O'REILLY®

Functional Programming with C#

Create More Supportable, Robust, and Testable Code

Early Release
RAW & UNEDITED

Simon J. Painter

Functional Programming with C#

Create More Supportable, Robust, and Testable Code

Simon J. Painter

Beijing • Boston • Farnham • Sebastopol • Tokyo

Functional Programming with C#

by Simon J. Painter

Printed in the United States of America.

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

- Acquisitions Editor: Suzanne McQuade
- Development Editor: Jill Leonard
- Production Editor: Gregory Hyman
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- November 2023: First Edition

Revision History for the Early Release

- 2022-04-20: First Release
- 2022-06-07: Second Release
- 2022-08-22: Third Release
- 2022-10-07: Fourth Release
- 2022-11-08: Fifth Release
- 2022-11-08: Fifth Release
- 2023-01-19: Sixth Release
- 2023-02-21: Seventh Release

See https://www.oreilly.com/catalog/errata.csp?isbn=9781492097075 for release details.

978-1-492-09707-5

Preface

Functional Programming (FP) is one of the greatest innovations in the history of software development, it’s also cool. Fun as well. Not only that, but it’s gaining traction year by year.

I attend developer’s conferences as often as I’m able, and I’ve noticed a trend. Each year, there always seems to be more content about Functional Programming, not less. There is often even an entire track dedicated to it, and the other talks even often include FP content somewhere as a talking point.

It’s slowly, but surely, becoming a big deal. Why is that?

With the growth of concepts like containerization and serverless applications, FP isn’t just a bit of fun for developer’s free-time projects; it’s not a fad that’ll be forgotten in a few years. It has real benefit to bring to our stakeholders.

In the .NET world there are several additional factors at play.

Mads Torgerson, the C# Lead Designer is himself a fan of functional programming, and one of the major driving forces behind the adoption of the functional paradigm into .NET. There’s also F# - the .NET functional language. F# and C# share a common runtime, consequently many functional features requested by the F# team often become available in C# in some form as well.

The big question though - What is it? And[1], will I need to learn an entire new programming language just to be able to use it? The good news is that if you’re a .NET developer, then you don’t need to spend large chunks of your own time learning a new technology just to stay up-to-date - you don’t even have to invest in another 3rd party library to add to your application’s dependencies - this is all possible with out-of-the-box C# code - albeit with a little tinkering around.

This book is going to introduce all the fundamental concepts of Functional Programming, demonstrate their benefits, and how they might be achieved in C# - not just for your own hobby programming, but with a very real eye towards how they can be used to bring immediate benefit to your work life as well.

Who Should Read This Book?

This book is intended for developers - whether professional, students, or hobbyist - who already have a basic grounding in C#. You don’t need to be an expert, but you’ll need to be familiar with the basics, and feel comfortable putting together at least a relatively simply C# application.

There will be some more advanced .NET topics covered, but I’ll provide explainations when they come up.

This book has been written with a few different catagories of people in mind:

- Those of you that have learned the basics of C#, but want to find ways to take your learning further. To learn more advanced techniques to write better, more robust code.
- .NET developers that have heard of Functional Programming, and perhaps even know what it is, but want to know how to get started writing code that way in C#.
- F# developers looking for ways to keep using the Functional toys you’re used to.
- Those migrating to .NET from another functional, or functional-supporting language (like Java).
- Anyone that really, truly loves coding. If you spend all day writing code in the office, then come home to write more for fun, then this book is probably for you.

Why I wrote this book

I’ve been interested in programming for as long as I can remember. When I was a young boy, we had a ZX Spectrum - an early British home computer developed by Sinclair Research in the early 80s. If anyone remembers the Commodore 64, it was a bit like that, but far more primitive. It had just 15 colours[2] - and one of them was black. I had the more advanced model with 48k of memory, though my Dad had the earlier machine - the ZX81 - which had a single kilobyte of memory available (and rubber keys). It couldn’t even have colored-in character sprites, just areas of the screen, so your game avatar would change colour to that of whatever they were standing in front of. In short, it was pure awesome on toast.

One of the best things about it was that it had an OS that effectively consisted of a text-based programming interface, and code was required to load a game (from a cassette tape, with the command LOAD “”), but there were also magazines and books for kids with code for games you could enter yourself, and it was from these that I developed my lasting obsession with the mysteries of computer code. Thanks so much, Usbourne Publishing!

When I was around 14 years old or so, a computer-based careers advice program at school suggested I could think about taking up a career in software development. This is the first time I realised that you could take this silly hobby and turn it into something that you could actually make money from!

After University was over, it was time to get a proper job, and that was where I got my first exposure to C#. So, the next step, I supposed, was to learn how to develop code *properly*. Easy, right? I’ll be honest, nearly 2 decades on, and I’m still trying to work that out.

One of the big turning points for me in my programming career was when I attended a developer’s conference in Norway, and finally started to understand what this “Functional Programming” thing I’d been hearing about was actually about. Functional code is elegant, concise and easy to read in a way that other forms of code just don’t seem to be. As with any type of code, it’s still possible to write horrible-looking codebases, but it still fundamentally feels like it’s finally code being done *properly*, in a way that I’ve never really felt from other styles of coding. Hopefully after reading this book, you’ll not only agree, but be interested in searching out the many other avenues that exist out there for exploring it further.

Navigating This Book

This is how I’ve organized this book:

The introduction talks about what exactly Functional Programming is, where it comes from, and why any of us should be interested in it. I argue that it brings significant business benefits to our employers, and that it’s a skill worth adding to your developer toolbelt.

- Chapter 1 looks at what you can do right now to start coding Functionally in C#, without having to reference a single new Nuget package, 3rd party library or hack around with the language. Nearly all of the examples in this chapter work with just about every version of C# since version 3. This chapter represents the very first steps into Functional Programming, all fairly easy code, but it sets the groundwork for what's to come.

- Chapter 2 provides some slightly less conventional ways to look at structures already available to us in C#. It includes ways to take the Functional Paradigm a bit futher. At this point there are still no extra code dependencies, but things start to look a bit more unusual here.
- Chapters 4 to 7 each show a component of the Functional Programming paradigm and how you can go about implementing it in C#. It's in these chapters that we start to play around with the structure of C# a little.
- Chapters 8 and 9 talk more about the practicalities of uing Functional C# in a business environment.

Feel free to dive in at the level you feel ready for. This isn't a novel[3], read the chapters in the order that makes sense to you.

Acknowledgments

The very first person I should thank is Kathleen Dollard. She gave a talk at NDC Oslo some years ago called "Functional Techniques for C#". It was the first real exposure I'd ever had to Functional Programming, and it was a real eye-opener (https://www.youtube.com/watch?v=rHmIf5xmKQg).

The other guru I’ve followed on this trail is Enrico Buananno, who’s book “Functional Programming in C#” (ISBN: 978-1617293955) was the first that allowed me to properly understand for the first time, how some of the hard-to-grasp functional concepts worked.

Ian Russell, Matthew Fletcher, Liam Riley, Max Dietze, Steve “Talks Code” Collins, Gerardo Lijs, Matt Eland, Rahul Nath, Siva Gudivada, Christian Horsdal, Martin Fuß, Dave McCollough, Sebastian Robbins, David Schaefer, Peter De Tender, Mark Seeman who read the early drafts and provided invaluable feedback. Thanks, folks!

My editor, Jill Leonard. She must have the patience of a saint to put up with me for a whole year!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data

types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at LINK TO COME.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Functional Programming with C#* by Simon J. Painter (O'Reilly). Copyright 2024 Simon Painter, 978-1-492-09707-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

Dedication

This book is dedicated to my wife, Sushma Mahadik. My Billi. Also to my two daughters, Sophie and Katie. Your daddy loves you, girls.

OK, The **two** questions

8 base colors, and a bright version of each. One was Black, though, and how on earth can you have bright black! So..15.

but if it were, you can guarantee the Butler would have done it!

Chapter 1. Introduction

If you’ve learned much programming before now - whether that be in C#, Visual BASIC, Python or whatever - then chances are what you learned was based around the programming paradigm that is currently the most dominent - Object Oriented.

Object-Oriented programming has been around for quite a long time. The precise date is a matter for debate, but it was likely invented somewhere around the late 50s and early 60s.

Object-Oriented coding is based around the idea of wrapping pieces of data - known as properties - and functionality into logical blocks of code called *Classes*, which are used as a sort of template from which we instantiate *Objects*. There’s a lot more to it: inheritance, polymorphism, Virtual and Abstract methods. All sorts of stuff like that.

This is however, not an Object-Oriented programming book. In fact, if you are already experienced with OO, you’ll probably get more from this book if you leave what you know already to one side.

What I’m going to be describing in this book is a style of programming that serves as an alternative to OO - functional programming. Functional Programming, despite gaining some mainstream recognition in the last few years, is actually as old - if not older - than OO. It’s based on mathematic principles that were developed by various people between the late 1800s and 1950s, and has been a feature of some programming languages since the 1960s.

In this book, I’ll be showing you how to implement it in C# without the necessity of learning a whole new programming language.

Before we get cracking with some code, I’d like to talk first about Functional Programming itself. What is it? Why should we be interested? When is it best used. All very important questions.

What is Functional Programming?

There are a few basic concepts in Functional Programming, many of which have fairly obscure names for what are otherwise not terribly difficult concepts to understand. I’ll try to lay them out here as simply as I can.

Is it a Language, an API, or what?

No, Functional Programming isn’t a language or a 3rd party plug-in library in Nuget, it’s a *paradigm*. What do I mean by that? There are more formal definitions of paradigms, but I think of it as being a *style* of programming. Like a guitar might be used as the exact same instrument, but to play many, often wildly different, styles of music, so also some programming languages offer support for different styles of working.

Functional programming is also as old as Object-Oriented coding - if not older. I’ll talk more about its origins later, but for now just be aware that it is nothing new, and the theory behind it not only predates OO, but largely also the computing industry itself.

It’s also worth noting that you can combine paradigms, like mixing rock and jazz. Not only can they combine, but there are times when you can use the best features of each to produce a better end result.

Programming paradigms come in many, many flavors[1] but for the sake of simplicity I’m only going to talk about the two most common in modern programming:

Imperative

This was the only type of programming paradigm for quite a long time. Procedural and Object-Oriented (OO) belong to this category. These styles of programming involve more directly instructing the executing environment with the steps that need to be executed in detail, i.e. Which variable contains which intermediate steps and how the process is carried out step-by-step in minute detail. This is programming as it usually gets taught in school/college/university/at work [delete where appropriate].

Declarative

In this programming paradigm we’re less concerned with the precise details of how we accomplish our goal, the code more closely resembles a description of what is desired at the end of the process, and the details (including things such as order of execution of the steps) are left more in the hands of the execution environment. This is the category Functional Programming belongs to. SQL also belongs here, so in some ways Functional Programming more closely resembles SQL than OO. When writing SQL statements you aren’t concerned with what the order of operations are (it’s not really SELECT then WHERE then ORDER BY), you aren’t concerned with how exactly the data transformations are carried out in detail, you just write a script that effectively describes the desired output. These are some of the goals of Functional C# as well, so those of you with a background working with SQL Server or other relational databases might find some of the ideas coming up easier to grasp than those that haven't.

There are many, many more paradigms besides these, but they're well beyond the scope of this book. In fairness, most of them are pretty obscure besides these two, so you're unlikely to run into them any time soon.

The Properties of Functional Programming

For the next few sections, I'm going to talk about each of the properties of Functional Programming, and what they really mean to a developer.

Immutability

If something can *change* then it can also said that it can *mutate*, like a Teenage Mutant Ninja[2] Turtle. The other way of saying something can mutate is that it is *mutable*. If, on the other hand, something cannot change at all, then it is *immutable*.

In programming, this refers to Variables that have their value set upon being defined, and after that point they may never be changed again. If a new value is required, then a new variable should be created, based on the old one. This is how all variables are treated in Functional code.

It’s a slightly different way of working compared to Imperative code, but it ends up producing programs that more closely resemble mathematical working, and encourages good structure, and more predictable - and hence *more robust* - code.

`DateTime` and `String` are both immutable data structures in .NET. You may *think* you’ve altered them, but behind the scenes every alteration creates a new item on the stack. This is why most new developers get the talk about concatenating strings in `For` loops and why you should never, *ever* do it.

Higher-order Functions

These are functions that are passed around as variables. This may be either as local variables, parameters to a function or return values from a function. The `Func<T,TResult>` or `Action<T>` delegate types are perfect examples of this.

In case you aren’t familiar with these delegates, this is how they work in brief.

They’re both forms of function stored in the form of variables. They both take sets of generic types, which represent their parameters and return types - if any. The difference between `Func` and `Action` is that `Action` doesn’t return any value - i.e. it’s a `void` function that won’t contain a `return` keyword. The last generic type listed in a `Func` is its return type.

These functions:

```
// Given parameters 10 and 20, this would output
// "10 + 20 = 30"
public string ComposeMessage(int a, int b)
{
 return a + " + " + b + " = " + (a + b);
}

public void LogMessage(string a)
{
 this.Logger.LogInfo("message received: " + a);
}
```

Could be re-written as delegate types like this:

```
Func<int, int, string> ComposeMessage =
 (a, b) => a + " + " + b + " = " + (a + b);

Action<string> LogMessage = a =>
 this.Logger.LogInfo($"message received: {x}");
```

These delegate types can be called exactly as if they were normal functions:

```
var message = ComposeMessage(10, 20);
LogMessage(message);
```

The big advantage of using these Delegates types is they’re stored in variables that can be passed around the codebase. They can be included as parameters to other functions, or as return types. Used properly, they’re among the more powerful features of C#.

