindrome("Isaac");
Console.WriteLine(palindrome1); // should be true
Console.WriteLine(palindrome2); // should be false
```

Ada proudly shows her work to Isaac, and while he is touched by her concern for the palindromicity of his own name, he notes that Ada's `IsPalindrome` method could really be part of the `String` class itself. "Don't be silly, Isaac," Ada replies, "I don't own the `String` class and it would be illegal for me to try and add something to it." Ada's programming instructors collectively smile at her reverence for encapsulation, but Isaac is unimpressed. "A public method operating on a class'

[1]Pronounced "link".

public interface does not break encapsulation," he replies. "Any reasonable type system should allow you to amend a type with such a method."

Indeed, C# agrees with Isaac, and provides a mechanism called **extension methods** for "adding" new methods to existing types, without having to recompile the original type or otherwise access its source code. An extension method is a static method that can be called as if it were a member method of a different class. Ada must make two changes to her method: it must be contained inside a `static` class, and its string parameter must be annotated as `this`:

```
namespace AdaExtensions {
  public static class StringExtensions {
    public static bool IsPalindrome(this string s) {
      // as before
    }
  }
}
```

Note that she has not modified the `String` class directly. To use her method as a member of a string value, one must add a `using` directive referencing the namespace of the class with the extension method (in this example, `using AdaExtensions;`). The `IsPalindrome` method can then be called on any instance of the `String` class:

```
bool palindrome1 = "Ada".IsPalindrome();
```

Ada has, in effect, added a new public method to the `String` class without modifying or having access to the class itself! Her `StringExtensions` class is referred to as an **extension class**.

Discussion

There is no *semantic* difference in what Ada has done compared to the static method of her first attempt, so what's the point? First, we reduce the number of keystrokes to call the method; your money-making fingers will appreciate doing less work without sacrificing clarity. Second, there is more implicit *meaning* in a member method call – which clearly operates on and relates to a specific instance of a type – than in a static method call to an unfamiliar class. We will use extension methods when working with `FluentAssertions` in Chapter **??**, in addition to their presence in LINQ itself.

Before you run off to write extension methods for every type you've ever used, consider this strong limitation: because an extension method is not a member of the class it is extending, it *does not have access to the class' private members*. Any operation you wish to add to another type must be fully expressible using the type's *public* members only, as in Ada's `IsPalindrome` method. In general, if you can write it as a static method, you can probably write it as an extension method instead, as long as it would make sense for the method to be a member of the class you are extending.

Tip

Extension methods are *opt-in*, as they require using a particular namespace before seeing the methods as part of the extended type. Someone who doesn't want the extensions cluttering the type's API can choose not to use the namespace and simply call an extension like any other static method. To respect this option, it is traditional to separate extension classes into a distinct namespace containing only such classes; that namespace often has the word "Extensions" in it.

Info

Extensions can add methods to classes, but not properties or fields in any form. There are proposals[a] to add "Extension Everything" to a future version of C#, but for now we have to be satisfied with only using extension methods.

[a] https://github.com/dotnet/roslyn/issues/11159

6.2 Delegates and Getting Funky

Most languages provide a way to declare a variable that refers to a function. This may seem strange if it's your first encounter with the idea: variables are supposed to hold values, and how can a function be a value? But it's really not so strange. Variables really refer to objects, and objects are *things*. *Things* are defined by the values they can equal, and the operations they can perform, and so a function can be seen as a thing just as any other variable type you're already comfortable with:

- an `int` can equal integer values from -2^{31} to $2^{31} - 1$. Arithmetic operations can be performed on `int` values.
- a `String` can equal any sequence of 0 or more characters. There are many operations on `String` values: `Substring`, `Contains`, `StartsWith`, `Length`, etc.
- a function can equal a block of code that takes a certain set of parameters and returns a particular type of value. The only operation we can perform on a function is to invoke/execute it.

And now you're convinced! A variable can refer to a function just as easily as it can refer to a string, a file, a network connection, or a bank account.

How, then, do we declare variables as referring to functions, and how do we invoke a function referred to by a variable? Like all variables, we need to declare the *type* of a function in order to create a variable referring to it. In Ye Olden Days of .NET, **delegates** were used to give meaningful names to function types. Declaring a delegate creates a new type in a library, just like declaring a new `class` or `struct`; the delegate type always represents a type of function, by identifying the *types of the function's parameters* and the *type of the function's return value*. The following C# code creates a new delegate type:

```
public delegate int MyDelegate(int a, int b);
```

The components of the declaration:

- Access modifier (`public`): like any type, sets the accessibility of the delegate.
- `delegate`: identifies this declaration as that of a delegate type, and not an actual function.
- Return type (`int`): the type of value returned by functions that this delegate represents.
- Type name (`MyDelegate`): the name of the delegate type, like the name of a class. Variables will be declared using this type name.
- Parameter list (`(int a, int b)`): a list of types (important) and names (less important) of the parameters that are accepted by any function this delegate represents.

The `MyDelegate` type above allows us to create a variable that can refer to any function that takes two `int`s as parameters (even if they aren't named `a` and `b` – the names are irrelevant) and returns an `int` value. The `Math.Max` static method is a function that matches this description; to create a variable that refers to that function, we declare it using the `MyDelegate` type, as we would any variable:

```
MyDelegate f = Math.Max;
```

We can reassign `f` to any other function matching the `MyDelegate` requirement, like `Math.Min`:

```
f = Math.Min;
```

Finally, to invoke the function that `f` currently refers to, we use normal function-call syntax:

```
f(5, 3); // returns 3, since f is Math.Min
```

Discussion

If you're a C or C++ programmer, you can think of delegates as a named type-safe function pointer. Our code is equivalent to the C-style function pointer

```
int (*f)(int,int) = min;
f(5, 3);
```

and the modern C++ `std::function` type

```
typedef std::function<int(int, int)> MyDelegate;
MyDelegate f = std::min;
f(5, 3);
```

I introduced delegates as "old school" because they represent a low-level capability of the CLI that was exposed through C# using a clunky system, and you rarely see modern C# projects using them. Having to declare and name delegate types any time you wanted to use function pointers quickly grew tiresome. Fortunately, C# 3.0 introduced a pair of types to represent any specific type of function: `Func` and `Action`.

Func<..., TResult>

The Func<..., TResult> delegate type is modern C#'s way of representing function pointers. A generic type, Func lets us declare references to functions as long as we know their return type and the types of their parameters. If we declare a Func with only one generic parameter type, then the function represented by the Func variable takes no parameters, and returns a value of the generic parameter type. If we supply multiple generic parameters, then the *last* is the return type of the function, and the rest, in order, give the types of the function's parameters. A few examples will help:

```
// A function with no parameters, returning string.
Func<string> f1 = Console.ReadLine;
// A function taking two int parameters and returning int.
Func<int, int, int> f2 = Math.Min;
// A function taking a string parameter and returning int.
Func<string, int> f3 = int.Parse;
```

Each of the Func variables above can be invoked with the normal function call syntax, as long as we provide the necessary parameters. One fun part of delegates and Func is that the compiler checks everything for us at compile time: if I declare

```
Func<int, int, int> f4 = Console.ReadLine
```

then my compiler will issue an error that the delegate type Func<int, int, int> is incompatible with the function Console.ReadLine. Likewise, if I take the variable f2 and attempt to invoke it with two strings

```
f2("Abra", "Kadabra")
```

the compiler will emit an error that the delegate type Func<int, int, int> cannot accept two string parameters.

Action<T, ...>

What if we want a delegate to a function that has a void return type? We unfortunately can't put void into a generic type parameter, because void is a special type that can *only* be used as a function's return type. Enter the Action<T> delegate, which is identical in use and behavior to Func, except that all Action<T, ...> delegates have a void return type. Thus, the generic parameters to Action<T> get to skip to the chase, specifying only the types of the function's parameters.

And if we want a delegate to a void function with no parameters? Then the non-generic base Action type, with no generic parameters, gets its moment of fame.

```
// A function taking a string parameter with no return value.
Action<string> f5 = Console.WriteLine;
```

```
// A function with no parameters and no return value.
Action f6 = SomeBoringFunction;
```

Avocado Facts

Avocado trees suffer from their fair share of pests and diseases, causing no end of heartbreak to commercial and home growers alike. Persea mites build silver webs on the undersides of avocado leaves and drink sap from their veins, leading to necrosis along the vein lines and the eventual death of the leaf. Mites won't kill the tree directly, but leaf loss affect the tree's overall health in other ways, like leading to sunburn. (Did you know trees can get sunburns? Young citrus and avocado trees are usually painted with a diluted white latex paint to protect them until they grow a sufficient canopy.)

A far bigger concern for all avocado enthusiasts is the mold genus *Phytophthora*, derived from the Greek words for "plant destruction". *Phytophthora cinnamomi* in particular is an enormously destructive invasive mold species that affects dozen of commercial crops worldwide, including avocado and cinnamon trees. Overly-wet soils aid the spread of Phytophthora, which in turn causes "root rot" in avocado trees, affecting the tree's ability to absorb water through its roots and leading to the death of leaves, branches, and eventually the entire tree. There is no cure, though horticultural best practices can limit the spread and damage once Phytophthora is found in a region's soil.

Delegates to non-static member methods

Recall the `Substring` method of the `string` class, which takes two integer parameters and returns a `string`. A delegate to such a function would be of type `Func<int, int, string>`, but how do we actually declare a delegate to an instance (non-static) method of a class? `Func<int, int, string> f = string.Substring` doesn't work, because it implies that `Substring` is a static method, which it is not. `Func<int, int, string> f = Substring` also doesn't work, because we haven't identified which class the method is a member of. Unlike static methods, instance methods require an actual object instance in order to invoke. To declare our delegate, we have to identify *which* string object we want to use the `Substring` of, like this:

```
string val = "Phytophthora cinnamomi";
Func<int, int, string> f = val.Substring;
f(4, 3); // returns "oph".
```

Info

There's a good technical explanation for why we need to indicate the instance we are invoking a member method on when capturing it as a delegate, besides the intuitive "how else would we know what string to use in the `Substring` method?" The explanation: `Substring` *isn't really a method taking two integers and returning a string.* Surprised? Perhaps not; we've already stated this fact in another fashion: in order to execute, `Substring` needs to know which `string` instance it is operating on.

"Data that needs to be known for a function to execute" sounds an awful lot like the description of a "parameter", and indeed, in the example above, the string `val` is actually secretly passed as a parameter to the `Substring` method when it is invoked. *Every* instance method, it turns out, is actually compiled to a form that takes an additional parameter besides those explicitly listed by the programmer. This secret doesn't have an academic name – I like to call it the "context parameter", as it establishes the context on which the method is being invoked, but I've also heard "instance parameter" and like that too – but it does have a real name in C#: `this`, a humble keyword that usually gets taught as being optional in method bodies (not to be confused with the first parameter of an extension method), but in reality refers to a very real parameter to each instance method. The CLI is responsible for passing the variable `val` as the "`this`" parameter to `Substring` in our example, and the body of `Substring` gets to use that parameter either explicitly (using `this.xx` to refer to fields of the parameter) or implicitly by directly referencing fields of the `String` class.

With this knowledge, we would more accurately describe the `Substring` method as being of the type `Func<string, int, int, string>`... in other words, it requires a `string` context parameter, two `int` formal parameters, and returns a `string`.

Python programmers are more aware of this fact than those with a C++ or Java background. In Python, the context parameter is explicit: every instance method of a class (including constructors!) must manually declare a parameter for the instance object, which by tradition is called `self`.

6.3 Lambda Expressions

All the delegates we've created so far have referred to actual functions from .NET (or one we can imagine existing, like `SomeBoringFunction`). What if we want a delegate to refer to a function that doesn't exist, say a function to tell if an integer is positive or not? Then we'd have to write the function first:

```
public static bool IsPositive(int x) {
  return x > 0;
}
```

and then declare a `Func` delegate as

```
Func<int, bool> test = IsPositive;
```

But that's a lot of work for a function that is so trivially easy and short as `IsPositive`. In the near future we will want the ability to create small, simple functions without going through all the syntax of a full C# method declaration. C# gives us that ability in the form of **lambda expressions**.

A lambda expression declares an **anonymous function**: a function with no name, belonging to no class. Anonymous functions are just as capable as any "normal" method, and can take parameters, declare variables, call other functions, return values, etc. They just don't get a name

of their own. What use is a function with no name, and how can we possibly call such a function? By assigning it to a delegate variable.

We use the => operator to declare a lambda expression. To the left of the =>, we place a parenthesized list of parameters to our anonymous function. We have two options on what to place to the right of the operator:

1. If we write a single expression (ex: `x > 0`, `y * 2`, `Math.Min(a, b) - 4`) then the body of the anonymous function consists of a `return` followed by the written expression.
2. If we place an open curly brace, we can write the body of the anonymous function as we would any other method, including manually `return`ing a value.

We can rewrite `IsPositive` as a lambda expression in these two styles as thus:

```
// A single expression for the function body.
Func<int, bool> test1 = (int x) => x > 0;
// A full curly-brace function body.
Func<int, bool> test2 = (int x) => {
  return x > 0;
};
```

Both `test1` and `test2` can be invoked as with any other delegate.

If our lambda expression only takes a single parameter, the parentheses around the parameter list can be removed. If the types of the parameters can be inferred from context, then we can omit those as well. Our third way of declaring `IsPositive` as a lambda expression is the shortest by using these two tricks:

```
// Inferring the type of x.
Func<int, bool> test3 = x => x > 0;
```

The compiler knows `test3` to be of type `Func<int, bool>`; it therefore *knows beyond all doubt* that the function `test3` refers to must take a single parameter of type `int`. Seeing the lambda expression has only one parameter, the compiler infers the type of `x` to be `int`, then type-checks the function body and finds it to be returning `bool`, which also matches `test3`'s declared type.

Why bother with this seemingly obscure feature? Because it forms a foundation for functional programming and LINQ, as both benefit from the brevity of lambda expressions for anonymous functions.

6.4 Functional Programming

The functional programming paradigm far exceeds the scope of this book. Thousands of texts, articles, and blog posts espousing the benefits of *immutability*, *first-class functions*, and *higher order functions* await the dedicated learner, and I encourage all my students to learn at least the basic principles of FP to apply to all their programming tasks. LINQ itself draws inspiration from three common FP functions called *map*, *filter*, and *reduce*, and knowing the origins of these functions can help better understand LINQ.

Filter

Ada wants to write a method to take a parameter list of integers and return a list containing all of the positive integers from the argument. She codes this simple method:

```
public static List<int> PositiveInts(List<int> values) {
  var results = new List<int>();
  foreach (int i in values) {
    if (i > 0) {
      results.Add(i);
    }
  }
  return results;
}
```

Tip

Note that Ada's method does not *modify* the parameter, instead returning a *new list* containing the results. The default expectation of a programmer who passes a collection to a method is that the collection will not modified by the method.

Suppose she now wants a method that returns all *negative* integers from a list. Imagine Ada writing an almost-identical method `NegativeInts` whose sole change is a different condition:

```
if (i < 0) {
```

Now imagine `EvenInts` – returning the even integers – and `OddInts`, `MultiplesOf5Ints`, etc., an unending stream of methods that all encode the same abstract pattern: given a list of values, return a new list of all values from the input that satisfy some arbitrary condition. Isaac has come across this pattern before, and tells Ada that functional programming languages name it **filter**. Isaac and Ada agree that both the type of the data in the list (can't we do something similar with strings, or floating-point numbers, or any other type?) and the exact rule for selecting the satisfying data (the **predicate**[2]) are unimportant to the general flow of the function.

Ada writes her own version of filter in C# using the `Func<T, TResult>` delegate we learned earlier:

```
public static List<T> Filter<T>(List<T> values, Func<T, bool> pred) {
  var results = new List<T>();
  foreach (T i in values) {
    if (pred(i)) {
      results.Add(i);
    }
  }
  return results;
}
```

[2] *predicate*: A function that returns either true or false.

and then calls it with an anonymous function to recreate the logic of `PositiveInts`:

```
List<int> myVals = ...;
List<int> positives = Filter(myVals, i => i > 0);
```

As we saw earlier, the lambda expression `i => i > 0` returns true if a value `i` is positive; with this test as the function `pred`, her `Filter` function returns a new list containing only the integer values in `myVals` that are positive. Voila!

Info

This is our first time writing a generic method, which we briefly learned about in Chapter **??**. The important bits of the `Filter` declaration start with the `<T>` that follows `Filter`, indicating that the method is a generic method. The `<T>` establishes the name of the single generic type as `T`; when the function is called later, `T` will be replaced by the actual type that C# infers based on the place that `Filter` is used. We can use `T` as the type of a variable within the parameter list, return type, and body of our generic method, and we do all three of those: establishing that the parameters are a `List<T>` (this is how the compiler will infer `T`: based on the type of list passed in) and a function taking a `T` and returning a true or false; giving the return type a list of `T`, exactly matching the parameter's type; and declaring a `new List<T>` in the body. We even doubly establish that each element of `values` is type `T` in the `foreach` loop.

When we call `Filter(myVals, i => i > 0)`, the C# compiler will infer `T` to be `int` based on the type of `myVals`. It will then verify that the inferred generic type is valid for the second parameter, the lambda expression `i => i > 0`. According to `Filter`'s parameter list, the `Func` passed to it must accept a `T` parameter, which means `i` must be an `int`. If that is true, then the expression `i > 0` is semantically valid, and returns type `bool`. Since this matches the type of `pred`, this call to `Filter` is allowed, and returns a `List<int>`.

Map

A second important function from FP is **map**. Suppose that Ada wants to take a list of integers and return a list of the squares of each of the integers in the argument list. In her own reliable way, she writes:

```
public static List<int> Squares(List<int> values) {
  var results = new List<int>();
  foreach (int i in values) {
    results.Add(i * i);
  }
  return results;
}
```

We repeat our exercise from before and imagine other methods like `Doubles` (`results.Add(i * 2)`), `Triples` (`results.Add(i * 3)`), even methods that result in non-integer values like `SquareRoots`

(`results = new List<double>(); ... results.Add(Math.Sqrt(i))`). In all cases Isaac and Ada identify three abstract components of our method: the *list of values* to iterate through; a *function* to convert each of the values into another value; and the *type* that the values are converted into. Ada can then write `Map` as:

```
public static List<TResult> Map<T, TResult>(List<T> values, Func<T, TResult>
    convert) {
  var results = new List<TResult>();
  foreach (T i in values) {
    results.Add(convert(i));
  }
  return results;
}
```

and, as before, recreate her method `Squares` using an anonymous function:

```
List<int> myVals = ...;
List<int> squares = Map(myVals, i => i * i);
```

Discussion

`Map` has two generic parameters, which allows us to use different types for the list of values passed in (type `T`) and the type that the given values get converted into (type `TResult`). As before, the compiler is able to infer the actual types for the generic parameters based on the context in which the function is called. Above, `T` is inferred to be `int` based on `myVals`, and `TResult` is also inferred as `int` because the lambda expression `i => i * i` takes a parameter of type `T` and invokes the `*` operator; when `T` is `int`, this results in another `int` value.

Since we use two generic parameters, we can invoke `Map` to return a type that does not match the input list's. Calling `Map(myVals, i => Math.Sqrt(i))` would establish `TResult` as `double`, the return type of the `Math.Sqrt` function. The return type from `Map` would then be `List<double>`.

Finally, Isaac notes that the output of either of these functions can be fed as input to the other, allowing Ada to, among other things, square all the positive integers in a list, ignoring the negatives:

```
List<int> myVals = ...;
List<int> results = Map(Filter(myVals, i => i > 0), i => i * i);
```

It's a little dense at first, but after practice Ada finds no trouble reading code like this. "Order of operations," she reminds herself; "first we filter to only contain `i`'s that are positive, then we square all those `i`'s, and end up with a list of integers."

6.5 Origins of LINQ

A few observations about Ada's code from the previous section will lead us to some core ideas of LINQ:

1. Do the Filter and Map functions only work if the parameters are `List` collections in particular? That is, when filtering a bunch of integers, is it *mandatory* that those integers be in an array list-like structure, or could we also filter other structures like binary trees or hash tables?
2. Must Filter and Map return their results specifically in a `List`? Could they be made more flexible by returning a different type? What type would that be?
3. Are there any types that cannot be filtered or mapped?

Ada knows the answers to these questions... do you?

1. "Of course not. All we do with the input list is iterate through it, one value at a time. It could be a linked list for all I care, as long as it is *enumerable*."
2. "Sometimes I will want the results in a particular structure, but usually I won't care, as long as the result is enumerable."
3. "None. All types have attributes that can be used in a filter predicate, or transformed into another type by a mapping function."

These answers form the starting observation for LINQ: **any enumerable sequence of values can be filtered, mapped, and manipulated in many other ways** to produce another sequence of values. The LINQ library, in the `System.Linq` namespace, adds a number of extension methods to the `IEnumerable<T>` interface. These methods implement filter, map, and many other methods for manipulating sequences.

6.6 Core LINQ Methods

`Where`

The first LINQ method we will examine is `Where`, which has the following signature:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> seq,
  Func<T, bool> pred)
```

You may recognize this signature as nearly identical to Ada's `Filter` function, and indeed, `Where` is LINQ's implementation of filter. When called on an `IEnumerable` of some type, `Where` will return another `IEnumerable` containing only the values that satisfy the given predicate. Isaac provides the first example filters an array of integers to only the positive values:

```
int[] myVals = {1, -5, 2, 4};
var filtered = myVals.Where(i => i < 0);
```

Warning

You must include a `using System.Linq;` directive to use LINQ methods.

What type is `filtered`? Since C# arrays implement `IEnumerable`, we can see that the `seq` parameter to `Where` is of type `IEnumerable<int>`, making the generic type `T` equivalent to `int`. Thus, the predicate `pred` is of type `Func<int, bool>` (a function from `int` to `bool`) and `filtered` is of type `IEnumerable<int>`.

What's interesting here is that we have no idea *how* `Where` actually works. We can be certain that it calls our `pred` method to decide which elements of `seq` to include in the returned `IEnumerable`, but what exact data structure is returned? Is it a list? Is it a plane? Is it Superman? We don't know, and we (probably) don't ultimately care: we can iterate through the result with an enumerator, and that's usually all we need.

Tip

One day you'll need to know how `Where` actually works, but today is not that day, young Padawan.

Select

Next we will examine the LINQ method `Select`:

```
public static IEnumerable<TResult> Select<T, TResult>(
  this IEnumerable<T> seq, Func<T, TResult> conv)
```

We again recognize this signature as being equivalent to Ada's `Map` function. When called on an `IEnumerable`, LINQ's `Select` will return another `IEnumerable` by mapping each value in the input to a new value using the mapping function `conv`. Isaac's example maps an array of `strings` to their lengths:

```
string[] myVals = {"Ada", "Isaac", "Neal", "Jaclyn"};
var mapped = myVals.Select(s => s.Length);
```

This time we must examine both parameters to `Select` to determine the type of `mapped`. `myVals` is clearly an `IEnumerable<string>`, setting the generic parameter `T` as `string`. But what about `TResult`? For that, examine the anonymous function used as the mapping function. `s` must be a `string` (since `T` is `string`), and the `Length` property of a `string` returns `int`, making `TResult` equate to `int`. Thus, `mapped` is `IEnumerable<int>`! We can then determine `mapped` to be a sequence of the integer values 3, 5, 4, and 6.

Aggregate

Sensing she's ready for her next challenge in functional thinking, Isaac asks Ada to program a method that finds the sum of all the elements of a sequence of integers. She complies:

```
public static int Sum(IEnumerable<int> seq) {
  int sum = 0;
  foreach (int i in seq) {
```

```
    sum = sum + i;
  }
  return sum;
}
```

Ada is then prompted to explain the behavior of her code when executed on the array `{1, 5, 4, 2, 3}`. "First, we add the current sum of 0 with the first element of the array, 1, to get a sum of 1. Then, we take the current sum of 1 and add it to the next element, 2. Then the sum of 3 is added to the next element, 3, and we repeat until the final sum of 15 is found."

Isaac draws something on the nearby chalkboard:

$$3 + (2 + (4 + (5 + (1 + 0)))) = 15$$

and Ada agrees that the math expression he wrote is equivalent to the work that her function performs. "What changes would you make," he continues, "to find the product of a sequence of integers?" Ada's answer is hopefully clear: start the calculation at 1 instead of 0, and use operator `*` instead of `+`. The chalkboard expression then changes to

$$3 * (2 * (4 * (5 * (1 * 1)))) = 120$$

and then one final challenge is issued: finding the *largest* integer in the sequence. Ada takes a moment before recognizing the solution: start the calculation at `int.MinValue`[3], and change the expression in the loop to `largest = Math.Max(largest, i)`. Sensing Isaac's next and last question, Ada adds a final expression to the chalkboard:

$$\max\left(3, \max\left(2, \max\left(4, \max\left(5, \max\left(1, -2^{31}\right)\right)\right)\right)\right) = 5$$

Do you sense a pattern in these functions? In each, we take some *initial value* and apply a two-argument function to that value and the first value of the sequence. We then apply the function again to the result of the previous function and the second value from the sequence; then again with the previous result and the third value from the sequence; etc., until we have a single value from the final function call. This pattern is a third famous function from functional programming. Lisp languages like Clojure call it **reduce**, while ML-derived languages like F# and Haskell call it **fold**. C++ calls the function **accumulate**. LINQ has the most clear name, in my opinion: **aggregate**.[4]

The LINQ method `Aggregate` has two different overloads, depending on what you want the *initial value* of the calculation to be. Each implementation repeatedly applies some given two-argument function to the elements of the sequence that `Aggregate` is called on, feeding the result of one iteration of the function to the next.

1. `public static T Aggregate<T>(Func<T, T, T> aggFunc)`: given a function of two arguments of type `T` that returns a value of type `T`, this method feeds the first **two** elements of the sequence to the function `aggFunc`, which must "combine" those values in some way

[3] The smallest (most negative) value an `Int32` CLI type can store, equal to -2^{31}.

[4] *aggregate*: to bring together; to collect into one sum, mass, or body.

into another value of type `T`. That result is then passed to the next iteration of the function along with the next value from the sequence, repeating until a single final value of type `T` is computed and returned.

2. `public static TResult Aggregate<T, TResult>(TResult initialValue, Func<TResult, T, TResult> aggFunc)`: this overload is more akin to the pattern we noticed above. We supply an initial value manually, of type `TResult`. The function we provide must take a parameter of type `TResult` along with one of type `T` (the type of values in the sequence), and produce a new value of type `TResult`. The final value produced by the aggregation is also of type `TResult`.

We can implement Ada's `Sum` method, along with `Product` and `LargestOf` using `Aggregate`:

- `array.Aggregate(0, (a, b) => a + b)` is a literal translation of Ada's `Sum` method. We give the explicit initial value of `0`, and a function that adds two integers together. This example uses the second of the two `Aggregate` options, in which the initial value is explicitly passed. `TResult` is `int` (based on the argument `0`), making `aggFunc` a `Func<int, int, int>`. Our lambda function adds two variables inferred to be integers, which definitely results in an `int`.

  To see why this computes the sum of the sequence `{1, 5, 4, 2, 3}`, we feed `0` as `a` and the first element of the array (`1`) to our lambda function, which results in a `1`. That value is then fed back as `a`, with a `b` of `5`, to get `6`. This repeats to find intermediate values of `10`, `12`, and finally `15`, which is returned.

- We can use the first `Aggregate` option to shorten this function call: `array.Aggregate((a, b) => a + b)` uses the first two elements of the sequence as the first `a` and `b` values respectively, then continues with the repetition as above.

- `Product` is a simple translation from our `Sum` equivalent: `array.Aggregate((a, b) => a * b)`.

- `LargestOf` is actually the shortest: `array.Aggregate(Math.Max)`. `Math.Max` takes two arguments of type `int` and returns a type `int`. `1` and `5` are passed to `Math.Max`, which returns `5`; `5` and `4` are passed to `Math.Max`, which returns `5`; etc., until `5` is returned as the answer.

`Aggregate` is the least important of the three core LINQ methods, which is great because it's also the hardest to know how to apply correctly. We'll have perhaps one or two opportunities to employ the function during our board game implementations in Part II.