Using Functional Programming techniques, Delegate types can be composed together to create larger, more complex functions from smaller, functional building blocks. Like lego bricks being placed together to make a model *Millennium Falcon*, or whatever you prefer. This is the real reason this paradigm is called **Functional** Programming, because we build our applications up with *functions*, not, as the name suggests, that the code written in other paradigms don’t function. Why would anyone ever use them if they didn’t?

In fact - a rule of thumb for you. If there’s a question, Functional Programming’s answer will amost certainly be “Functions, Functions and more Functions”.

NOTE

There are two kinds of callable code modules. Functions and Methods. The difference is that functions always return a value, but methods don't. In C#, functions return data of some kind, whereas methods have a return type of `void`. Methods almost inevitable involve side-effects, and as such we should avoid any use of them in our code - except where unavoidable. Logging might be an example of usage of methods that is not only unavoidable, but also essential to good production code.

Expressions Rather than Statements

A couple of definitions are required here.

Expressions are discrete units of code that evaluate to a value. What do I mean by that?

In its simplest form, these are expressions:

```
const int exp1 = 6;
const int exp2 = 6 * 10;
```

We can feed values in too, to form our expressions, so this is one as well:

```
public int AddTen(int x) => x + 10;
```

This too. It does carry out an operation - i.e. evaluate a boolean, but it's used ultimately to return a `bool`, so it's an expression:

```
public bool IsTen(int x) => x == 10;
```

you can also consider ternary `if` statements to be expressions, if they're used purely for determinging a value to return:

```
var randomNumber = this._rnd.Generate();
var message = randomNumber == 10
 ? "It was ten"
 : "it wasn't ten";
```

Another quick rule of thumb - if a line of code has a single equals sign, it's likely to be an expression, because it's assigning a value to something. There's some gray area in that rule. Calls to other functions could have all sorts of unforseen consequences. It's not a bad rule to keep in mind, though.

Statements on the other hand are pieces of code that *don't* evaluate to data. These take the form more of an instruction to do something. Either an instruction to the executing environment to change the order of execution via keywords like `if`, `where`, `for`, `foreach`, etc. or calls to functions that don't return anything - and by implication instead carry out some sort of operation. Something like this:

```
this._thingDoer.GoDoSomething();
```

A final rule of thumb[3] if there is *no* equals sign, it is *definitely* a statement.

Expression-based Programming

If it helps, think back to mathematics lessons from your school days. Remember those lines of working you used to have to write out when you were producing your final answer? Expression-based programming is like that.

Each line is a complete calculation, and builds on one or more previous lines. By writing expression-based code, you're leaving your working behind, set in stone while the function runs. Amongst other benefits, it's easier to debug, because you can look back at all of the previous values and know they've not been changed by a previous iteration of a loop, or anything like that.

This might seem like an impossibility, almost like being asked to program with your arms tied behind your back. It's entirely possible though, and not even necessarily difficult. The tools have mostly been there for about a decade in C#, and there are plenty of more effective structures.

Here's an example of what I mean:

```
public decimal CalculateHypotenuse(decimal b, dec
{
 var bSquared = b * b;
```

```
   var cSquared = c * c;
   var aSquared = bSquared + cSquared;
   var a = Math.Sqrt(aSquared);
   return a;
  }
```

Now strictly speaking, you could write that out as one long line, but it wouldn't look so nice & easy to read and understand, would it? I could also write it something like this to save all the intermediate variables:

```
  public decimal CalculateHypotenuse(decimal b, dec
  {
   var returnValue = b * b;
   returnValue += c * c;
   returnValue = Math.Sqrt(returnValue);
   return returnValue;
  }
```

The issue here is that it's a little harder to read without variable names, and also all of the intermediate values are lost - if there was a bug we'd have to step through and examine `returnValue` at each stage. In the expression-based solution all of the working is kept where it is.

After a little experience working in this manner, it will actually seem odd and even a little awkward and clunky to go back to the old way.

Referential Transparency

This is a scary-sounding name for a simple concept. There is a concept in functional programming called “pure functions”. These are functions with the following properties:

- They make no changes to anything outside of the function. No state can be updated, no files stored, etc.
- Given the same set of parameter values, they will always return the exact same result. No. Matter. What. Regardless of what state the system is in.
- They don’t have any unexpected side effects. Exceptions being thrown is included in that.

The terms comes from the idea that given the same input, the same output always results, so in a calculation you can essentially swap the function call with the final value, given those inputs. In this example:

```
var addTen = (int x) => x + 10;
var twenty = addTen(10);
```

The call to addTen with a parameter of 10 will *always* evaluate to 20, with no exceptions. There are no possible side effects in a function this simple, either. Consequently, the reference to addTen(10) could in principle be exchanged for a constant value of 20 with no side effects. This is Referential Transparency.

Here are some pure functions:

```
public int Add(int a, int b) => a + b;

public string SayHello(string name) => "Hello " +
 (string.IsNullOrWhitespace(name)
  ? "I don't believe we've met.  Would you like a
  : name);
```

Note no side effect can occur (I made sure a null check was included with the string) and nothing outside the function is altered, only a new value generated and returned.

Here are impure versions of those same functions:

```
public void Add(int a) => this.total += a; // Alt

public string SayHello() => "Hello " + this.Name;
// Reads from state instead of a parameter value
```

In both of these cases there are reference to properties of the current class that are beyond the scope of the function itself. The Add function even modifies that state property. No null check on the SayHello function either. All of these factors mean we cannot consider these functions to be “pure”.

How about these?

```
public string SayHello() => "Hello " + this.GetNa

public string SayHello2(Customer c)
{
 c.SaidHelloTo = true;
 return "Hello " + (c?.Name ?? "Unknown Person")
}

public string SayHello3(string name) =>
 DateTime.Now + " - Hello " + (name ?? "Unknown F
```

None of these are likely to be pure.

SayHello relies on a function outside itself. I don't actually know what `GetName()` does[4]. If it's simply returning a constant, then we *can* consider `SayHello()` to be pure. If on the other hand, it's doing a lookup in a database table, then the possibility exists for missing data or lost network packets resulting in errors being thrown, all examples of unexpected side effects. If a function *had* to be used for retrieving the name, I'd consider re-writing this with a `Func<T,TResult>` delegate to inject the functionality safely into our SayHello function.

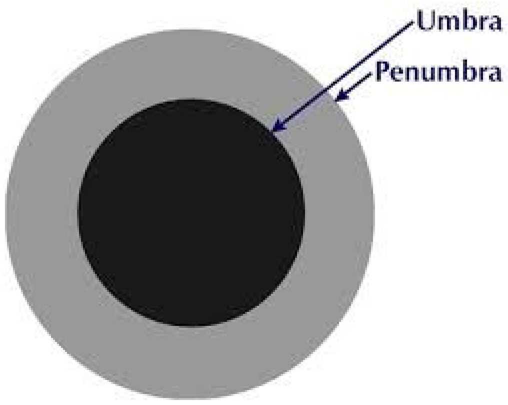
SayHello2 modifies the object being passed in - a clear side effect from use of this function. Passing objects by reference and modifying them like this isn’t all that unusual a practice in Object-Oriented code, but it’s absolutely not a thing done in functional programming. I’d perhaps make this pure by separating out the update to the object properties and the processing of saying hello into separate funtions.

SayHello3 uses `DateTime.Now`, which returns a different value each and every time it’s ever used. The absolute oposite of a pure function. One easy way to fix this is by adding a `DateTime` parameter to the function and passing the value in.

referential transparency is one of the features that massively increases the testability of functional code. It does mean that other techniques have to be used to track state, I’ll get into that later.

There is also a limit to how much “purity” we can have in our application, especially once we have to interact with the outside world, the user, or some 3rd party libraries that don’t follow the functional paradigm. In C#, we’re always going to have to make compromises here or there.

There’s a metaphor I usually like to wheel out at this point. A Shadow has two parts: the Umbra and Penumbra[5]. The Umbra is the solid dark part of a shadow, most of the shadow in fact. The Penumbra is the grey fuzzy circle around the outside, the part where Shadow and Not-Shadow meet and one fades into the other. In C# applications, I imagine that the pure area of the code base is the Umbra, and the areas of compromise are the Penumbra. My task is to maximize the Pure area, and minimize as much as humanly possible the non-Pure area.



If you want a more formal definition of this architectural pattern, Gary Bernhardt has given talks calling it Functional Core, Imperative Shell[6].

Recursion

If you don’t understand this, see: Recursion Otherwise, see: Seriously, Recursion

Seriously, Recursion

Recursion has been around for as long as programming, pretty much. It’s a function that calls itself in order to effect an indefinite (but hopefully not infinite) loop. This should be familiar to anyone that’s ever written code to traverse a folder structure, or perhaps written an efficient sorting algorithm.

A recursive function typically comes in 2 parts:

- A condition, used to determine whether the function should be called again, or whether an end-state has been reached (e.g. the value we’re trying to calculate has been found, there are no sub-folders to explore, etc.)
- A return statement, which either returns a final value, or references the same function again, depending on the outcome of the end-state condition.

Here is a very simple recursive Add[7]:

```
public int AddUntil(int startValue, int endValue
{
        if (startValue >= endValue)
                return startValue;
        else
                return AddUntil(startValue + 1, 
}
```

Silly though the example above is, note that I never change the value of either parameter integers. Each call to the recursive function used parameter values based on the ones it itself received. This is another example of *Immutability* - I’m not changing values in a variable, I’m making a call to a function using using an expression based on the values received.

Recursion is one of the methods Functional Programming uses as an alternative to statements like While and ForEach. There are some performance issues in C#, however. There is a whole chapter coming up at a later point to discuss Recursion in more detail, but for now just use it cautiously, and stick with me. All will become clear…

Pattern Matching

In C# this is basically `Switch` statements with "go-faster" stripes. F# takes the concept further, though. We've pretty much had this in C# for a few versions now. The `Switch` expressions introduced in C# 8 introduced our own native implementation of this concept, and the Microsoft team has been enhancing it regularly.

It's switching where you can change the path of execution on the type of the object under examination, as well as its properties. It can be used to reduce a large, set of nested if-statements like this:

```
public int NumberOfDays(int month, bool isLeapYea
{
 if(month == 2)
 {
  if(isLeapYear)
   return 29;
  else
   return 28;
 }
 if(month == 1 || month == 3 || month == 5 || mon
  month == 8 || month == 10 || month == 12)
   return 31;
  else
   return 30;
}
```

into a few fairly concise lines, like this:

```
  public int NumberOfDays(int month, bool isLeapYea
          (month, isLeapYear) switch
          {
                  { month: 2, isLeapYear: true } =>
                  { month: 2 } => 28,
                  { month: 1 or 3 or 5 or 7 or 8 o
                  _ => 31
          };
```

It’s an incredible, powerful feature, and one of my favourite things[8].

There are many examples of this coming up in the next couple of chapters, so skip ahead if you’re interested in seeing more about what this is all about.

Also, for those stuck using older versions of C#, there are ways of implementing this, and I’ll show a few tips on it later.

Stateless

Object-Oriented code typically has a set of state objects, which represent a process - either real, or virtual. These state objects are updated periodically, to keep in sync with whatever it is they represent. Something like this, for example:

```
public class DoctorWho
{
        public int NumberOfStories { get; set; }
        public int CurrentDoctor { get; set; }
        public string CurrentDoctorActor { get; s
        public int SeasonNumber { get; set; }
}

public class DoctorWhoRepository
{
        private DoctorWho State;

        public DoctorWhoRepository(DoctorWho init
        {
                this.State = initialState;
        }
        public void AddNewSeason(int storiesInSea
        {
                this.State.NumberOfStories += sto
                this.State.SeasonNumber++;
        }

        public void RegenerateDoctor(string newAc
        {
                this.State.CurrentDoctor++;
                this.State.CurrentDoctorActor = n
        }
}
```

Well, forget ever doing that again if you want to do Functional Programming. There is no concept of a central state object, or of modifying its properties, like in the code sample, above.

Seriously? Feels like purest craziness, doesn't it? Strictly, there **is** a state, but it's more of an emergent property of the system.

Anyone that's ever worked with React-Redux has already been exposed to the Functional approach to state (which was, in turn, inspired by the Functional Programming language, *Elm*). In Redux, the application state is an immutable object, which isn't updated, but instead a function is defined by the developer that takes the old state, a command, and any required parameters, then returns a new state object based on the old one. This process became infinitely easier in C# with the introduction of Record types in C# 9. I’ll talk more on that later. For now though, a simple version of how one of the repository functions might be refactored to work functionally might be something like this:

```
public DoctorWho RegenerateDoctor(DoctorWho olc
{
  return new DoctorWho
  {
    NumberOfStories = oldState.NumberOfStories,
    CurrentDoctor = oldState.CurrentDoctor + 1,
    CurrentDoctorActor = newActorName,
    SeasonNumber = oldState.SeasonNumber
  };
}
```

There would obviously be a difference in how this was used, outside the repository as well. In fact, calling it a repository is probably a bit of a mistake now. I’ll talk more about the strategies required for writing code without state objects later. Hopefully this is enough to get an idea of how Functional code might work.

Baking Cakes

If you want a slightly higher level of description of the difference between those paradigms. Here's how they'd both make cupcakes[9]:

An Imperative Cake

This isn't real C# code, just a sort of .NET themed pseudo-code to give an impression of the imperative solution to this imaginary problem.

```
Oven.SetTemperatureInCentigrade(180);
for(int i=0; i < 3; i++)
{
        bowl.AddEgg();
        bool isEggBeaten = false;
        while(!isEggBeaten)
        {
                Bowl.BeatContents();
                isEggBeaten = Bowl.IsStirred();
        }
}
for(int i == 0; i < 12; i++)
{
        OvenTray.Add(paperCase[i]);
        OvenTray.AddToCase(bowl.TakeSpoonfullOfC
}
Oven.Add(OvenTray);
Thread.PauseMinutes(25);
Oven.ExtractAll();
```

For me, this represents typical convoluted imperative code. Plenty of little short-lived variables cooked up to track state. It’s also very concerned with the very precise order of things. It’s more like instructions given to a robot with no intelligence at all, needing everything spelled out for them.

A Declarative Cake

Here’s what an entirely imaginary bit of Declaritive code might look like to solve the same problem:

```
    Oven.SetTemperatureInCentigrade(180);
    var cakeBatter = EggBox.Take(3)
      .Each(e => Bowl.Add(e)
                     .Then(b =>
                         b.While(x => !x.IsStirred,
                       )
                     )
            .DivideInto(12)
        .Each(cb =>
          OvenTray.Add(PaperCaseBox.Take(1).Add(cb
        );
```

That might look odd and unusual for now, if you’re unfamililar with Functional Programming, but over the course of this book, I’m going to explain how this all works, what the benefits are, and how you can implement all of this yourself in C#.

What's worth noting though, is that there are no state tracking variables, no `If` or `While` statements. I'm not even sure what the order of operations would necessarily be, and it doesn't matter, because the system will work so that any necessary steps are completed at the point of need.

This is more like instructions for a slightly more intelligent robot. One that can think a little for itself, at least as far as instructions that might sound something like "do this until such-and-such a state exists" which in procedural code would exist by combining a While loop and some state tracking code lines.

Where does Functional Programming Come From?

The first thing I want to get out of the way is that despite what some people might think, Functional Programming is old. *Really* old - by computing standards at least. My point being - it isn't like the latest trendy JavaScript framework, here this year, so much old news next year. It predates all modern programming languages, and even computing itself to some extent. Functional has been around for longer than any of us, and it's likely to be around long after we're all happily retired. My slightly belabored point is that it’s worth investing your time and energy to learn and understand it. Even if one day you find yourself no longer working in C#, most other languages support Functional concepts to a greater or lesser degree (JavaScript does to an extent that most languages can only dream of), so these skills will remain relevant throughout the rest of your career.

A quick caveat before I continue with this section - I’m not a mathematician. I love mathematics, it was one of my favourite subjects at school, college & university, but there eventually comes a level of higher, theoretical mathematics that leaves even me with glazed-over eyes and a mild headache. That said, I’ll do my best to talk briefly about where exactly Functional Programming came from. Which was, indeed, that very world of theoretical mathematics.

The first figure in the history of Functional Programming most people can name is usually Haskell Brooks Curry (1900-1982), an American mathematician that now has no fewer than three programming languages named after him, as well as the Functional concept of “Currying” (of which, more later). His work was on something called “Combinatory Logic” - a mathematical concept that involves writing out functions in the form of lambda (or arrow) expressions, and then combining them to create more complex logic. This is the fundamental basis of Functional Programming. Curry wasn’t the first to work on this though, he was following on from papers and books written by his mathematical predecessors, people like:

- Alonzo Church (1903-1955, American) - It’s Church that coined the term “Lambda Expression” that we use in C#, and other languages, to this day.
- Moses Schönfinkel (1888-1942, Russian) - Schönfinkel wrote papers on Combinatory logic that were one of the bases for Haskell Curry’s work
- Friedrich Frege (1848-1925, German) - Arguably the first person to describe the concept we now know as Currying. As important as it is to credit the correct people with discoveries, Freging doesn’t quite have the same ring[10].

The first Functional programming languages were:

- IPL (Information Processing Language), developed in 1956 by Allen Newell (1927-1992, American), Cliff Shaw (1922-1991, American) and Herbert Simon (1916-2001, American)
- LISP (LISt Processor), developed in 1958 by John McCarthy (1927-2011, American). I hear tell that LISP still has its fans to this day, and is still in production use in some businesses. I’ve never seen any direct evidence of this myself, however.

Interestingly, neither of these languages are what you would call “pure” functional. Like C#, Java, and numerous other languages, they adopted something of a hybrid approach, unlike the modern “pure” functional languages, like Haskell and Elm.

I don’t want to dwell too long on the (admittedly, fascinating) history of Functional Programming, but it’s hopefully obvious from what I have shown, that it has a long and illustrious pedigree.

Who Else Does Functional Programming?

As I’ve already said, Functional Programming has been around for a while, and it’s not just .NET developers that are showing an interest. Quite the opposite, many other languages have been offering Functional Paradigm support for a lot longer than .NET.

What do I mean by support? I mean that it offers the ability to implement code in the Functional Paradigm. This comes in roughly two flavours:

Pure Functional Languages
Intended for the developer to write exclusively Functional code. All variables are immutable, offers Currying, Higher-order Functions, etc. out-of-the-box. Some features of Object-Orientation might be possible in these languages, but it’s very much a secondary concern to the team behind them.

Hybrid or Multi-Paradigm Languages

> These two terms can be used entirely interchangably. They describe programming languages that offer the features to allow code to be written in two or more paradigms. Often two or more at the same time. Supported paradigms are typically Functional and Object-Oriented. There may not be a perfect implementation available of any supported paradigms. It’s not unusual for Object Orientation to be fully supported, but not all of the features of Functional to be available to use.

Pure Functional Languages

There are also well over a dozen pure functional languages around, here is a brief look at the most popular three in use today:

> *Haskell*
> Haskell is used extensively in the banking industry. It’s often recommended as a great starting place for anyone wanting to really, really get to grips with Functional Programming. This may well be the case, but honestly, I don’t have the time or headspace free to learn an entire programming language I never intend to use in my day job.

If you’re really interested in becoming an expert in the Functional Paradigm before working with it in C#, then by all means go ahead and seek out Haskell content. A frequent recommendation for that is “Learn You a Haskell For Great Good” by Miran Lipovača[11]. I have never read this book myself, but friends of mine have and say it’s great.

Elm

Elm seems to be gaining some traction these days, if for no other reason that the Elm system for performing updates in the UI has been picked up and implemented in quite a few other projects, including ReactJS. This “Elm Architecture” is something I want to save for a later chapter.

Elixir

A general-purpose programming language based on the same Virtual Machine that Erlang runs on. It’s very popular in industry and even has its own conferences annually.

PureScript

PureScript compiles to JavaScript, so it can be used to create functional front-end code, as well as server-side code and desktop applications in isometric programming environments - i.e. those like Node.JS that allow the same language to be used client and server side.

Is It Worth Learning a Pure Functional Language First?

For the time being at least, OO is the dominent paradigm for the vast majority of the software development world, and the Functional Paradigm something that has to be learned afterwards. I don’t rule out that changing in future, but for now at least, this is the situation we’re in.

I have heard people argue the point that, coming from OO, it would be best to learn Functional Programming in its pure form *first* then come back to apply that learning in C#.

If that’s what you *want* to do, go for it. Have fun. I have no doubt that it’s a worthwhile endeavor.

To me, this perspective puts me in mind of those teachers we used to have here, in the UK that insisted that children should learn Latin, because as the root of many European languages, knowledge of Latin can easily be transferred to French, Italian, Spanish, etc.

I disagree with this somewhat [12]. Unlike Latin, Pure Functional languages aren’t necessarily *difficult*, though they are very unlike Object-Oriented development. In fact, there are fewer concepts to learn witih FP compared to OO. This said - those that have spent their careers heavily involved in OO development will likely find it harder to adjust.

Where Latin and pure functional languages are similar though is that they represent a purer, ancestral form. They are both of only limited value outside of a small number of specialist interests.

Learning Latin is also almost entirely *useless* unless you’re interested in Law, classical literature, Ancient History, etc. It’s far more useful to learn modern French or Italian. They’re easier languages to learn by far, and you can use them *now* to visit lovely places and talk to the nice people that live there. There are some great French-language comics from Belgium too. Check ‘em out. I’ll wait.

In the same way, very few places will ever actually use Pure Functional languages in production. You’d be spending a lot of time having to make a complete shift in the way you work, and end up learning a language you’ll probably never use outside of your own hobby code. I’ve been doing this job for a long time, and I’ve never yet encountered a company using anything more progressive in actual production than C#.

The lovely thing about C# is that is supports both OO *and* Functional style code, so you can shift between them as you please. Use as many features from one paradigm or the other as you like without any penalty. The paradigms can sit fairly comfortably alongside each other in the same codebase, so it’s an easy environment to transition from pure OO to Functional at a pace that suits you, or vice-versa.

That isn’t possible in a pure Functional language, even if there are a lot of Functional features that aren’t possible in C#.

What about F#? Should I be learning F#?

This is probably the most common question I get asked. What about F#? It’s not a pure Functional language, but the needle is far closer to being a proper implementation of the paradigm than C#. It has a wide variety of functional features straight out-of-the-box, as well as being easy to code in and highly performant - why not use that?

I always like to check the available exits in the room before I answer this question. F# has a passionate userbase, and they are probably all much smarter folks than me[13]. But…

It’s not because F# isn’t easy to learn. It is, from what I’ve seen, and most likely it’s easier to learn than C# if you’re entirely new to programming.

It’s not that F# won’t bring business benefits, because I honestly believe it will.

It’s not that F# can’t do absolutely everything any other language can do. It most certainly can. I’ve seen some impressive talks on how to make full-stack F# web applications.

It’s a professional decision. It isn’t hard to find C# developers, at least in any country I’ve ever visited. If I were to put the names of every attendee of a big developers’ conference in a hat and draw one at random, there’s a better than even chance it would be someone that can write C# professionally. If a team decides to invest in a C# codebase, it’s not going to be much of a struggle to keep the team populated with engineers that will be able to keep the code well maintained, and the business relatively content.

Developers that know F# on the other hand are relatively rare. I don’t know many. By adding F# into your codebase you may be putting a dependency on the team to ensure you always have enough people available that know it, or else take a risk that some areas of the code will be hard to maintain, because few people know how.

I should note that the risk isn’t as high as introducing an entirely new technology, like, say, Node.JS. F# is still a .NET language and compiles to the same Intermediate Language. You can even easily reference F# projects from C# projects in the same solution. It would still be an entirely unfamiliar syntax to the majority of .NET developers, however.

It’s my firm wish that this changes as time goes on. I’ve liked very much what I’ve seen of F#, and I’d love to do more of it. If my boss told me that a business decision had been made to adopt F#, I’d be the first to cheer!

Fact is though, it’s not really a very likely scenario at present. Who knows what the future will bring. Maybe a future edition of this book will have to be heavily re-written to accomodate all the love for F# that’s suddenly sprung up, but for now I can’t see that on the near horizon.

My recommendation would be to try this book first. If you like what you see, maybe F# might be the next place you go on your Functional journey.

Multi-Paradigm Languages

It can probably be argued that all languages besides the Pure Functional languages are some form of hybrid. In other words, that at least *some* aspects of the functional paradigm can be implemented. That's likely true, but I'm just going to look briefly at a few where it can be implemented entirely, or mostly, and as a feature provided explicitly by the team behind it.

JavaScript

JavaScript is of course almost the wild-west of programming languages in the way that nearly anything can be done with it, and it does Functional very, very well. Arguably better than it does Object Orientation. Have a look for *Javascript: The Good Parts* by Douglas Crockford and some of his online lectures (for example *https://www.youtube.com/watch?v=_DKkVvOt6dk*) if you want an insight into how to do JS Functionally and properly.

Python

Python has rapidly become a favourite programming language for the open source community, just over the last few years. It surprised me to find out it's been around since the late 80s! Python supports higher-order functions and has a few libraries available: *itertools* and *functools* to allow further functional features to be implemented.

Java

The Java platform has the same level of support for Functional features as .NET. Further to that, there are spin-off projects such as Scala, Clojure and Kotlin that offer far more Functional features than the Java language itself does.

F#

I’ve already discussed this at length in a previous section, so I won’t go into it much more now. This is .NET’s more purely Functional style language. It’s also possible to have interoperability between C# and F# libraries, so you can have projects that utilize all the best features of both.

C#

Microsoft has slowly been adding in support for Functional Programming ever since somewhere near the beginning. Arguably the introduction of Delegate Covariance and Anonymous Methods in C# 2.0 all the way back in 2005 could be considered the very first item to support the Functional paradigm. Things didn’t really get going properly until the following year when C# 3.0 introduced what I consider one of the most transformative features ever added to C#- LINQ.

I’ll talk more about it later, but LINQ is deeply rooted in the Functional paradigm, and one of our best tools for getting started writing Functional-style code in C#. In fact, it’s a stated goal of the C# team that each version of C# that is released should contain further support for Functional

Programming than the one before it. There are a number of factors driving this decision, but amongst them is F#, which often requests new functional features from the .NET runtime folks that C# ends up benefiting from too.