6.7 Other Fun Operations

Counts and emptiness

Want to know how many elements are in an `IEnumerable`, or if that enumeration is empty? Look no further!

1. The `Count()` method will return an integer count of the number of elements in the enumeration. **Be careful with this method**: although `Count()` will "cheat" and call the `.Count` property (which is typically a constant-time access of a member field) if a given `IEnumerable` is also an `ICollection`, not all enumerations can take advantage of this optimization. If what you're enumerating is not actually a collection, then calling `Count()` will force the .NET virtual machine to "walk" through the entire sequence counting as it goes along. Calling `Count()` twice in a row is doubly disastrous, as it walks the enumeration twice!

   Moral of the story: if you *really* need to know the exact length of a sequence, consider converting it to a list (see the next section), or at the very least ensure you only call `Count()` once for any sequence.

Discussion

The "walk through the enumeration" implementation of `Count()` can be visualized as an aggregation! `sequence.Aggregate(0, (a, b) => a + 1)` will add `1` to an initial value of `0` for each element `b` in the sequence (the elements themselves are more or less ignored), returning the count of the sequence.

Don't actually do this, of course. Just call `Count()`.

2. The `Any()` method will return true if and only if the sequence has at least one element, and false if it is empty. An overload of `Any` accepts a predicate function and returns true if at least one element of the sequence satisfies the predicate; it is equivalent to calling `Where(...).Any()`.

   Although `Any()` is conceptually equivalent to `Count() > 0`, we know from the discussion of `Count()` that we would be foolish to use it as a means of checking for (non-)emptiness.

Conversion methods

The `IEnumerable<T>` interface is a little lacking in functionality, and if you want to do anything more complicated than iterating through an entire sequence returned by a LINQ method, you'll probably want to convert that sequence into a concrete data structure like `List` or `Dictionary`. You might be tempted to call a constructor: `List<T>`, for example, has a constructor taking an `IEnumerable<T>` and initializing the elements of the list using those from the sequence. Avoid this temptation! LINQ provides methods to convert an `IEnumerable<T>` to a `List` or a `Dictionary` that use fancy tricks to improve efficiency in certain cases. Those methods are:

1. `ToList()`: constructs a `List<T>` large enough to fit the elements of the enumerable sequence, then adds the elements of the sequence to that list, such that the first element of the sequence is at list index `0`. Reference types are copied to the list as a reference to the same object that is in the sequence (we call this a **shallow copy**); value types are copied by value, placing a duplicate of the original element in the list (a **deep copy**).

Once we have a `List<T>`, we can do useful things like find the index of a particular data element, retrieve items based on index, or insert new items into the sequence, all of which are either impossible or inefficient on `IEnumerables`.

2. `ToDictionary()`: converting a sequence to a dictionary isn't quite so straightforward. We probably imagine that the *values* of the dictionary will be the same values as in the sequence, but what keys will be associated with those values? We must supply `ToDictionary()` with a function called the *key selector*, a function of type `Func<T, TKey>` that transforms an element of the sequence to the key you want associated with that value in the final dictionary.

   For example, suppose we have an `IEnumerable<Hero>` using the `Hero` class from Chapter **??**. We can convert that sequence into a dictionary that maps from a `Hero`'s `Name` to the `Hero` object itself by calling `heroes.ToDictionary(h => h.Name)`; the lambda function is the key selector, transforming a `Hero` into a `string`. The resulting value is of type `Dictionary<string, Hero>`.

Extracting single elements

`Where` returns a sequence of values matching a predicate, but what if you only want one value back? You could take the resulting `IEnumerable`, call `GetEnumerator`, force the enumerator to find the first element with `MoveNext()`, and then read that element with the `Current` property... but that's an obnoxious amount of work. You could also call `.ToList()` and then use the indexer property to get the element at position 0, but that requires copying every element from the sequence to a new list, and that's an obnoxious amount of work for the virtual machine. Instead, two methods are provided to unwrap a single element from a sequence:

1. `First()`: returns the first element of the sequence. This method *throws an exception* if the sequence is empty, which you should catch and deal with if this outcome is at all possible. You can avoid an exception by calling `FirstOrDefault()` instead, which will return `null` if given an empty sequence of *reference types*, or a default-constructed instance if the sequence is of value types. Since default-constructed value types can be difficult to differentiate from "real" values, I recommend only using `FirstOrDefault` on sequences of reference types.

   Example: given a sequence of `Hero` objects, we can retrieve the first hero in the sequence that has at least 20 hit points by calling `heros.Where(h => h.HitPoints >= 20)` `.FirstOrDefault()`.

2. `Single()`: does the same thing as `First()`, including a `SingleOrDefault()` option, however these methods also throw an exception if the sequence has *more than one element*. You should use `Single` if your program would be in error if a prior LINQ call resulted in more than one value, such as finding a particular name among a sequence of heroes whose names are *supposed* to be unique, or finding the one square on a chess board that contains black's king piece. Either of these would be in error if the `Where` clause identifying the desired values returned more than one value, and `Single` enforces that condition better than

`First`.

Both `Single` and `First` have an overload taking a predicate function (`Func<T, bool>`) that returns the first element of the sequence that satisfies the predicate. This single method call can replace any situation in which a `Where` is followed by a `First`/`Single`, such as the example `heroes.Where(h => h.HitPoints >= 20).FirstOrDefault()`. This option simplifies the calls to `heroes.FirstOrDefault(h => h.HitPoints >= 20)`, and can be directly read as "return the first hero with at least 20 hit points", rather than "filter heroes to only those with at least 20 hit points, then return the first of them".

Sorting sequences

The `OrderBy()` method allows us to sort an enumeration by selecting a key to represent each data element, and then sorting by the natural (ascending) ordering of that key. Key types must implement the `IComparable` interface. For example, we can use `OrderBy` to sort our sequence of `Hero` objects by their names: `heroes.OrderBy(h => h.Name)`. The resulting sequence will contain the same heroes as in the original sequence, but ordered in increasing alphabetical order by their `Name` property.

It's important to note that `OrderBy` does **not** sort the original sequence passed to it; a new sequence is produced in the order requested, using a stable[5] quicksort to perform the ordering.

Two small things to remember about ordering:

- `ThenBy()` can be used to sort a sequence on an additional key, after first sorting by the key specified in a preceding `OrderBy()`. For example, `heroes.OrderBy(h => h.Level).ThenBy(h => h.Name)` sorts a sequence of heroes in increasing order by level, and for any two or more heroes with the same level, those are sorted among themselves by name.
- `OrderByDescending()` can be used to sort a sequence in descending order, and its equivalent `ThenByDescending` can add a descending secondary sort key.

Selecting subsequences

We can combine the `Skip()` and `Take()` methods to extract subsequences from an existing `IEnumerable<T>`:

- `Skip(int n)`: given a sequence of values, skip the first `n` elements of the sequence, and return a new `IEnumerable<T>` consisting of the remaining elements from the original sequence.
- `Take(int n)`: given a sequence of values, return a new `IEnumerable<T>` that only contains up to the first `n` elements of the original sequence.

[5] If two keys are equal, then whichever key was before the other in the original sequence will still be before the other after the sort is complete.

Together, we can implement the rough equivalent of a "substring" operation on general `IEnumerable` sequences: `someSequence.Skip(4).Take(3)` is equivalent to calling `Substring(4, 3)` on a `string` value.

Discussion

In fact, the `String` class itself implements `IEnumerable<char>`, so any LINQ method can be called on `string` variables to access individual characters of the string. The LINQ query `"Phytophthora".Skip(3).Take(5)` returns `"topht"`, the same result as calling the `String` method `"Phytophthora".Substring(3, 5)`. As with our discussion about `Aggregate()` and `Count()`, there's no real reason to do this... it's just a trick to show off in front of your friends.

What if you don't know exactly how many elements to skip or take? `SkipWhile` and `TakeWhile` to the rescue!

- `SkipWhile(Func<T, bool> pred)`: given a predicate function, skip elements of the sequence as long as the elements satisfy the predicate. Returns a new `IEnumerable<T>` consisting of the first element that does not satisfy the predicate, *and all elements that follow.*
- `TakeWhile(Func<T, bool> pred)`: returns a new `IEnumerable<T>` that contains all elements *up to but not including* the first element in the sequence that does not satisfy the predicate.

6.8 Exercises

LINQ is a powerful tool, but it takes some experience to recognize which LINQ functions to use, and in which order, to solve a particular problem. The following exercises should help you recognize situations where LINQ can help you write shorter code with greater clarity. One solution is worked for you. In these exercises, suppose we have a class called `Film` representing a motion picture film, (partially) implemented like this:

```
class Film {
  public string Title {get;}
  public string ProductionCompany {get;}
  public double Budget {get;}
  public double Earnings {get;}
  public int LengthInMinutes {get;}
  public int YearOfRelease {get;}
}
```

and a variable `IEnumerable<Film> films` initialized to some sequence of `Film` objects. For each of the following questions, first decide the *type* of the value you are being asked to produce, then write the required C# expression(s).

1. Create a sequence of titles for all films that were produced by Pixar Animation Studios.

   Answer:
   The result should be `IEnumerable<string>`.
   `films.Where(f => f.ProductionCompany == "Pixar Animation Studios").Select(f => f.Title);`

2. Create a sequence of the 5 highest-earning films.
3. Find the average budget of films produced by Lucasfilm Ltd. (Multiple statements may be required.)
4. Determine the title of the film that has the highest profit-per-minute-of-length ratio. (Profit: the difference between earnings and budget.)
5. Reorder the films by the year of release in ascending order. Calculate the total earnings of only the films including and following the first film in the reordered sequence that is less than 120 minutes in length.

The `PartI` folder of the book's source code (`https://github.com/nealterrell/checkmate-source`) has a C# project called `LinqExercises`. In the `Program.cs` file of that project, you will find a variable `films` that you can use to test your answers, and compare them to the correct solutions given in the source code.

Part II

Implementing Board Games

Chapter 7:
Board Game Design

And so we begin our second goal of the book: the development of a sophisticated implementation of chess.

That goal is actually much more broad: as hinted in the Introduction, our ultimate plan is the development of an application that plays a wide range of arbitrary board games. To start towards our goal, we'll begin with a design exercise to help identify behaviors and traits of board games that need to be modeled in our software. We'll apply principles from object-oriented programming to design a flexible framework for representing board games, and then build an application that consumes our framework and allows a user to play any board game that plugs into the framework using a command-line interface. We'll introduce several software design techniques that will make our framework easy to implement and work with, and other scattered lessons about .NET, LINQ, and class design will strengthen our C# programming skills.

We won't design any specific games in this chapter, but the lessons and code developed now will aid us when we program Othello in Chapter **??** and chess in Chapter **??**.

Since this is the first chapter of Part II, it may help to review the Problem, Design, Solution section of the Introduction on page iv, which will remind you of the change in book format starting with this chapter.

7.1 Problem

We want to build a console application for playing any arbitrary two-person board game. We have in mind, in particular, the games Othello, chess, checkers, tic-tac-toe, connect four, and mancala, knowing from personal experience that these games all fit into a particular design. The user will select a game from a list of implemented options, and will then play the game by typing commands into the console. The commands will include applying a move, undoing a move, showing the history of moves applied to the game, and indicating who is winning the game.

We want our application to be agnostic to the exact game being played. Our main game loop, which will ask the user what command to perform and then apply that command, should not have any knowledge of specific game types. We do not want to see logic like "if the user chose to play Othello, then ask where they want to place a piece; else if the user is playing chess, ask them to select a piece and then an ending location; else..." That will allow us to develop new game types in the future and seamlessly plug them into our application.

The games themselves will need to be programmed. With an eye toward the future, we want our game logic to be independent of the *presentation* of the game; we will want to use the game logic code from our console application to drive a graphical user interface of the same game. Our game implementations should not assume that we are using a console for input, or that they are being printed as text output.

Also considering the future, we want our game framework to support an artificial-intelligence opponent. An AI will need to query certain information about game boards in order to make optimal play decisions, so we will be sure to make that information available through the interfaces we design.

7.2 Design

Application framework

How do we build an application to play board games that doesn't care what exact game is being played? By programming only to an *interface*. For now, we don't literally mean the C# `interface` keyword... rather, we mean the publicly-known capabilities of some "thing" we are modeling. The interface of a "list" indicates that it can add, remove, and retrieve items at specific indexes; the interface of an automobile says that we can turn the engine on/off, engage the engine, and engage the breaks. Our first job is to design the interfaces necessary for our board game application.

Let's start by designing an interface that identifies a set of operations that any board game can perform. These operations should allow a game-agnostic application to "drive" any game that implements the interface... as long as we can identify those operations.

Let's picture the least complex game we have in mind: tic-tac-toe. A simple tic-tac-toe application would probably look something like this:

1. Create a representation of a tic-tac-toe board.
2. Repeatedly:
   (a) Display the tic-tac-toe board to the user, and identify whose turn it is.
   (b) Ask the user to select one of the nine squares to place a piece on.
   (c) Verify that the selected square is empty, and ask again if it is not.
   (d) Place the current player's piece on the selected square, and update the current player.
   (e) Loop as long as the game is not over.

Now imagine playing Othello in a similar application... what changes?

1. Create a representation of an **Othello** board.
2. Repeatedly:
   (a) Display the **Othello** board to the user, and identify whose turn it is.

   (b) Ask the user to select one of the **sixty-four** squares to place a piece on.

   (c) Verify that the selected square is **a valid move for the current player**, and ask again if it is not.

   (d) Place the current player's piece on the selected square, **flip enemy pieces**, and update the current player.

   (e) Loop as long as the game is not over.

Not much has changed! Indeed, we can abstract this sequence of operations into a game-agnostic application, by noticing that every board game application we have in mind follows a general pattern:

1. Create a representation of the **game board**.
2. Repeatedly:
   (a) Display the **game board** to the user, and identify whose turn it is.

   (b) Ask the user to select one of the **possible moves for the current board**

   (c) Verify that the selected move is **one of the possible moves**, and ask again if it is not.

   (d) **Apply the selected move**, and **update the game's state**.

   (e) Loop as long as the game is not over.

Or, more succinctly:

1. Create the game model.
2. Repeatedly:
   (a) Display.

   (b) Input.

   (c) Validate.

   (d) Apply.

   (e) Loop.

All we have to do is design a framework for representing different board games that enable an application built in this way.

Board game models

With an idea of what our main application will look like, we can turn our focus to representing the state of a board game and operations on that state.

We want to separate the *model* of our games from their visual representations, and from the application code itself. To do so, we will implement the **model, view, controller** (MVC) software

architecture. In MVC, code is separated into different modules based on its concerns. Code for representing the state of the application – in this case, the current state of a game's board – goes into the *model* module. Classes in the model only talk to other classes in the model, and these classes do not concern themselves with how they will be used by others. The goal of a class in the model is to provide a representation of a single concrete idea, as self-contained as possible, with as few assumptions about the outside world as possible.

We will design an interface called `IGameBoard` (following the .NET tradition, in which interface names are prefixed with "`I`") to represent the state and rules of any arbitrary board game. Board games are so varied in implementation that it may seem impossible to create an interface to represent *any* game, but we've already done the hard work of identifying the operations necessary to be a game: the operations needed to program our main application. We will require that any game designed for our application implement the following operations:

- **Get all possible moves**: a game should be able to generate a sequence of all moves that the current player could choose for any current game board state, reflecting the rules of the game itself. The moves can be represented in a game-specific way (Othello identifies a move based on its row/column coordinates; connect four needs only a column to identify a move; chess perhaps needs a starting location and ending location).
- **Apply a move**: given an entry from the game's sequence of possible moves, a game should be able to update its own internal state to reflect the application of that move, in a game-specific way.
- **Undo a move**: with an eye towards an artificial intelligence opponent in the future, games should be able to undo the last move applied to the board, restoring the game to whatever state it was in prior to the application of that move.
- **Identify a history of applied moves**: a game should keep track of the order that moves were applied to it, to allow a user to know how a game played out from the beginning.
- **Identify the current player**: a game should be able to report whose turn it currently is.
- **Indicate when the game is done**: a game should know when it has completed due to the actions of the players, in a game-specific way.
- **Indicate the advantaged player**: a game should be able to indicate which player is currently "ahead" in the game, and report that advantage as a relative value in a game-specific way.

Any game we plan to implement will need a model class that implements `IGameBoard`. The class will declare any members needed to represent the specific game, in a way that is compatible with the `IGameBoard` operations.

Game moves

Game boards have to report possible moves and accept moves to be applied... but what is a "move"? We can identify four different ways in which the abstract idea of a move shows up in

our plans so far:

1. A user must input a move to apply during the main game loop.
2. The main loop must validate a move, making sure it is one of the board's possible moves.
3. A board must accept a move to apply, assuming it is valid.
4. A board must remember an applied move in its move history.

A fifth requirement will support our long-term plans for our main application:

5. A move in the move history should remember which player applied the move.

Those requirements specify many ways in which moves are *used*, but really only one *behavior*: moves must be able to compare themselves for equality, so that a controller can ensure that a selected move is equal to one of a board's possible moves. We also note only one *trait*: a move should know which player applied the move.

Views and output

The view class(es) for a particular game will be separated from its model logic. Views have to be designed to work with a particular output medium, and so this module will be designed with console input and output in mind. In general, *anything* to do with a game's representation in a console window, or with parsing textual representations of game components, belongs in a view class.

The view for a game is concerned with two primary tasks: printing a text representation of a game state to a given output stream, and parsing text entered by a user to represent a move for a game. Each game will implement its own view class that is capable of these two tasks, and the controller will use an instance of a view class to perform the *Display* and part of the *Input* steps from the abstract game loop earlier on page 25.

Controller

The controller portion of an MVC application "drives" the program by initializing the model, generating output with the view, and then gathering input from the user with which to update the model. Our controller module will consist of a single class with a `Main` method that acts as the application's entry point. The controller will implement the abstract "play a game" loop from the *Application framework* section on page 24.

Miscellaneous

We observe that *most* of the games we have in mind use square game boards with positions that can be identified using row/column coordinates. Rather than write dozens of methods that pass pairs of `int` parameters to identify positions, we'll encapsulate the idea into a single type and use

it when appropriate. We'll build the `BoardPosition` class as an immutable value type (`struct`), checking to make sure we follow the guidelines we developed on page **??** regarding value vs. reference types:

1. `BoardPosition` logically represents a single value, similar to primitive types. (It is a single position on a game board.)
2. Only 8 bytes of member state is necessary (two 4-byte integers), making stack allocation viable.
3. It will be immutable, with operations for creating new `BoardPositions` based on existing values.
4. It will not have to be boxed frequently. We cannot see a reason to ever box a `BoardPosition`.

Avocado Facts

"Grow an avocado tree at home!" shouts the Internet. With five simple steps, we are assured, we can grow our own avocado tree straight from the seeds we so callously throw away. Just cut a small slice from the bottom, stick a pair of toothpicks into the sides, and suspend over a cup of water until roots grow. Holy guacamole!