The Benefits of Functional Programming

I hope that you picked this book up because you're already sold on Functional Programming and want to get started right away. This section might be useful for team discussions about whether or not to use it at work.

Concise

While not a feature of Functional Programming, my favourite of the many benefits is just how concise and elegant it looks, compared to Object-Oriented or Imperative code.

Other styles of code are much more concerned with the low-level details of *how* to do something, to the point that sometimes it can take an awful lot of code-staring just to work out what that something even *is*. Functional programming is orientated more towards describing *what* is needed. The details of precisely which variables are updated how and when to achieve that goal are less of our concern.

Some developers I've spoken to about this have disliked the idea of being less involved with the lower levels of data processing, but I'm personally happier to let the execution environment take care of that, then it’s one thing fewer that I need to be concerned with.

It feels like a minor thing, but I honestly love how concise Functional code is compared to the Imperative alternatives. The job of a developer is a difficult one[14], and we often inherit complex codebases that we need to get to grips with quickly. The longer and harder it is for you to work out what a function actually does, the more money the business is losing by paying you to do that, rather than writing new code. Functional code often reads in a way that describes - in something approaching natural language - what it is that’s being accomplished. It also makes it easier to find bugs, which again saves time and money for the business.

Testable

One thing a lot of people describe as their favorite feature of functional programming is how incredibly testable it is. It really is, as well. If your codebase isn’t testable to something close to 100%, then there’s a chance you didn’t follow the paradigm correctly.

Test-Driven Development (TDD) and Behavior-Driven Development (BDD) are important professional practices now. These are programming techniques that involve writing automated unit tests for the production code *first*, then writing the real code required to allow the test to pass. It tends to result in better-designed, more robust code. Functional Programming enables these practices neatly. This in turn results in better codebase and fewer bugs in production.

Robust

It’s also not just the testability that results in a more robust codebase. Functional Programming has structures within it that actively prevent errors either from occurring in the first place.

That, or they prevent any unexpected behaviour further on, making it easier to report the issue accurately. There is no concept of NULL in Functional Programming. That alone saves an incredible number of possible errors, as well as reducing the number of automated tests that need to be written.

Predictable

Functional code starts at the beginning of the code block and works its way to the end. Exclusively in order. That’s something you can’t say of Procedural Code, with its Loops and branching If statements. There is only a single, easy to follow flow of code.

When done properly there aren’t even any Try/Catch blocks, which I’ve often found to be some of the worst offenders when it comes to code with an unpredictable order of operations. If the Try isn’t small in scope and tightly coupled to the Catch, then sometimes it can be the code equivalent of throwing a rock blindly up into the air. Who knows where it'll land and who or what might catch it. Who can say what unexpected behavior might arise from such a break in the flow of the program.

Improperly designed Try/Catch blocks have been at the back of many instances of unexpected behavior in production that I've observed over my career, and it's a problem that simply doesn't exist in the Functional paradigm.

Improper error handling is still possible in Functional code, but the very nature of Functional Programming discourages it.

Better Support for Concurrency

There are two recent developments in the world of software devleopment that have become very important in the last few years:

Containerization

This is provided by products such as Docker and Kubernetes, amongst others. This is the idea that instead of running on a traditional server[15] the application runs on something sort of like a mini-Virtual Machine (VM) which is generated by a script at deploy time. It isn't quite the same, there's no hardware emulation, but from a user perspective the result is roughly the same. It solves the "it worked on my machine" problem that is sadly all-too familiar to many developers. Many companies have software infrastructure that involves stacking up many instances of the same application in an array of containers, all processing the same source of input. Whether that be a queue, user requests, or whatever. The environment that hosts them can even be configured to scale up or down the number of active containers depending on demand.

Serverless

This might be familiar to .NET developers as Azure Functions or AWS Lambdas. This is code that isn't deployed to a traditional web server, such as IIS, but rather as a single function that exists in isolation out on a cloud hosting environment. This allows both the same sorts of automatic scaling as is possible with containers, but also micro-level optimizations, where more money can be spent on more critical functions and less money on functions where the output can take longer to complete.

In both of these technologies, there is a great deal of utilization of concurrent processing; i.e. multiple instances of the same functionality working at the same time on the same input source. It's like .NET's Async features, but applied to a much larger scope.

The problem with any sort of asyncronous operations tends to occur with shared resources, whether that's in-memory state or a literal shared physical or software-based external resource.

Functional Programming operates without state, so there can be no shared state between threads, containers or serverless functions.

When implemented correctly, following the Functional paradigm makes it much easier to implement these much-in-demand technological features, but without giving rise to any unexpected behavior in production.

Reduce Code Noise

In audio processing they have a concept called the *Signal-to-Noise* Ratio. This is a measure of how clear a recording is, based on the ratio of the volume level of the signal (the thing you want to listen to) to the noise (hiss, crackle, rumble or whatever in the background).

In coding, the *signal* is the business logic of a block of code - the thing it is actually trying to accomplish. The *what* of the code.

The *noise* is all of the boilerplate code that must be written in order to accomplish the goal. The For-loop definition, If statements, that sort of thing.

Compared to Procedural code, neat, concise Functional programming has significantly less boilerplate, and so has a much better Signal-to-noise ratio.

This isn't just a benefit to developers. Robust, easier to maintain codebases means the business needs to spend less money on maintainance and enhancements.

The Best Places to Use Functional Programming

FP can do absolutely anything that any other paradigm can, but there are areas where it’s strongest and most beneficial - and other areas where it might be necessary to compromise and incorporate some Object Orientation features, or slightly bend the rules of the Functional Paradigm. In .NET at least, compromises necessarily have to be made, because any base classes or add-on libraries all tend to be written following the Object-Oriented paradigm. This doesn’t apply to Pure Functional languages.

Functional Programming is good where there’s a high degree of predictability. For example, data processing modules - functions that convert data from one form to another. Business logic classes that handle data from the user or database, then pass it on to be rendered elsewhere. Stuff like that.

The stateless nature of Functional Programming makes it a great enabler of concurrent systems - like heavily asynchronous codebases, or places where several processors are listening concurrently to the same input queue. When there is no shared state, it’s just about impossible to get resource contention issues.

If your team is looking into using Serverless applications - such as Azure Functions, then Functional Programming enables that nicely for most of the same reasons.

It’s worth considering Functional Programming for highly business-critical systems because the paradigm works in a way that makes it easier to produce code that’s less error-prone and more robust than applications coded with the Object-Oriented paradigm. If it’s incredibly important that the system should stay up, and not have a crash and burn (i.e. terminate unexpectedly) in the event of an unhandled exception or invalid input, then Functional Programming might be the best choice.

Where You Should Consider Using Other Paradigms?

You don’t have to ever do any such thing, of course. Functional can do anything, but there are a few areas where it might be worth looking around for other paradigms - purely in a C# context. And it’s also worth mentioning again that C# is a hybrid language, so many paradigms can quite happily sit side by side, next to each other, depending on the needs of the developer. I know which I prefer, of course!

Interactions with external entities is one area for consideration. I/O, user input, 3rd party applications, web APIs, that sort of thing. There’s no way to make those pure functions (i.e. without side effects), so compromise is necessary. The same goes for 3rd party modules imported from Nuget packages. There are even a few older Microsoft libraries that are simply impossible to work with Functionally. This is still true in .NET Core. Have a look at the `SmtpClient` or `MailMessage` classes in .NET if you want to see a concrete example.

In the C# world, if performance is your projects only, overwhelming concern, trumping all others, even readability and modularity, then following the Functional paradigm might not be the best idea. There’s nothing necessarily inherently poor in the performance of Functional C# code, but it’s not necessarily going to be the most utterly performant solution either.

I would argue that the benefits of Functional Programming far outweigh any minor loss of performance, and these days, most of the time it’s very easy to chuck a bit more hardware (virtual or physical, as appropriate) at the app - and this is likely to be an order of magnitude cheaper than the cost of additional developer time that would otherwise be required to develop, test, debug and maintain the codebase that is written in imparative-style code. This changes if - for example - you’re working on code to be placed on a mobile device of some sort, where performance is critical, because memory is limited and can’t be updated.

How Far Can We Take This?

Unfortunately it simply isn’t possible to implement the entirety of the Functional paradigm in C#. There are all sorts of reasons for that, including the need for backwards compatability in the language and limitations imposed on what remains a strongly-typed language.

The intention of this book isn’t to show you how all of it can be done, but rather to show where the boundaries are between what is and isn’t possible. I’ll also be looking into what’s practical, especially with an eye to those of you maintaining a production codebase. This is ultimately a practical, pragmatic guide to Functional coding styles.

~~Monads~~ – Actually don’t worry about this yet

Monads are often thought to be the Functional horror story. Look on Wikipedia for definitions, and you’ll be presented with a strange letter soup containing Fs, Gs, arrows and more brackets than you’ll find under the shelves of your local library. The formal definitions are something I find - even now - utterly illegible. At the end of the day, I’m an engineer, not a mathematician.

Douglas Crockford once said that the curse of the Monad is that the moment you gain the ability to understand it, you lose the ability to explain it. So I won't. They might make their presence known somewhere in this book, however. Especially at unlikely times.

Don't worry, it'll be fine. We'll work though it all together. Trust me…

Summary

In this first exciting installment of *Functional Programming with C#*, our mighty, awe-inspiring hero - you - bravely learned just what exactly Functional Programming is, and why it's worth learning.

There was an initial, brief introduction to the important features of the Functional paradigm:

- Immutabilty
- Higher-Order Functions
- Prefer Expressions over Statements
- Referential Transparency
- Recursion
- Pattern Matching
- Stateless

There was a discussion of the areas Functional Programming is best used in, and where perhaps a discussion needs to be had regarding whether to use it in its pure form or not.

We also looked at the many, many benefits of writing applications using the Functional Paradigm.

In the next thrilling episode, we'll start looking at what you can do in C# right here, right now. No new 3rd party libraries or Visual Studio extension required. Just some honest-to-goodness out-of-the-box C# and a little ingenuity.

Come back just over the page to hear all about it. Same .NET time. Same .NET channel[16].

including Vanilla, and my personal favorite - Banana

They were Hero turtles when I was growing up in the UK in the 90s. I think the TV folks were trying to avoid the violent connertations of the word "ninja". They nevertheless still let us *see* our heroes regularly use sharp, bladed implements on their villains

Credit must be given to functional programming supremo Mark Seeman (https://blog.ploeh.dk/) for giving me these handy rules.

Because I made it up for the same of this example

Ok, art bods, I know there are actually about 12, but that's more than I need for this metaphor to work

A talk on that subject is avialable here: *https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell*

Never do this particular example in production code. I'm keeping it simple for the purposes of explanation

For some reason Julie Andrews won't return my calls to discuss an updated .NET version of one of her famous songs

with a little creative liberty taken

e.g. "I can't get this Freging code to work!"

available to read online for free at *http://www.learnyouahaskell.com*. Tell 'em I sent you

Although I **am** learning Latin. Insipiens sum. Huiusmodi res est ioci facio.

especially F# guru Ian Russell, who helped with the F# content in this book. Thanks, Ian!

at least that’s what we tell our managers

virtual or otherwise

or book, if we’re being picky

Part I. What Are We Already Doing?

Believe it or not, there’s a good chance you’ve been doing functional coding to a greater or lesser extent if you’ve been coding with .NET for any amount of time.

This first part is all going to be about showing you just how much of your everyday code is - or could easily be - functional. All of this without installing a single library beyond those provided by Microsoft. No tricky theory either.

Think of this section as the shores of our journey to the wet, dark, mysterious depths of functional programming. You’re still on dry land, and everything should feel vaguely familiar.

Part Two is where we’ll start looking more into functional concepts. If you’re finding Part One too easy as you’re reading, then feel free to skip ahead to Part Two.

Chapter 2. What Can We Do Already?

Some of the code and concepts discussed in this chapter may seem trivial to some, but bear with me. I don’t want to introduce too much too soon. More experienced developers might like to skip ahead to Chapter 3, in which I talk about the more recent developments in C# for Functional progammers, or Chapter 4 where I demonstrate some novel ways to use features you might already be familiar with to achive some Functional features.

In this chapter, I’m going to look at the Functional Programming features that are possible in just about every C# codebase in use in production today.

I’m going to assume at least .NET Framework 3.5, and with some minor alterations, all of the code samples provided in this chapter will work in that environment. Even if you work in a more recent version of .NET, but are unfamiliar with Functional Programming, I still recommend reading this chapter, as it should give you a decent starting point in programming with the Functional Paradigm.

Those of you familiar already with Functional code, and just want to see what’s available in the latest versions of .NET, it might be best to skip ahead to the next chapter.

Getting Started

Functional Programming is easy, really it is! Despite what many people think, it’s easier to learn than Object-Oriented progrmanning. There are fewer concepts to learn, and actually less to get your head around.

If you don’t believe me, try explaining Polymorphism to a non-technical member of your family! Those of us that are comfortable with Object Orientation have often been doing it so long that we’ve forgotten how hard it may have been to get our heads around it at the beginning.

Functional programming isn’t hard to understand at all, just different. I’ve spoken to plenty of students coming out of university that embrace it with enthusiasm. So, if *they* can manage it…

The myth does seem to persist though, that to get into Functional Programming, there’s a whole load of stuff that needs learning first. What if I told you though, that if you’ve been doing C# for any length of time, you’ve already most likely been writing Functional code for a while? Let me show you what I mean…

Your First Functional Code

Before we start with some functional code, let’s look at a bit of non-functional. A style you most likely learned somewhere very near the beginning of your C# career.