Go ahead and try for yourself. I'll check back with you in ten years. If you're lucky, your tree *might* be growing fruit by then. But you probably won't like them. Like all fruit trees, avocado trees are not *true to seed*: the plant that grows from a seed of a parent won't be the same as the parent. Trees from seed usually have poor quality fruits, or don't bear fruit at all! You'll have to put aside your dreams of swimming in Hass for now... there are explanations and solutions for this genetic behavior, but all in good time.

7.3 Implementation

Projects and assemblies

We'll start with a Visual Studio Blank Solution, which will give us greater control over our physical project organization than if we made a C# project directly. The name of a Solution does not affect any of the code that gets generated, but we will follow the naming schemes we outlined in the Design section. (See Figure 7.1.)

This will be a multi-project solution, and we'll start by organizing the projects into *src* (for source code) and *test* (for unit tests) Solution Folders. Solution Folders don't automatically map to file system directories, so we'll have to set the path of each project we create by hand, but will give us a little more organization within Visual Studio. (Figure 7.2.)

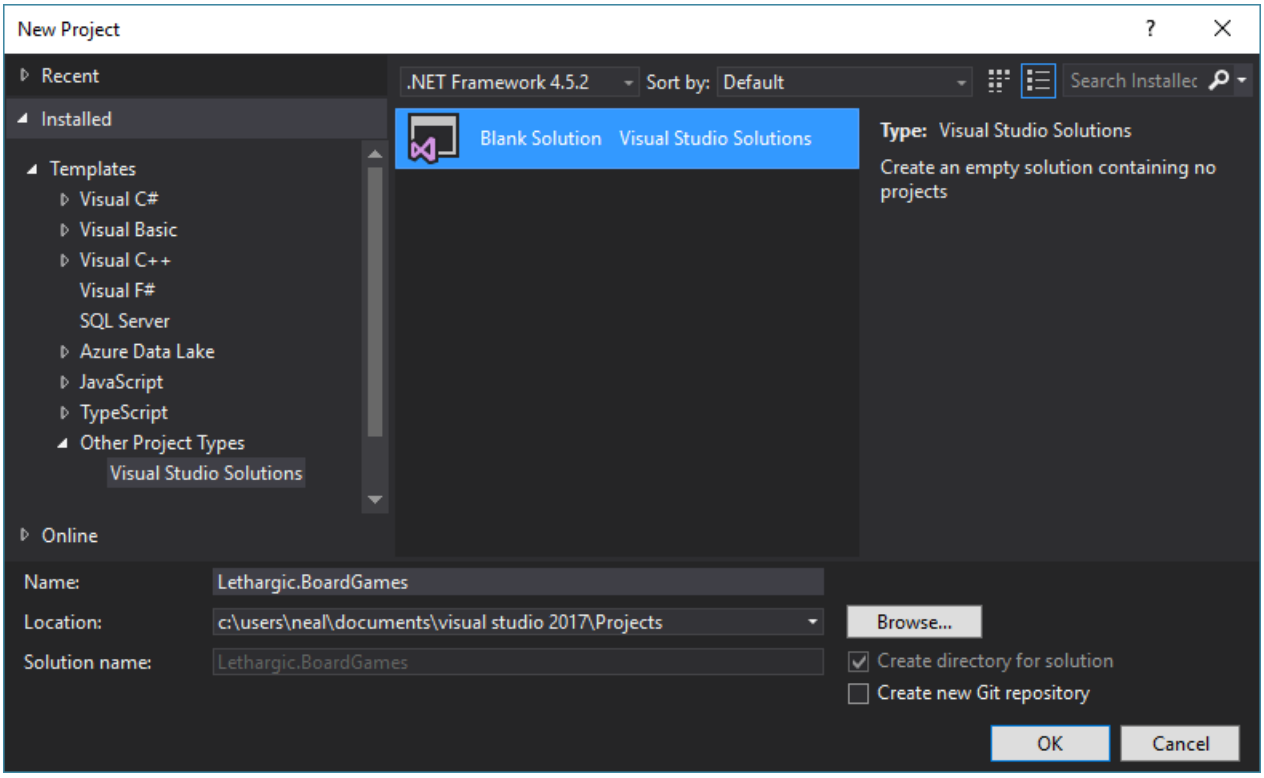

Figure 7.1: Creating a new Blank Solution in Visual Studio 2017.

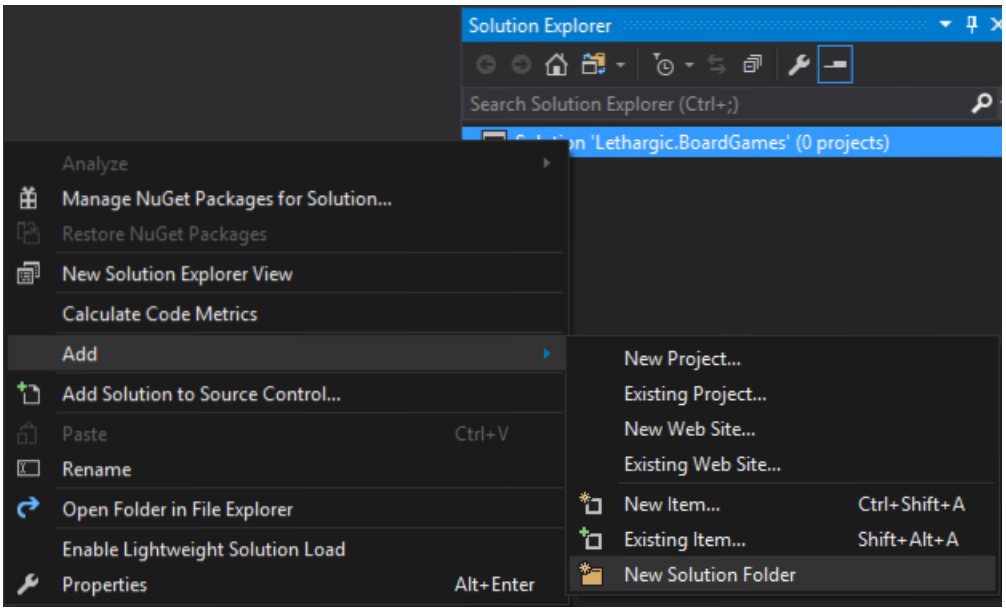

Figure 7.2: Adding Solution Folders.

Next we'll set up the projects we need for this chapter. Following our MVC design pattern, we'll need separate projects for our *model* classes, our *view* classes, and our *controller* classes. For each

we need to choose an output type and a target framework.

- **Output type**: in .NET, this refers to whether a project will be compiled as a *.dll* assembly with no entry point (main method), or as an application that can be executed. Generally speaking, Class Libraries should be selected for any project that is *not* the entry point for an application. We want our model and view projects to be Class Libraries, and our controller project to be a Console Application.

  "Class Library" used to be descriptive enough to get started, but in modern .NET development we need to be a little more specific. Just look at how many project types match the filter "C# class library" in Visual Studio (some may be missing depending on your installation choices). (See Figure 7.3.)

- **Target framework**: given the growth of the .NET ecosystem detailed in Part I, Visual Studio allows us to target specific flavors of .NET depending on our needs. We will target **.NET Standard 1.0** for our model and view projects, as we anticipate the code in those projects being platform-agnostic. (The game of chess does not change if you play on Windows vs. Linux.) Our controller project needs to target a specific CLI implementation; we will choose .NET Core, given its cross-platform nature.

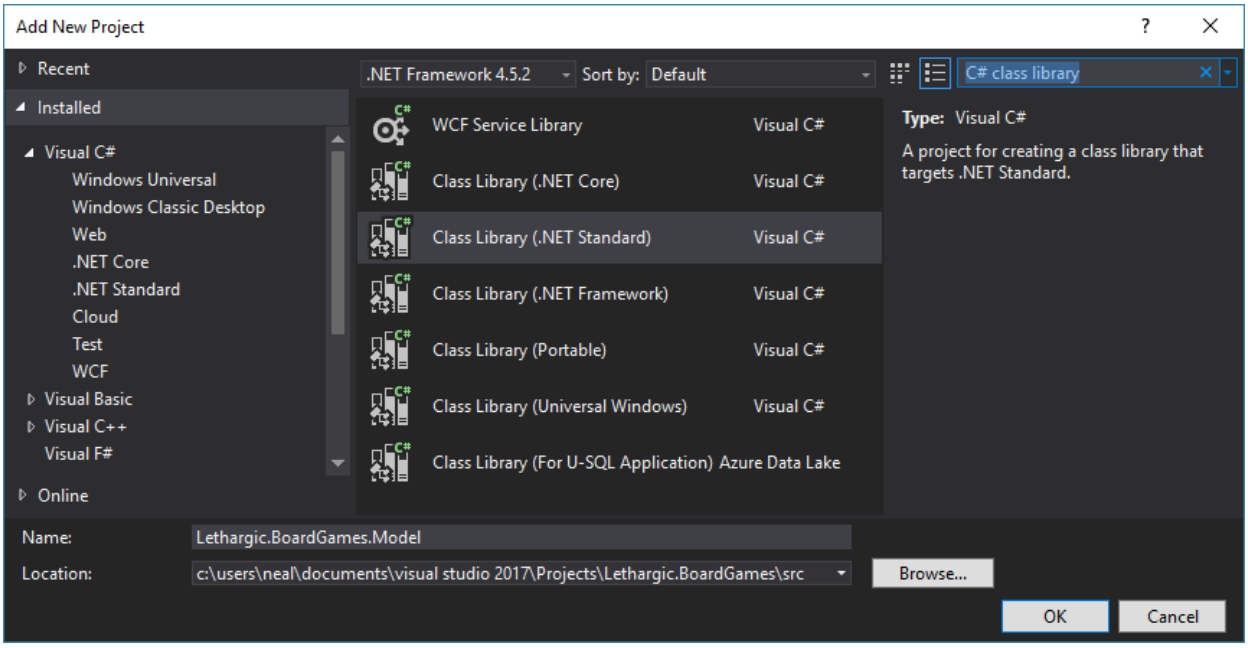

Figure 7.3: Adding Solution Folders.