A Non-Functional Film Query

In my quick, made-up example, I’m getting a list of all films from my imaginary data store and creating a new list, copied from the first, but only those items in the Action genre[1]

```
public IEnumerable<Film> GetFilmsByGenre(string
{
 var allFilms = GetAllFilms();
 var chosenFilms = new List<Film>();

 foreach (var f in allFilms)
 {
  if (f.Genre == genre)
  {
      chosenFilms.Add((f));
  }
 }

 return chosenFilms;

}

var actionFilms = GetFilmsByGenre("Action");
```

What’s wrong with this code? At the very least, it’s not very elegant. That’s a lot we’ve written to do something fairly simple.

We’ve also instantiated a new object that’s going to stay in scope for as long as this function is running. If there’s nothing more to the whole function than this, then there’s not much to worry about. But, what if this were just a short excerpt from a very long function? In that instance, the allFilms and actionFilms variables would both remain in scope, and thus in memory all that time, even if they aren’t in use.

There may not necessarily be copies of all of the data held within the item that’s being replicated, depending on whether it’s a class, a struct or whatever else. At the very least though, there’s a duplicate set of references being held unnecessarily in memory for as long as both items are in scope. That's still more memory than we strictly need to hold.

We're also forcing the order of operations. We've specified when to loop, when to add, etc. Both where and when each step should be carried out. If there were any intermediate steps in the data transformations to be carried out, we'd be specifying them too, and holding them in yet more potentially long-life variables.

I could solve a few problems with a `yield` return like this:

```
public IEnumerable<Film> GetFilmsByGenre(string g
{
 var allFilms = GetAllFilms();

 foreach (var f in allFilms)
 {
  if (f.Genre == genre)
  {
      yield return f;
  }
 }
}

var actionFilms = GetFilmsByGenre("Action");
```

This hasn't done more than shave a few lines off, however.

What if there were a more optimal order of operations than the one we've decided on? What if a later bit of code actually meant that we don't end up returning the contents of actionFilms? We'd have done the work unnecessarily.

This is the eternal problem of procedural code. Everything has to be spelled out. One of our major aims with Functional Programming is to move away from all that. Stop being so specific about every little thing. Relax a little, and embrace declaritive code.

A Functional Film Query

So, what would that code sample above look like written in a Functional style? I'd hope many of you might already guess at how you would re-write it.

```
public IEnumerable<Film> GetFilmsByGenre(IEnumera
 source.Where(x => x.Genre == genre);

 var allFilms = GetAllFilms();
 var actionFilms = GetFilmsByGenre(allFilms, "Act
```

If anyone at this point is saying “isn’t that just LINQ?”, then yes. Yes, it is. I’ll let you all in on a little secret - LINQ follows the Functional paradigm.

Just quickly, for anyone that’s not yet familiar with the awesomeness of LINQ. It’s a library that’s been part of C# since the early days, and provides a rich set of functions for filtering, altering and extending collections of data. Functions like `Select`, `Where` and `All` are from LINQ and commonly used around the world.

Think back for a moment to the list of features of Functional Programming, and see how many LINQ implements…

- Higher-order Functions - The lambda expressions passed to LINQ functions are all functions, being passed in as parameter variables.
- Immutability - LINQ doesn’t change the source array, it returns a new `Enumerable` based on the old one.
- Expressions instead of Statements - We’ve eliminated the use of a `ForEach` and an `If`
- Referential Transparency - The Lambda Expression I’ve written here does actually conform to Referential Transparency (I.e. “no side effects”), though there’s nothing enforcing that. I could easily have referenced a string variable outside the Lambda. By requiring that the source data be passed in as a parameter, I’m also making it easier to test without requiring the creation & setup of a Mock of some kind to

represent the data store connection. Everything the function needs is provided by its own parameters.

The iteration could well be done by recursion too, for all I know, but I have no idea what the source code of the Where function looks like. In the absence of evidence to the contrary, I'm just going to go on believing that it does.

This tiny little one-line code sample is a perfect example of the Functional approach in many ways. We're passing around functions to perform operations against a collection of data, creating a new collection based on the old one.

What we've ended up with by following the Functional paradigm is something more concise, easier to read and therefore far easier to maintain.

Results-Oriented Programming

A common feature of Functional code is that it focuses much more heavily on the end result, rather than on the process of getting there. An entirely Procedural method of building a complex object would be to instantiate it empty at the beginning of the code block, then fill in each property as we go along.

Something like this:

```
var sourceData = GetSourceData();
var obj = new ComplexCustomObject();

obj.PropertyA = sourceData.Something + sourceData
obj.PropertyB = sourceData.Ping * sourceData.Pon

if(sourceData.AlternateTuesday)
{
  obj.PropertyC = sourceData.CaptainKirk;
  obj.PropertyD = sourceData.MrSpock;
}
else
{
   obj.PropertyC = sourceData.CaptainPicard;
   obj.PropertyD = sourceData.NumberOne;
}

return obj;
```

The problem with this approach is that it’s very open to abuse. This silly little imaginary codeblock I’ve created here is short and easy to maintain. What often happens with production code however, is that the code can end up becoming incredibly long, with multiple data sources that all have to be pre-processed, joined, re-processed, etc. You can end up with long blocks of If-statements nested in If-statements, to the point that the code starts resembling the shape of a Family Tree.

For each nested If-statement, the complexity effectively doubles. This is especially true if there are multiple return statements scattered around the codebase. The risk increases of inadvertently ending up with a Null or some other unexpected value if the increasingtly complex codebase isn't thought through in detail. Functional Programming discourages structures like this, and isn't prone to this level of complexity, or of the potential unexpected consequences.

In our code sample above, we have PropertyC and PropertyD defined in 2 different places. It's not too hard to work with here, but I've seen examples where the same property is defined in around half a dozen places across multiple classes and sub-classes[2].

I don't know whether you've ever had to work with code like this? It's happened to me an awful lot.

These sorts of large, unweildy codebases only ever get harder to work with over time. With each addition, the actual speed at which the developers can do the work goes down, and the business can end up getting frustrated because they don't understand why their "simple" update is taking so long.

Functional code should ideally be written into small, concise blocks, focusing entirely on the end product. The expressions it prefers are modelled on mathematical working, so you really want to write it like small formulas, each precisely defining a value and all of the variables that make it up. There shouldn’t be any hunting up and down the codebase to work out where a value comes from.

Something like this:

```
function ComplexCustomObject MakeObject(SourceDat
    new ComplexCustomObject
    {
       PropertyA = source.Something + source.Some
       PropertyB = source.Ping * source.Pong,
       PropertyC = source.AlternateTuesday
                     ? source.CaptainKirk
                     : source.CaptainPicard,
       PropertyD = source.AlternateTuesday
                     ? source.MrSpock,
                     : source.NumberOne
    };
```

I know I’m now repeating the AlternateTuesday flag, but it means that all of the variables that determine a returned property are defined in a single place. It makes it much simpler to work with in the future.

In the event that a property is so complicated that it will either need multiple lines of code, or a series of Linq operations that takes up a lot of space, then I’d create a break-out function to contain that complex logic. I’d still have my central, result-based return at the heart of it all, though.

A few words about Enumerables

I sometimes think Enumerables are one of the most under-used and least understood features of C#. An Enumerable is the most abstract representation of a collection of data - so abstract that it doesn't contain any data itself, it's actually just a description held in memory of how to go about getting the data. An Enumerable doesn't even know how many items there are available until it iterates through everything - all it knows is where the current item is, and how to iterate to the next.

This is called *Lazy Evaluation* or *deferred Execution*. Being lazy is a good thing in development. Don't let anyone tell you otherwise[3].

In fact, you can even write your own entire customised behaviour for an Enumerable if you wanted. Under the surface, there's an object called an Enumerator. Interacting with that can be used to either get the current item, or iterate on to the next. You can't use it to determine the length of the list, and the iteration only works in a single direction.

Have a look at this code sample:

First a set of simple logging functions that pop a message in a List of strings:

```
IList<string> c = new List<string>();


public int DoSomethingOne(int x)
{
        c.Add(DateTime.Now + " - DoSomethingOne (
        return x;
}

public int DoSomethingTwo(int x)
{
        c.Add(DateTime.Now + " - DoSomethingTwo (
        return x;
}

public int DoSomethingThree(int x)
{
        c.Add(DateTime.Now + " - DoSomethingThree
        return x;
}
```

Then a bit of code that calls each of those “DoSomething” functions in turn with different data.

```
var input = new[]
{
        75,
        22,
        36
};

var output = input.Select(x => DoSomethingOne(x)
         .Select(x => DoSomethingTwo(x))
         .Select(x => DoSomethingThree(x))
         .ToArray();
```

What do you think the order of operations is? You might think that the runtime would take the original input array, apply DoSomethingOne to all 3 elements to create a second array, then again with all three elements into DoSomethingTwo, and so on.

If I were to examine the content of that List of strings, I'd find something like this:

```
18/08/1982 11:24:00 - DoSomethingOne(75)
18/08/1982 11:24:01 - DoSomethingTwo(75)
18/08/1982 11:24:02 - DoSomethingThree(75)
18/08/1982 11:24:03 - DoSomethingOne(22)
18/08/1982 11:24:04 - DoSomethingTwo(22)
18/08/1982 11:24:05 - DoSomethingThree(22)
18/08/1982 11:24:06 - DoSomethingOne(36)
18/08/1982 11:24:07 - DoSomethingTwo(36)
18/08/1982 11:24:08 - DoSomethingThree(36)
```

It’s almost the exact same as you might get if you were running this through a `For` / `ForEach` loop, but we’ve effectively handed over control of the order of operations to the runtime. We’re not concerned with the nitty-gritty of temporary holding variables, what goes where and when. Instead we’re just describing the operations we want, and expecting a single answer back at the end.

It might not always look exactly like that, it depends on what the code that calls it looks like. But the intent always remains, that Enumerables only actually produce their data at the precise moment it’s needed. It doesn’t matter where they’re defined, it’s when they’re *used* that makes a difference.

Using Enumerables instead of solid arrays, we’ve actually managed to implement some of the behaviors we need to write Declarative code.

Incredibly, the log file I wrote above would still look the same if I were to re-write the code like this:

```
var input = new[]
{
    1,
    2,
    3
};
```

```
var temp1 = input.Select(x => DoSomethingOne(x)
var temp2 = input.Select(x => DoSomethingTwo(x)
var finalAnswer = input.Select(x => DoSomething
```

temp1, temp2 and finalAnswer are all Enumerables, and none of them will contain any data until iterated.

Here’s an experiment for you to try. Write some code like this sample. Don’t copy it exactly, maybe something simpler like a series of selects amending an integer value somehow. Put a break point in and move the operation pointer on until final answer has been passed, then hover over finalAnswer in Visual Studio. What you’ll most likely find is that it can’t display any data to you, even though the line has been passed. That’s beause it hasn’t actually performed any of the operations yet.

Things would change if I did something like this:

```
var input = new[]
{
    1,
    2,
    3
};

var temp1 = input.Select(x => DoSomethingOne(x)
var temp2 = input.Select(x => DoSomethingTwo(x)
var finalAnswer = input.Select(x => DoSomething
```

Because I’m specifically now calling `ToArray()` to force an enumeration of each intermediate step, then we really will call DoSomethingOne for each item in input before moving onto the next stop.

The log file would look something like this now:

```
18/08/1982 11:24:00 - DoSomethingOne(75)
18/08/1982 11:24:01 - DoSomethingOne(22)
18/08/1982 11:24:02 - DoSomethingOne(36)
18/08/1982 11:24:03 - DoSomethingTwo(75)
18/08/1982 11:24:04 - DoSomethingTwo(22)
18/08/1982 11:24:05 - DoSomethingTwo(36)
18/08/1982 11:24:06 - DoSomethingThree(75)

18/08/1982 11:24:07 - DoSomethingThree(22)
18/08/1982 11:24:08 - DoSomethingThree(36)
```

For this reason, I nearly always advocate for waiting as long as possible before using `ToArray()` or `ToList()` [4], because this way we can leave the operations unperformed for as long as possible. And potentially even never performed if later logic prevents the enumeration from occurring at all.

There are some exceptions. Either for performance, or for avoiding multiple iterations. While the Enumerable remains un-enumerated it doesn’t have any data, but the operation itself remains in memory. If you pile too many of them on top of each other - especially if you start performing recursive operations, then you might find that you fill up far too much memory and performance takes a hit, and possibly even end up with a stack overflow.

Prefer Expressions to Statements

In the rest of this chapter, I’m going to give more examples of how Linq can be used more effectively to avoid the need to use statements like If, Where, For, etc. or to mutate state (i.e. change the value of a variable).

There will be cases that aren’t possible, or aren’t ideal. But, that’s what the rest of this book is for.

The Humble Select

If you’ve read this far in the book, you’re most likely aware of Select statements, and how to use them. There are a few features though, that most people I speak to don’t seem to be aware of, and they’re all things that can be used to make our code a little more functional.

The first thing was something I’ve already shown in the previous section - you can chain them. Either as a series of Select function calls - literally one after the other, or in a single code line; or else you can store the results of each Select in a different local variable. Functionally these two approaches are identical. It doesn’t even matter if you call ToArray after each one. So long as you don’t modify any resulting arrays or the object contained within them, you’re following the Functional paradigm.

The important thing is to get away from is the Imperative practice of defining a List, looping through the source objects with a ForEach and then adding each new item to the List. This is long-winded, harder to read, and honestly quite tedious. Why do things the hard way? Just use a nice, simple Select statement.

Passing working values via tuples

Tuples were introduced in C#7. Nuget packages do exist to allow some of the older versions of C# to use them too. They’re basically a way to throw together a quick-and-dirty collection of properties, without having to create and maintain a class.

If you’ve got a few properties you want to hold onto for a minute in one place, then dispose of immediately, Tuples are great for that.

If you have multiple objects you want to pass between Selects, or multiple items you want to pass in or out of one, then you can use a Tuple.

```
   var filmIds = new[]
   {
       4665,
       6718,
       7101
   };

   var filmsWithCast = filmIds.Select(x => (
       film: GetFilm(x),
       castList: GetCastList(x)
   ));

  var renderedFilmDetails = filmsWithCast.Select(x
                @$"
  Title: {x.film.Title}
  Director: {x.film.Director}
  Cast: {string.Join(", ", x.castList)}
  ".Trim());
```

In my example, above, I use a Tuple to pair up data from two look-up functions for each given film Id, meaning I can run a subsequent Select to simplify the pair of objects into a single return value.

Iterator value is required

Som what if you’re Select-ing an Enumerable into a new form, and you need the iterator as part of the transformation? Something like this:

```
var films = GetAllFilmsForDirector("Jean-Pierre
                                   .OrderByDescendin

var i = 1;

Console.WriteLine("The films of visionary French
Console.WriteLine("Jean-Pierre Jeunet in descend
Console.WriteLine(" of financial success are as

foreach (var f in films)
{
    Console.WriteLine($"{i} - {f.Title}");
    i++;
}

Console.WriteLine("But his best by far is Amelie
```

We can use a feature of Select statements that surprisingly few people know about - that it has an override that allows us access to the iterator as part of the Select. All you have to do is provide a Lambda expression with 2 parameters, the second being an integer which represents the index position of the current item.

This is how our functional version of the code looks:

```
var films = GetAllFilmsForDirector("Jean-Pierre
                                .OrderByDescending(x =

Console.WriteLine("The films of visionary French
Console.WriteLine("Jean-Pierre Jeunet in descend
Console.WriteLine(" of financial success are as

var formattedFilms = films.Select((x, i) => $"{i
Console.WriteLine(string.Join(Environment.NewLin

Console.WriteLine("But his best by far is Amelie
```

Using these techniques, there's nearly no circumstance that could exist where you need to use `ForEach` loop with a List. Thanks to C#'s support for the Functional paradigm, there are nearly always Declaritive methods available to solve problems.

The two different methods of getting the "i" index position variable are a great example of Imperative vs Delarative code. The Imperative, Object-Oriented method has the developer manually creating a variable to hold the value of i, and also explicitly set the place for the variable to be incremented. The declaritive code isn't concerned with where the variable is defined, or in how each index value is determined.

N.b - Notice that I used `string.Join` to link the strings together. This is not only another one of those hidden gems of the C# language, but it's also an example of Aggregation, that is - converting a list of things into a single thing. That's what we'll walk through in the next few sections.

No Starting Array

The last trick for getting the value of i for each iteration is great if there's an array - or collection of some other kind - available in the first place. What if there isn't an array? What if you need to arbitrarily iterate for a set number of times?

These are the situations - somewhat rare - where you need a good, old-fashioned `For` loop instead of a `ForEach`. How do you create an array from nothing?

Your two best friends in this case are two static methods - `Enumerable.Range` and `Enumerable.Repeat`.

Range creates an array from a starting integer value, and requires you to tell it how many elements the array you want should have. It then creates an array of integers based on those specifications.

For example:

```
var a = Enumerable.Range(8, 5);
var s = string.Join(", ", a);
// s = "8, 9, 10, 11, 12"
// That's 5 elements, each one higher than the la
// starting with 8.
```

Having whipped up an array, we can then apply LINQ operations to get our final result. Let’s imagine I was preparing a description of the 9 times table for one of my daughters[5].

```
var nineTimesTable = Enumerable.Range(1,10)
 .Select(x => x + " times 9 is " + (x * 9));

var message = string.Join("\r\n", nineTimesTable
```

Here’s another example for you, what if I wanted to get all of the values from a grid of some kind, where an x and y value are required to get each value. I’ll imagine there’s a grid repository of some kind that I can use to get values.

Imagining that the grid is a 5x5, this is how I’d get every value:

```
var coords = Enumerable.Range(1, 5)
          .SelectMany(x => Enumerable.Range(1, 5)
                    .Select(y => (X: x, Y: y))
);

var values = coords.Select(x => this.gridRepo.Get
```

The first line here is generating an array of integers with the values `[1, 2, 3, 4, 5]`. I then use another `Select` to convert each of these integers into another array using another call to `Enumerable.Range`. This meant I now have an array of 5 elements, each of which was iteself an array of 5 integers. Using a Select on that nested array, I converted each of those sub-elements into a tuple which took one value from the parent array (x) and one from the sub-array (y). SelectMany is used to flatten the whole thing out to a simple list of all of the possible coordinates, which would look something like this: `(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (2, 2)`…and so on.

Values can be obtained by Selecting this array of coordinates into a set of calls to the repository's GetVal function, passing in the values of X and Y from the tuple of coordinates I created on the previous line.

Another situation we might be in is needing the same starting value in each case, but needing to transform it in different ways, depending on the position within the array. This is where `Enumerable.Repeat` comes in.

`Enumerable.Repeat` creates where each value is exactly the same, and you can specify exactly how many repeat elements you want.

You can't use `Enumerable.Range` to count backwards. What if we wanted to do the previous example, but start at (5,5) and move backwards to (1,1). Here's an example of how you'd do it:

```
var gridCoords = Enumerable.Repeat(5, 5).Select(
          .SelectMany(x => Enumerable.Repeat(5, 5)
                  .Select((y, i) => (x, y - i))
);

var values = coords.Select(x => this.gridRepo.Get
```

This looks a lot more complicated, but it isn't really. What I've done is swap out the `Enumerable.Range` call for a 2-step operation.

First a call to `Enumerable.Repeat` which is repeating the integer value of 5 - 5 times. This results in an array like this: `[5, 5, 5, 5, 5]`.

Having done that, I'm then selecting using the overloaded version of `Select` which includes the value of i, then deducting that i value from the current value in the array. This means that in the first iteration, the return value is the current value i the array (5) minus the value of i (0 for the first iteration), this gives simply 5 back. In the next iteration, the value of i is 1, so 5-1 means 4 is returned. And so on.

At the end of it we get back an array that looks something like this: `(5, 5), (5, 4), (5, 3), (5, 2), (5, 1), (4, 5), (4, 4)` …etc.

There are ways to take this further still, but for this chapter I’m sticking to the relatively simple cases, ones that don’t require hacking around with C#. This is all out-of-the-box functionality that anyone can use right away.

Many to One - The subtle art of Aggregation

We’ve looked at loops for converting one thing into another, X items in → X new items out. That sort’ve thing. There’s another use case for loops that I’d like to cover - reducing many items into a single value.

This could be making a total count, calculating Averages, Means or other statistical data, or other more complex aggregations.

In Procedural code, we’d have a loop, a state tracking value and inside the loop we’d update the state constantly, based on each item from our array. Here’s a very simple example of what I’m talking about:

```
var total = 0;
foreach(var x in listOfIntegers)
{
{
  total += x;
}
```

There’s actually an in-built Linq method for doing this:

```
    var total = listOfIntegers.Sum();
```

There really shouldn’t ever be a need to do this sort of operation “long-hand”. Even if we’re creating the sum of a particular property from an array of Objects, Linq still has us covered:

```
    var films = GetAllFilmsForDirector("Alfred Hitch
    var totalRevenue = films.Sum(x => x.BoxOfficeRev
```

There’s another function for calculating Means in the same manner called Average. There’s nothing for calculating Median, so far as I’m aware.

I could calculate the Median with a quick bit of functional style code, however. It would look like this:

```
  var numbers = new [] {
      83,
      27,
      11,
      98
   };

   bool IsEvenNumber(int number) => number % 2 == (

   var sortedList = numbers.OrderBy(x => x).ToArray
   var sortedListCount = sortedList.Count();

   var median = IsEvenNumber(sortedList.Count())
                                    ? sortedList.Skip
                                    : sortedList.Skip

  // median = 55.
```

There are more complex aggregations that are required sometimes. What if we wanted - for example - a sum of two different values from an Enumerable of complex objects?

Procedural code might look like this:

```
   var films = GetAllFilmsForDirector("Christopher

   var totalBudget = 0.0M;
   var totalRevenue = 0.0M;

   foreach (var f in films)
   {
       totalBudget += f.Budget;
       totalRevenue += f.BoxOfficeRevenue;
   }
```

We could use two separate Sum function calls, but then we'd be iterating twice through the Enumerable, hardly an efficient way to get our information. Instead, we can use another strangely little-known feature of Linq - the aggregate function. This consists of the following components:

- Seed - a starting value for the final value.
- An Aggregator function, this has two parameters - the current item from the Enumerable we're aggregating down, and the current running total.

The seed doesn't have to be a primitive type, like an integer or whatever, it can just as easily be a complex object. In order to re-write the code sample, above, in a Functional style, however, we just need a simple Tuple.

```
    var films = GetAllFilmsForDirector("Christopher

    var (totalBudget, totalRevenue) = films.Aggregat
           (Budget: 0.0M, Revenue: 0.0M),
           (runningTotals, x) => (
                           runningTotals.Budget + x
                           runningTotals.Revenue + x
                   )
   );
```

In the right place, Aggregate is an incredibly powerful feature of C#, and one worth taking the time to explore and understand properly.

It’s also an example of another concept important to Functional Programming - recursion.

Customised Iteration Behavior

Recursion sits at the back of a lot of Functional versions of Iteration. For the benefit of anyone that doesn’t know, it’s a function that calls itself repeatedly until some condition or other is met.

It’s a very powerful technique, but has some limitations to bear in mind in C#. The most important two being:

- If developed improperly, it can lead to infinite loops, which will literally run until the user terminates the application, or all available space on the stack is consumed. As Treguard, the legendary Dungeon Master of the popular British Fantasy RPG gameshow *Knightmare* would put it: “Oooh, Nasty”[6].
- In C# they tend to be consume a lot of memory compared to other forms of iteration. There are ways around this, but that’s a topic for another chapter.

I have a lot more to say about recursion, and we’ll get to that shortly, but this for the purposes of this chapter, I’ll give the simplest example I can think of.

Let’s say that you want to iterate through an Enumerable but you don’t know how long for. Let’s say you have a list of delta values for an integer (i.e. the amount to add or subtract each time) and you want to find out how many steps it is until you get from the starting value (whatever that might be) to 0.

You could quite easily get the final value with an Aggregate call, but we don’t want the final value. We’re interested in all of the intermediate values, and we want to stop prematurely through the iteration. This is a simple arithmetic operation, but if complex objects were involved in a real-world scenario, there might be a significant performance saving from the ability to terminate the process early.

In Procedural code, you’d probably write something like this:

```
var deltas = GetDeltas().ToArray();
var startingValue = 10;
var currentValue = startingValue;
var i = -1;

foreach(var d in deltas)
{
   if(currentValue == 0)
   if(currentValue == 0)
   {
     break;
   }
   i++;
   currentValue = startingValue + d;

}

return i;
```

In this example I’m returning -1 to say that the starting value is already the one we’re looking for, otherwise I’m returning the zero-based index of the array that resulted in 0 being reached.

This is how I’d do it recursively:

```
var deltas = GetDeltas().ToArray();

int GetFirstPositionWithValueZero(int currentVal
    currentValue == 0
        ? i
        : GetFirstPositionWithValueZero(current

return GetFirstPositionWithValueZero(10);
```

This is Functional now, but it’s not really ideal. Nested functions have their place, but I don’t personally find the way it has to be used here as readable as the code could be. Delightfully recursive, but I think it could be made clearer.

The other major problem is that this won’t scale up well if the list of deltas is large. I’ll show you what I mean.

Let’s imagine there are only 3 values for the Deltas: 2, -12 & 9. In this case we’d expect our answer to come back as 1, because the second position (i.e. index=1) of the array resulted in a zero (10+2-12). We would also expect that the 9 will never be evaluated. That’s the efficiency saving we’re looking for from our code here.

What was actually happening with the recursive code, though.

First, it called GetFirstPositionWithValueZero with a current value of 10 (i.e. the starting value) and i was allowed to be the default of -1.

The body of the function is a ternary if statement. If zero has been reached, return i, otherwise call the function again but with updated values for current and i.

This is what’ll happen with the first delta (i.e. i=0, i.e. 2), so GetFirstPositionWithValueZero is called again with the current value now updated to 12 and i as 0.

The new value is not 0, so the second call to GetFirstPositionWithValueZero will call itself again, this time with the current value updated with delta[1] and i incremented to 1. delta[1] is -12, which would mean the third call results in a 0, which means that i can simply be returned.

Here's the problem though…

The third call got an answer, but the first two calls are still open in memory and stored on the stack. The third call returns 1, which is passed up a level to the second call to GetFirstPositionWithValueZero, which now also returns 1, and so on… Until finally the original first call to GetFirstPositionWithValueZero returns the 1.