Following the naming guidelines set earlier, we will create two **Class Library (.NET Standard)** projects named `Lethargic.BoardGames.Model` and `Lethargic.BoardGames.View`. After that, we add a **Console App (.NET Core)** project named `Lethargic.BoardGames.ConsoleApp`.

Warning

Be careful to add `\src` to the Location at the bottom of the Add New Project windows, and to create the projects in the first place by right-clicking the `src` Solution Folder in the Solution Explorer window. Both are needed for the projects to end up in the physical `\src` file system folder and the logical `src` Solution Folder.

By default, Visual Studio will initialize our .NET Standard projects to use the highest installed version. Newer versions of .NET Standard add more APIs (classes and methods) to the standard, giving developers a greater range of capabilities they can use when targeting .NET Standard... but we don't need those capabilities. Our code is simple enough that .NET Standard 1.0 will suffice, and so we will change it. Right-click on the two Class Library projects in the Solution Explorer and go to *Properties*. In the *Application* tab, change the *Target framework* to **.NETStandard 1.0**. (Figure 7.4.)

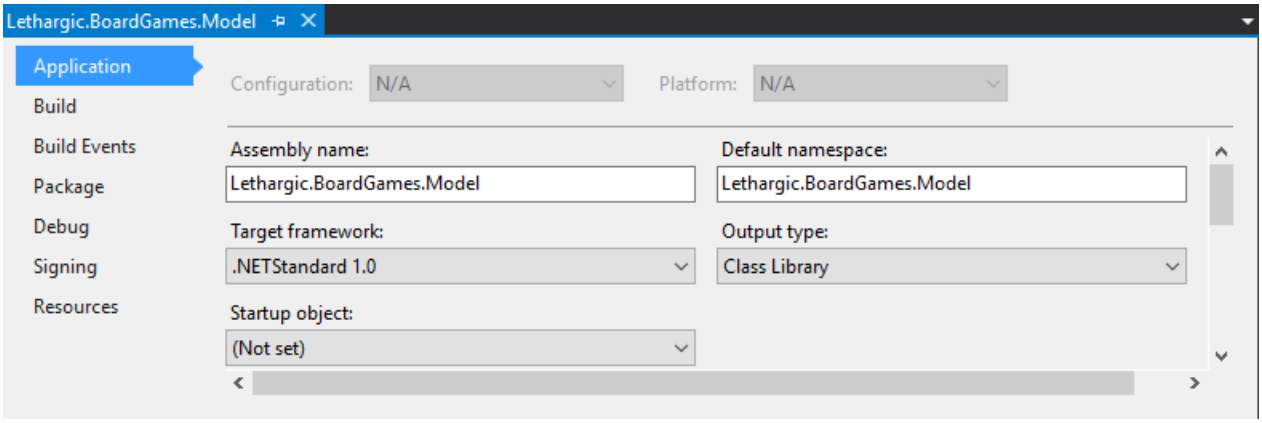

Figure 7.4: Changing to .NET Standard 1.0.

Tip

When choosing a version of .NET Standard, choose the lowest-numbered standard that contains all the APIs your code needs. That will maximize the number of systems your code is compatible with.

Finally, our projects need to *reference* each other so that code defined in the model can be used in the view, and both the view and model can be used in the controller. We right-click on *Dependencies* under our `Lethargic.BoardGames.View` project in the Solution Explorer, then select *Add Reference*, and check `Lethargic.BoardGames.Model` in the dialog that appears. We repeat this for the `Lethargic.BoardGames.ConsoleApp` project, adding both the `.Model` and `.View` projects as references.

At the end of the entire Part III, we will have a Solution layout like this:

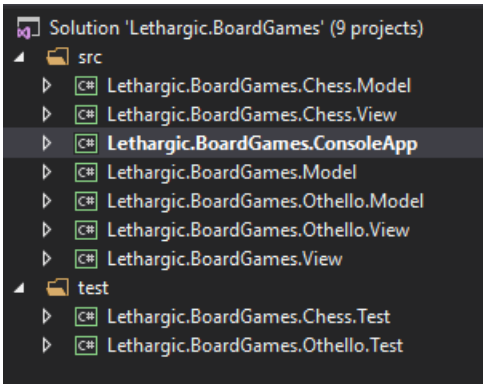

Figure 7.5: The Visual Studio projects you will have by the end of Part II.

For now, you will only have the three projects we just created.

Implementing `IGameMove`

Remember how we only identified one behavior and one trait for game move objects? That behavior (testing for equality) and trait (indicating a player) leads us to a very simple interface that all move types must implement:

```
interface IGameMove : IEquatable<IGameMove> {
  int Player { get; }
}
```

This interface will serve as the base type for any type that represents a move on a game board, with each game type that we implement creating its own `IGameMove`-derived type. It requires only a single method: `bool Equals(IGameMove other)`, for comparing two move objects for equality. Any board operation that manipulates "moves" will accept or return values that implement this interface.

The `Player` property will use the semantics defined for the `CurrentPlayer` property of `IGameBoard`, explained on page 34. This property will only be defined for moves that have been applied to a board, and will remember the number of the player who applied the move.

Implementing `IGameBoard`

In contrast to move objects, we noted many behaviors and traits that all game board models should implement (page 25). We will translate those to C# methods and properties of the `IGameBoard` class:

```
interface IGameBoard {
  IEnumerable<IGameMove> GetPossibleMoves();
  void ApplyMove(IGameMove m);
```

```
  void UndoLastMove();

  IReadOnlyList<IGameMove> MoveHistory { get; }
  int CurrentPlayer { get; }
  bool IsFinished { get; }
  GameAdvantage CurrentAdvantage { get; }
}
```

Any class that represents the state of a game's board will implement this interface. Note how we translated our English requirements for board types into this C# interface:

- **Get all possible moves**: a method returning a sequence of `IGameMove` objects representing the possible moves for the current board state.
- **Apply a move**: a method taking an `IGameMove`-derived value appropriate to the current game type.
- **Undo a move**: a `void` method requiring no other inputs.
- **Identify a history of applied moves**, **Identify the current player**, **Indicate when the game is done**, and **Indicate the advantaged player**: get-only properties for these board traits.

There are many notable decisions in this design:

Properties vs. methods

Any operation that could be expensive should be a method, not a property. `GetPossibleMoves` is the only method that *could* be a property; "a sequence of possible moves" could be seen as a semantic "property" of a game board. However, generating a list of possible moves could be an expensive operation for a game, and so we make it a method. The other properties feel like true properties, and we expect they will most likely be implemented using private members that get updated when the board state changes.

`GetPossibleMoves`' return type

Internally, we expect that most games will use a list to represent the game's possible moves, but that information is not important to the consumers of an `IGameBoard`. A sequence of possible moves will most likely be used to iterate through in its entirety: perhaps to print all possible moves to the console, or to enumerate the moves searching for one that matches a selection made by the user. We can't see a strong justification for knowing more about the collection used to represent the sequence, and so we choose `IEnumerable<IGameMove>` as the return type.

Discussion

Applying the lessons we learned in Chapter **??**, we note that this return type on a method indicates that the returned collection is a temporary value that is not tied to a game's chang-

ing state. We will allow a consumer to get possible moves for a game, then apply a move to the game, and *not* expect the previously-collected sequence of possible moves to magically update.

`ApplyMove` takes an `IGameMove`

Note that `ApplyMove` takes the base interface `IGameMove` as a parameter. In theory, this allows a misinformed or malicious user to pass *any* move type to *any* board type's `ApplyMove`. An Othello board can't be expected to make sense of a chess board's move, so this operation clearly makes no semantic sense and would be in violation of the interface. Still, good defensive programming will require us to validate that any `IGameMove` passed to an `IGameBoard` instance be verified to be of the correct expected move type.

In addition, we make it part of `ApplyMove`'s contract that the move passed as a parameter **must** be an object that was returned as part of a `GetPossibleMoves` enumeration.

Using `IReadOnlyList<T>`

We recognize that the history of moves applied to a board should be represented using a list, as the order that moves were applied to a board is an important fact that should be exposed through the interface. But we don't want to allow consumers of an `IGameBoard` to *add or remove* game moves to/from the history list, which would otherwise be allowed if we exposed the history as an `IList`. `IReadOnlyList<T>` allows retrieval of applied moves based on index, but not adding or removing moves.

Representing the current player

Although all the games we have in mind are two-player games, we don't want to enforce that as a requirement in the interface. If we did want to do this, we'd represent the current player as an enum with only two possible values. Without that option, we'll stick to an integer, and expect games to report players using the positive whole numbers, e.g., 1,2,.... Using a 4-byte `int` might seem like overkill, but it gives flexibility, and games can use a smaller type if they want, letting .NET coerce their smaller member to an `int` for the property's return value.

Representing the advantage

The `CurrentAdvantage` property has a return type of `GameAdvantage`, which is a type we have not seen. This type will simply bundle a player number with an integer representing their "advantage". Each game will define what an "advantage" is: in Othello, advantage will be the difference in the number of pieces each player controls, whereas tic-tac-toe might not define advantage at all. We decide to use a class here to allow for games that have more than 2 players; otherwise, an integer value might suffice, with positive indicating player 1's advantage and negative indicating player 2's.