If you want to see that a little graphically, imagine it looking something like this:

```
GetFirstPositionWithValueZero(10, -1)
   GetFirstPositionWithValueZero(12, 0)
      GetFirstPositionWithValueZero(0, 1)
      return 1;
   return 1;
return 1;
```

That's fine with 3 items in our array, but what if there are hundreds!

Recursion, as I’ve said, is a powerful tool, but it comes with a cost in C#. Purer Functional languages (including F#) have a feature called *Tail Call Optimised Recursion* which allows the use of recursion without this memory usage problem.

Tail Recursion is an important concept, and one I’m going to return to later in a whole chapter dedicated to it, so I’m not going to dwell on it in any further detail here.

As it stands, out-of-the-box C# doesn’t permit Tail Recursion, even though it’s available in the .NET Common Language Runtime (CLR). There are a few tricks we can try to make it available to us, but they’re a little too complex for this chapter, so I’ll talk about them at a later juncture.

For now, consider recursion as it’s described here, and keep in mind that you might want to be careful where and when you use it.

Immutability

There’s more to Functional Programming in C# than just Linq. Another important feature I’d like to discuss is Immutability (i.e. a variable may not change value once declared). To what extent is it possible in C#?

Firstly, there are some newer developments with regards to Immutability in C# 8 and upwards. See the next chapter for that. For this chapter, I’m restricting myself to what is true of just about any version of .NET.

To begin, let’s consider this little C# snippet:

```
public class ClassA
{
  public string PropA { get; set; }
  public int PropB { get; set; }
  public DateTime PropC { get; set; }
  public IEnumerable<double> PropD { get; set; }
  public IList<string> PropE { get; set; }
}
```

Is this immutable? It very much is not. Any of those properties can be replaced with new values via the setter. The IList also provides a set of functions that allows its underlying array to be added to or removed from.

We could make the setters private, meaning we’d have to instantiate the class via a detailed contructor:

```
public class ClassA
{
  public string PropA { get; private set; }
  public int PropB { get; private set; }
  public DateTime PropC { get; private set; }
  public IEnumerable<double> PropD { get; privat
```

```
    public IList<string> PropE { get; private set;

    public ClassA(string propA, int propB, DateTim
    {
      this.PropA = propA;
      this.PropB = propB;
      this.PropC = propC;
      this.PropD = propD;
      this.PropE = propE;
    }

  }
```

Is it immutable now? No, honestly it’s not. It’s true that you can’t outright replace any of the properties with new objects outside ClassA, which is great. The properties can be replaced inside the class, but the developer can ensure that no such code is ever added. You should hopefully have some sort of code review system to ensure that, as well.

PropA and PropC are fine - strings and DateTime are both immutable in C#. The int value of PropB is fine too - ints don’t have anything you can change except its value.

There are still several problems, however.

PropE is a List, which can still have values added, removed and replaced, even though we can't replace the entire object. If we didn't actually need to hold a mutable copy of PropE, we could easily replace it with an IEnumerable or IReadOnlyList.

The `IEnumerable<double>` value of PropD seems fine at first glance, but what if it was passed to the constructor as a `List<double` which is still referenced by that type in the outside world? It would still be possible to alter its contents that way.

There's also the possibility of introducing something like this:

```
public class ClassA
{
  public string PropA { get; private set; }
  public int PropB { get; private set; }
  public DateTime PropC { get; private set; }
  public IEnumerable<double> PropD { get; private
  public IList<string> PropE { get; private set;
  public SubClassB PropF { get; private set; }

  public ClassA(string propA, int propB, DateTime
  {
    this.PropA = propA;
    this.PropB = propB;
    this.PropC = propC;
    this.PropD = propD;
```

```
      this.PropE = propE;
      this.PropF = propF
    }

  }
```

All properties of PropF are also potentially going to be mutable - unless this same structure with private setters is followed there too.

What about classes from outside your codebase? What about Microsoft classes, or those from a 3rd party Nuget package? There's no way to enforce immutability.

Unfortunately there simply isn't any way to enforce universal immutability, not even in the most recent versions of C#. I would assume that for backwards compatibility reasons, there is never going to be.

It would be lovely to have a native C# method of ensuring immutability by default, but there isn't one - and isn't ever likely to be for reasons of backwards compatibility. My own solution is that when coding, I simply *pretend* that Immutability exists in the project, and never change any object. There's nothing in C# that provides any level of enforcement whatsoever, so you'd simply have to make a decision for yourself, or within your team, to act as if it does.

Putting it all Together - a Complete Functional Flow

I’ve talked a lot about some simple techniques you can use to make your code more functional right away. Now, I’d like to show a complete, if minute, application written to demonstrate an end-to-end functional process.

I’m going to write a very simple CSV parser. In my example, I want to read in the complete text of a CSV file containing data about the first few series of Doctor Who[7]. I want to read the data, parse it into a Plain Old C# Object (POCO, i.e. a class containing only data and no logic) and then aggregate it into a report which counts the number of episodes, and the number of episodes known to be lost for each season. [8]. I’m simplifying CSV parsing for the purposes of this example. I’m not worrying about quotes around string fields, commas in field values or any values requiring additional parsing. There are 3rd party libraries for all of that! I’m just proving a point.

This complete process represents a nice, typical functional flow. Take a single item, break it up into a list, apply list operations, then aggregate back down into a single value again.

This is the structure of my CSV file:

- [0] - Season Number. Integer value between 1 and 39. I’m running the risk of dating this book now, but there are 39 seasons to date.
- [1] - Story Name - a string field I don’t care about
- [2] - Writer - ditto
- [3] - Director - ditto
- [4] - Number of Episodes - in Doctor Who, all stories comprise between 1 and 14 episodes. Until 1989, all stories were multi-part serials.
- [5] - Number of Missing Episodes - the number of episodes of this serial not known to exist. Any non-zero number is too many for me, but such is life.

I want to end up with a report that has just these fields:

- Season Number
- Total Episodes
- Total Missing Episodes
- Percentage Missing

Let’s crack on with some code….

```
var text = File.ReadAllText(filePath);

// Split the string containing the whole contents
// file into an array where each line of the orig
// (i.e. each record) is an array element
var splitLines = text.Split(Environment.NewLine)
```

```
// Split each line into an array of fields, split
// source array by the ',' character.  Convert to
// for each access.
var splitLinesAndFields = splitLines.Select(x =>

// Convert each string array of fields into a dat
// parse any non-string fields into the correct t
// Not strictly necessary, based on the final agg
// that follows, but I believe in leaving behind
// extendible code
var parsedData = splitLinesAndFields.Select(x =>
{
   SeasonNumber = int.Parse(x[0]),
   StoryName = x[1],
   Writer = x[2],
   Director = x[3],
   NumberOfEpisodes = int.Parse(x[4]),
   NumberOfMissingEpisodes = int.Parse(x[5])
});

// group by SeasonNumber, this gives us an array
// objects for each season of the TV series
var groupedBySeason = parsedData.GroupBy(x => Sea

// Use a 3 field Tuple as the aggregate state:
// S (int) = the season number.  Not required fo
//                 the aggregation, but we need a
//                to pin each set of aggregated t
//                to a season
// NumEps (int) = the total number of episodes in
//                serials in the season
// NumMisEps (int) = The total number of missing
//                from the season
var aggregatedReportLines = groupedBySeason.Selec
    x.Aggregate((S: x.Key, NumEps: 0, NumMisEps:
      (acc, val) => (acc.S,
                       acc.NumEps + val.NumberOfE
                        acc.NumMisEps + val.Numbe
    )
);

// convert the Tuple-based results set to a prope
// object and add in the calculated field Percent
// not strictly necessary, but makes for more rea
// and extendible code
var report = aggregatedReportLines.Select(x => ne
{
   SeasonNumber = x.S,
   NumberOfEpisodes = x.NumEps,
   NumberOfMIssingEpisodes = x.NumMisEps,
   PercentageMissing = (x.NumMisEps/x.NumEps)*100
});

// format the report lines to a list of strings
var reportTextLines = report.Select(x => $"{x.Sea
$"{x.NumberofMissingEpisodes}\t{x.PercentageMissi

// join the lines into a large single string with
// characters between each line
var reportBody = string.Join(Environment.NewLine,
var reportHeader = "Season\tNo Episodes\tNo Missi

// the final report consists of the header, a new
var finalReport = $"{reportHeader}{Environment.Ne
```

In case you're curious, the results would look something like this (the "\t" characters are tabs, which make it a bit more readable):

```
Season    No Episodes    No Missing Eps    Percentag
1          42                9                   21
2          39                2                   5.1
3          45                28                  62
4          43                33                  76
5          40                18                  45
6          44                8                   18
7          25                0                   0
8          25                0                   0
9          26                0                   0

...
```

Note, I could have made the code sample more concise and written just about all of this together in one long, continuous fluent expression like this:

```
var reportTextLines = File.ReadAllText(filePath)
                      .Split(Environment.NewLine
                      .Select(x => x.Split(",").
                      .GroupBy(x => x[0])
                      .Select(x =>
    x.Aggregate((S: x.Key, NumEps: 0, NumMisEps:
      (acc, val) => (acc.S,
                      acc.NumEps + int.Parse(va
                       acc.NumMisEps + int.Parse
    )
)
.Select(x => $"{x.S}, {x.NumEps},{x.NumMisEps},{


var reportBody = string.Join(Environment.NewLine,
var reportHeader = "Season,No Episodes,No Missing

var finalReport = $"{reportHeader}{Environment.Ne
```

There’s nothing wrong with that sort of approach, but I like splitting it out into individual lines for a couple of reasons:

- The variable names provide some insight into what your code is doing. We're sort've semi-enforcing a form of code commenting.
- It's possible to inspect the intermediate variables, to see what's in them at each step. This makes debugging easier, because like I said in the previous chapter - it's like being able to look back on your working in a mathematics answer, to see which step it was that you went wrong on.

There isn't any ultimate functional difference, nothing that would be noticed by the end user, so which style you adopt is more a matter of personal taste. Write in whatever way it seems best to you. Do try and keep it readable, and easy for everyone to follow, though.

Taking it Further - Develop Your Functional Skills

Here's a challenge for you. If some or all of the techniques described to you here were new, then go off and have fun with them for a bit.

Challenge yourself to writing code with the following rules:

- Treat all variables as immutable - do not change any variable value once set. Basically treat everything as if it were a constant.
- None of the following statements are permitted - `If`, `For`, `ForEach`, `While`. `If` is acceptable only in a Ternary expression -

i.e. the single-line expression in the style: someBoolean ? valueOne : valueTwo.

- Where possible write as many functions as small, concise arrow functions (a.k.a. Lambda Expressions).

Either do this as part of your production code, or else go out and look for a code challenge site, something like The Advent of Code (https://adventofcode.com) or Project Euler (https://projecteuler.net). Something you can get your teeth into.

If you don't want to go through the bother of creating an entire solution for these exercises in Visual Studio, there's always LINQPad (*https://www.linqpad.net/*) for a quick and easy way to rattle off some C# code.

After you've got the hang of this, you'll be ready to move onto the next step. I hope you're having fun so far!

Summary

In this chapter, we looked at a variety of simple Linq-based techniques for writing Functional-style code immediately in any C# codebase using at least .NET Framework 3.5, because these features are ever-green and have been in place for all of those years in every subsequent version of .NET without needing to be updated or replaced.

We discussed the more advanced features of Select statements, some of the less well-known features of Linq and methods for Aggregating and Recursion.

In the next chapter, I'll look at some of the most recent developments in C# that can be used in more up-to-date codebases.

I'm more of an SF (or Sci-fi, if you prefer) fan, truth be told.

And in one example, a couple of definitions were also outside the codebase in Database Stored Procedures

Except your employer. They pay your bills. They hopefully send you birthday cards once a year too, if they're nice.

As a Functional programmer, and a believer in exposing the most abstract interface possible, I literally **never** use `ToList()`. Only ever `ToArray()`, even if `ToList` is every so slightly faster.

No, Sophie. It's not good enough to just use your fingers!

Check out the man himself [here (https://www.youtube.com/watch?v=OISR3al5Bnk)](https://www.youtube.com/watch?v=OISR3al5Bnk)

For those of you unacquainted, this is a British SF series that has been running on-and-off since 1963. It is, in my own opinion, the greatest TV series ever made. I’m not taking any arguments on that

Sad to say, the BBC junked many episodes of the series in the 1970s. If you have any of those, please do hand them back.

Chapter 3. Functional Coding in C# 7 and Beyond

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the third chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

I’m not sure when exactly the decision was made to make C# a hybrid Object-Oriented/Functional language. The very first foundation work was laid in C# 3. That was when features like Lambda Expressions and Anonymous Types were introduced, which later went on to form parts of Linq in .NET 3.5.

After that though, there wasn’t much new in terms of Functional features for quite some time. In fact, it wasn’t really until the release on C# 7 in

2017 that Functional Programming seemed to become relevant again to the C# team.

From C# 7 onwards, every version of C# has contained something new and exciting to do more Functional style coding, a trend that doesn’t currently show any signs of stopping!

In the last chapter, we looked at Functional features that could be implemented in just about any C# codebase likely to still be in use out in the wild. In this chapter, we’re going to throw away that assumption and look at all the features you can make use of if your codebase is allowed to use any of the very latest features - or at least those released since C# 7.