Indicating the winner

If `IsFinished` is true, the game needs to also report which player won the game. That will be done through the `CurrentAdvantage` property, whose value will be interpreted as the winner of the game if and only if `IsFinished` is true; otherwise, it indicates who is "winning".

Implementing `IConsoleView`

The `IConsoleView` interface defines the capabilities of a class in the view module of our console application. `IConsoleView` objects are responsible for printing a corresponding `IGameBoard` to a text output stream, and for parsing a `string` representation of an `IGameMove` to be used with a game board.

```
interface IConsoleView {
  string BoardToString(IGameBoard board);
  string MoveToString(IGameMove move);
  IGameMove ParseMove(string moveText);
  string PlayerToString(int player);
}
```

`BoardToString`

`BoardToString` takes an `IGameBoard` as a parameter and returns a string representation of that board, suitable for printing to a text stream writer that is *not* assumed to be standard output. View classes can thus be used to "print" a board to a file, a network connection, or any other text output stream.

As noted in the discussion of `ApplyMove`'s parameter, in theory an `IConsoleView` could be asked to print a model from another game. We consider this a semantic error that implementations should guard against.

`MoveToString`

`MoveToString` takes a move object created by a model of the same game and returns a string to represent that move in textual output.

Why not move `BoardToString` and `MoveToString` to the built-in `ToString()` methods of model classes? First, to clearly separate concerns: models have no care for how they are represented, and including a `ToString` designed for console output would violate that principle. Second, this separation allows us to implement `ToString` in model classes for a different purpose: to aid in debugging, as many debuggers will use an object's `ToString` output to give an at-a-glance summary of an object's state.

ParseMove

`ParseMove` takes a string representation of a move and constructs a model object matching that representation. In general, calling `ParseMove` on a string created by `MoveToString` should produce an `IGameMove` that is equivalent to the original object sent to `MoveToString`.

PlayerToString

Finally, each board type uses different labels for its players: "black" and "white" fit for Othello and chess, but tic-tac-toe uses "X" and "O", connect four is usually "Blue" and "Red", etc. An `IConsoleView`'s last responsibility is producing a string representation for a given player value.

Main application

Although our Problem this chapter suggested a robust main with many different commands (apply a move, undo a move, show the history, etc.), we'll start simple: a `Main` method in a .NET Core application that performs our abstract game loop.

```
public static void Main(string[] args) {
  IGameBoard board = ...; // construct some IGameBoard type
  IConsoleView view = ...; // construct corresponding IConsoleView

  while (!board.IsFinished) {
    // Print the board to the console.
    Console.WriteLine(view.BoardToString(board));

    // Print the possible moves.
    Console.WriteLine("Possible moves:");
    IEnumerable<IGameMove> possMoves = board.GetPossibleMoves();
    Console.WriteLine(string.Join(",",
      possMoves.Select(view.MoveToString)));

    // Print the current player and input their move.
    Console.WriteLine("It is {0}'s turn.",
      view.PlayerToString(board.CurrentPlayer));
    Console.WriteLine("Enter a move: ");
    string input = Console.ReadLine();

    // Parse the move and make sure there is an equivalent possible move.
    IGameMove toApply = view.ParseMove(input);
    IGameMove foundMove = possMoves.FirstOrDefault(toApply.Equals);
    if (foundMove == null) {
      Console.WriteLine("Sorry, that move is invalid.");
    }
    else {
```

```
      board.ApplyMove(foundMove);
    }
  }
}
```

A few lines might need some explanation:

- `... // construct some IGameBoard type`: we haven't programmed a concrete game implementation yet, so we can't actually construct a game to play. Once we do, we can finish these two lines to construct the appropriate `IGameBoard`- and `IConsoleView`-derived classes for our target game. Later we'll discuss ways of discovering all game types automatically.
- `string.Join`: this method takes a string *separator* and an enumerable of objects, and returns a single string by concatenating all the objects in the enumerable together, with one copy of the separator between each pair of adjacent objects. If the objects are not already strings, their `ToString()` method will be called. For example,

  ```
  string.Join("/", new string[] {"leanpub.com", "checkmate-csharp", "read"})
  ```

  returns the string `"leanpub.com/checkmate-csharp/read"`.
- `possMoves.Select(view.MoveToString)`: this LINQ call takes a sequence of possible moves (of unknown game type), and calls the `MoveToString()` method of the game's `IConsoleView` to convert each into a string for printing. The new sequence of strings is then `Joined` with commas.
- `possMoves.SingleOrDefault(toApply.Equals)`: another LINQ call, this finds the first entry in `possMoves` that is equivalent to the move entered by the user. Remember that the `IGameMove` interface requires overloading `bool Equals(IGameMove other)` to check for two move objects being equivalent. Note that we are not *calling* `toApply.Equals()`; we are passing a reference to the `Equals` method of `IGameMove` when bound to the specific object instance referred to by `toApply`.

Discussion

This LINQ call demonstrates two techniques from Chapter 6. We choose `SingleOrDefault` because a list of possible moves should probably not contain two different move objects that are both equal to the user's input; that would indicate an implementation error, and the exception that `SingleOrDefault` throws would indicate such to the programmer. The expression `toApply.Equals` is an example of taking a delegate to a non-static method of an object, and allows us to pass a one-argument predicate function to `SingleOrDefault` that is bound specifically to the object the user has chosen to apply.

Miscellaneous

BoardPosition

Our implementation of a board position value type:

```
public struct BoardPosition : IEquatable<BoardPosition> {
  public int Row { get; private set; }
  public int Col { get; private set; }

  public BoardPosition(int row, int col) {
    Row = row;
    Col = col;
  }

  public BoardPosition Translate(int rDelta, int cDelta) {
    return new BoardPosition(Row + rDelta, Col + cDelta);
  }

  public bool Equals(BoardPosition other) {
    return Row == other.Row && Col == other.Col;
  }
}
```

Note that `Row` and `Col` are read-only properties, with no way to mutate a `BoardPosition` object once it is created. This is intentional, to hide any confusion about value vs. reference semantics.

Discussion

Even experienced programmers often miss when an object is a `struct`. If that object has mutator methods and has been passed/copied to another variable, mutating the original object will *not* mutate the copied variable, because structs are copy-by-value (value semantics). By making `BoardPosition` immutable, this confusion can never happen, as you can't mutate the original in the first place.

We implement `IEquatable<BoardPosition>` to inherit a strongly-typed `bool Equals(BoardPosition other)` method. This allows consumers of the class to compare two `BoardPosition` objects using this method, instead of the `Object` method `bool Equals(Object other)`. (The `Object` method requires a boxing and unboxing operation when passing a value type like `BoardPosition`.)

Finally, the `Translate` utility method will be useful later when using `BoardPosition` objects to "walk" along a game board. We will often see code like:

```
BoardPosition p = ...; // some position we've already considered.
// Do something with p;
// Then move up one square, regardless of where p is.
p = p.Translate(-1, 0);
```

If the rows of a board are numbered like an array, then translating by (-1, 0) is akin to moving up one row.

`BoardDirection`

Our implementation of a board direction value type is simple enough; if you need the source, you can find it in the book source code online. Its notable properties are `RowDelta` and `ColDelta`, giving the "X" and "Y" coordinates of the direction vector, respectively. It also owns a static property named `CardinalDirections`, which is a sequence of `BoardDirection` objects representing one-square movements in the eight cardinal directions.

For convenience, we will also amend the `BoardPosition` type with a second `Translate` overload, this one taking a `BoardDirection` parameter:

```
public BoardPosition Translate(BoardDirection dir) {
  return Translate(dir.RowDelta, dir.ColDelta);
}
```

7.4 Summary

In this chapter we laid the foundations for our board game framework by defining the interfaces that allow us to program a sophisticated application *without even knowing the specific game we want to play*. By identifying the behaviors and traits common to many board games and encoding them into a C# interface, we will be able to implement data types for Othello, chess, and many other games and plug those types into our game-agnostic main application. We will demonstrate this process – implementing a game type, and integrating it into our main application – starting in the next chapter, but we can actually preview the process now with a very simple board game: tic-tac-toe[1].

Let's suppose we have a way of identifying moves on a tic-tac-toe board, and also a way of representing the current state (which player has a piece at which squares) of a tic-tac-toe board. Can you define in English a process for implementing these `IGameBoard` methods in the context of tic-tac-toe?

1. `GetPossibleMoves`: produce a list of all valid squares where the current player can move.
2. `ApplyMove`: given a move representation, update the board state with the new move.
3. `UndoMove`: with access to the most recently-applied move, revert the game state to "before" the move.
4. `IsFinished`: determine whether the game is over.
5. `CurrentAdvantage`: if the game is over, determine which player won.

[1] https://en.wikipedia.org/wiki/Tic-tac-toe

Perhaps we lack the C# prowess to actually code your ideas, but I'm confident you were able to imagine solutions to these operations. Congratulations! You've just designed your first board game implementation, and after finishing Chapter **??**, I'm sure you'll be able to program tic-tac-toe and plug it into the system we've created here.

C# lessons

- **.NET versions**: when to create Class Library vs. Application projects, and when to choose .NET Standard, .NET Library, or .NET Core.
- **`IReadOnlyList<T>`**: an interface for an immutable list whose elements can be accessed by index.
- **`string.Join`**: concatenating the elements of a list into a single string using a separator.
- **Value types**: reasons to make a new value type using the `struct` keyword.