Tuples

Tuples were introduced in C#7. Nuget packages do exist to allow some of the older versions of C# to use them too. They’re basically a way to throw together a quick-and-dirty collection of properties, without having to create and maintain a class.

If you’ve got a few properties you want to hold onto for a minute in one place, then dispose of immediately, Tuples are great for that.

If you have multiple objects you want to pass between Selects, or multiple items you want to pass in or out of one, then you can use a Tuple.

This is an example of the sort of thing you might consider using Tuples for:

```
 var filmIds = new[]
 {
     4665,
     6718,
     7101
 };

// Turns each int element of the filmIds array
// into a tuple containing the film and cast list
// as separate properties

 var filmsWithCast = filmIds.Select(x => (
     film: GetFilm(x),
     castList: GetCastList(x)
 ));


// 'x' here is a tuple, and it's now being conver

var renderedFilmDetails = filmsWithCast.Select(x
 "Title: " + x.film.Title +
 "Director: " + x.film.Director +
 "Cast: " + string.Join(", ", x.castList));
```

In my example, above, I use a Tuple to pair up data from two look-up functions for each given film Id, meaning I can run a subsequent Select to simplify the pair of objects into a single return value.

Pattern Matching

Switch statements have been around for longer than just about any developers still working today. They have their uses, but they're quite limited in what can be done with them. Functional Programming has taken that concept and moved it up a few levels. That's what Pattern Matching is.

It was C# 7 that started to introduce this feature to the C# language, and it has been subsequently enhanced multiple times in later versions, and most likely there will be yet more features added in the future.

Pattern matching is an amazing way to save yourself an awful lot of work. To show you what I mean, I'll now show you a bit of Procedural code, and then how pattern matching is implemented in a few different versions of C#.

Procedural Bank Accounts

For our example, let's imagine one of the classic Object-Oriented worked examples - Bank Accounts. I'm going to create a set of bank account types, each with different rules for how to calculate the amount of interest. These aren't really based on real banking, they're straight out of my imagination.

These are my rules:

- A standard bank account calculates interest by multiplying the balance by the interest Rate for the account
- A premium bank account with a balance of 10,000 or less is a standard Bank account
- A premium bank account with a balance over 10,000 applies an interest rate augmented by a bonus additional rate
- A Millionaire's bank account, who owns so much money it's larger than the largest value a decimal can hold (It's a really, really big number - around 8*10^28, so they must be very wealthy indeed. Do you think they'd be willing to lend me a little, if I were to ask? I could do with a new pair of shoes). They have an overflow balance property to add in all that money they own that is over the max decimal value, that they can't store in the standard balance property like us plebs. They need the interest to be calculated based on both balances.
- A Monopoly player's bank account. They get an extra 200 for passing Go. I'm not implementing the "Go Direct to Jail" Logic, there are only so many hours in a day.

These are my classes:

```
public class StandardBankAccount
{
    public decimal Balance { get; set; }
    public decimal InterestRate { get; set; }
}

public class PremiumBankAccount : StandardBankAc
{
    public decimal BonusInterestRate { get; set;
}

public class MillionairesBankAccount : StandardB
{
    public decimal OverflowBalance { get; set; }
}

public class MonopolyPlayersBankAccount : Standa
{
    public decimal PassingGoBonus { get; set; }
}
```

The procedural approach to implementing the CalculateInterest feature for bank accounts - or as I think of it the “long-hand” approach, could possibly look like this:

```
public decimal CalculateNewBalance(StandardBankA
{
  // If real type of object is PremiumBankAccour
  if (sba.GetType() == typeof(PremiumBankAccount
  {
    // cast to correct type so we can access the
    var pba = (PremiumBankAccount)sba;
    if (pba.Balance > 10000)
    {
        return pba.Balance * (pba.InterestRate -
    }
  }

 // if real type of object is a Millionaire's ba
 if(sba.GetType() == typeof(MillionairesBankAcco
 {
   // cast to the correct type so we can get acc
   var mba = (MillionairesBankAccount)sba;
   return (mba.Balance * mba.InterestRate) +
            (mba.OverflowBalance * mba.InterestR
 }

   // if real type of object is a Monopoly Playe
 if(sba.GetType() == typeof(MonopolyPlayersBankA
 {
   // cast to the correct type so we can get acc
   var mba = (MonopolyPlayersBankAccount)sba;
   return (mba.Balance * mba.InterestRate) +
                mba.PassingGoBonus
    }

    // no special rules apply
    return sba.Balance * sba.InterestRate;
   }
```

As is typical with Procedural code, the above code isn't very concise, and might take a little bit of reading to understand its intent. It's also wide open to abuse if many more new rules are added once the system goes into production.

The Object-Oriented approach would be either to use an interface, or polymorphism - i.e. create an abstract base class with a virtual method for the CalculateNewBalance function. The issue with that is that the logic is now split over many places, rather than being contained in a single, easy-to-read function.

In the sections that follow, I'll show how each subsequent version of C# handled this problem.

Pattern Matching in C# 7

C# 7 gave us two different ways of solving this problem. The first was the new `is` operator - a much more convenient way of checking types than had previously been available. An `is` operator can also be used to automatically cast the source variable to the correct type.

Our updated source would look something like this:

```
public decimal CalculateNewBalance(StandardBankA
{
    // If real type of object is PremiumBankAcco
    if (sba is PremiumBankAccount pba)
    {
        if (pba.Balance > 10000)
        {
            return pba.Balance * (pba.InterestRa
        }
    }

    // if real type of object is a Millionaire's
    if(sba is MillionairesBankAccount mba)
    {
       return (mba.Balance * mba.InterestRate) +
               (mba.OverflowBalance * mba.Intere
    }

    // if real type of object is a Monopoly Play
    if(sba is MonopolyPlayersBankAccount mba)
    {
      return (mba.Balance * mba.InterestRate) +
               mba.PassingGoBonus;
    }
    // no special rules apply
    return sba.Balance * sba.InterestRate;
}
```

Note in the code sample above, that with the `is` operator, we can also automatically wrap the source variable into a new local variable of the correct type.

This isn’t bad, it’s a little more elegant, and we’ve saved ourselves a few redundant lines, but we could do better, and that’s where another feature of C# 7 comes in - type switching.

```
    public decimal CalculateNewBalance(StandardBankA
    {
      switch (sba)
      {
         case PremiumBankAccount pba when pba.Balanc
           return pba.Balance * (pba.InterestRate +
         case MillionairesBankAccount mba:
           return (mba.Balance * mba.InterestRate) +
                      (mba.OverflowBalance & mba.Inter
         case MonopolyPlayersBankAccount mba:
           return (mba.Balance * mba.InterestRate) +
         default:
           return sba.Balance * sba.InterestRate;
    }
  }
```

Pretty cool, right? Pattern Matching seems to be one of the most developed features of C# in recent years. As I'm about to show, every major version of C# since this has continued to add to it.

Pattern Matching in C# 8

Things moved up a notch in C# 8, pretty much the same concept, but with a new, updated matching syntax that more closely matches JSON, or a C# object initializer expression. Any number of clauses to properties or sub-properties of the object under examination can be put inside the curly braces, and the default case is now represented by the _ discard character.

```
  public decimal CalculateNewBalance(StandardBank
    sba switch
    {
      PremiumBankAccount { Balance: > 10000 } pba
                        (pba.InterestRate + pba.Bon
      MillionairesBankAccount mba => (mba.Balance
                    (mba.OverflowBalance & mba.Inte
      MonopolyPlayersBankAccount mba =>
                   (mba.Balance * mba.InterestRate)
      _ => sba.Balance * sba.InterestRate
    };
  }
```

Also, switch can now **also** be an expression, which you can use as the body of a small, single-purpose function with surprisingly rich functionality. This means it can also be stored in a Func delegate for potential passing around as a Higher-Order function.

This is an example using an old childhood game: Scissor, Paper Stone. Known in the US as Rock, Paper, Scissors and in Japan as Janken. I've created a `Func` delegate in the following example with the following rules:

1. Both players drawing the same = draw
2. Scissors beats paper
3. Paper beats stone
4. Stone beats scissors

This function is specifically determining what the result is from *my* perspective against my imaginary adversary.

```
public enum SPS
{
        Scissor,
        Paper,
        Stone
}

public enum GameResult
{
        Win,
        Lose,
        Draw
}

var calculateMatchResult = (SPS myMove, SPS thei
        (myMove, theirMove) switch
        {
                _ when myMove == theirMove => Gar
                ( SPS.Scissor, SPS.Paper) => Game
                ( SPS.Paper, SPS.Stone ) => GameI
                (SPS.Stone, SPS.Scissor) => GameI
                _ => GameResult.Lose
        };
```

Having stored it in a ‘Func<SPS,SPS>’ typed variable, I can pass it around to wherever needs it.

This can be as a parameter to a function, so that the functionality can be injected at run-time:

```
public string formatGames(IEnumerable<(SPS,SPS)>
string.Join("\r\n", game.Select((x, i) => "Game '
```

If I wanted to test the logic of this function without putting the actual logic into it, I could easily inject my own `Func` from a test method instead, so I wouldn't have to care what the real logic is - that can be tested in a dedicated test elsewhere.

It's another small way to make the structure even more useful.

Pattern Matching in C# 9

Nothing major added in C# 9, but a couple of nice little features. The `and` and `not` keywords from `is` expressions now work inside the curly braces of one of the patterns in the list, and it's not necessary any longer to have a local variable for a cast type if the properties of it aren't needed.

Although not ground breaking, this does continue to reduce the amount of necessary boilerplate code, and gives us an extra few pieces of more expressive syntax.

I've added a few more rules into the next example using these features. Now there are two categories of PremiumBankAccounts with different levels of special interest rates[1] and another bank account type for a Closed account, which shouldn't generate any interest.

```
public decimal CalculateNewBalance(StandardBankA
  sba switch
  {
    PremiumBankAccount { Balance: > 10000 and <=
                                   (pba.InterestRate + pba
    PremiumBankAccount { Balance: > 20000 } pba
                            (pba.InterestRate + pba.BonusIn
    MillionairesBankAccount mba => (mba.Balance
                           (mba.OverflowBalance + mba.Inter
    MonopolyPlayersBankAccount {CurrSquare: not
                          (mba.Balance * mba.InterestRate)
    ClosedBankAccount => 0,
    _ => sba.Balance * sba.InterestRate
  };
}
```

Not bad, is it?

Pattern Matching in C# 10

Like C# 9, C# 10 includes just the addition of another nice time-and-boilerplate-saving feature. A simple syntax for comparing the properties of sub-objects belonging to the type being examined.

```
public decimal CalculateNewBalance(StandardBankA
  sba switch
  {
    PremiumBankAccount { Balance: > 10000 and <=
                  pba.Balance * (pba.InterestRate -
    MillionairesBankAccount mba =>
                  (mba.Balance * mba.InterestRate)
    MonopolyPlayersBankAccount {CurrSquare: not 
                  (mba.Balance * mba.InterestRate)
    MonopolyPlayersBankAccount {Player.FirstName
                  (mba.Balance * mba.InterestRate)
    ClosedBankAccount => 0,
    _ => sba.Balance * sba.InterestRate
  };
```

In this slightly silly example, it’s now possible to exclude all “Simon"s from earning so much money in Monopoly when passing Go. Poor, old me.

I’d suggest taking another moment at this point to examine the function, above. Think just how much code would have to be written if it weren’t done as a Pattern Matching expression! As it is, it **technically** comprises just a single line of code. One…really long…line of code, with a whole ton of NewLines in to make it readable. Still, the point stands.

C# 11

C# 11 contains a new pattern matching feature which probably has a somewhat limited scope of useage, but will be devestatingly useful when something fits into that scope.

The .NET team have added in the ability to match based on the contents of an Enumerable and even to deconstruct elements from it into separate variables.

Let’s imagine we were creating a very simple text-based adventure game. These were a big thing when I was very young. Adventure games you played by typing in commands. Imagine something like Monkey Island, but with no graphics, just text. You had to use your imagination a lot more back then.

The first task would be to take the input from the user and decide what it is they’re trying to do. In English commands will just about universially have their relevant verbs as the first word of the sentance. “GO WEST”, “KILL THE GOBLIN”, “EAT THE SUSPICIOUS-LOOKING MUSHROOM”. The relevant verbs here are GO, KILL and EAT respectively.

Here’s how we’d use C# 11 pattern matching:

```
var verb = input.Split(" ") switch
{
        ["GO", "TO",.. var rest] => this.actions
        ["GO", .. var rest] => this.actions.GoTo(
        ["EAT", .. var rest] => this.actions.Eat(
        ["KILL", .. var rest] => this.actions.Ki
};
```

The “..” in the above switch expression means “I don’t care what else is in the array, ignore it”. Putting a variable after it is used to contain everything else in the array besides those bits that’re specifically matched for.

In my example above, if I were to enter the text “GO WEST”, then the GoTo action would be called with a single-element array ["WEST"] as a parameter, because “GO” was part of the match.

Here’s another neat way of using it. Imagine I’m processing people’s names into data structures and I want 3 of them to be FirstName, LastName and an array - MiddleNames (I’ve only got one middle name, but plenty of folks have many).

```
public class Person
{
        public string FirstName { get; set; }
        public IEnumerable<string> MiddleNames {
```

```
        public string LastName { get; set; }
}

// The real name of Doctor Who actor, Sylvester M
var input = "Percy James Patrick Kent-Smith".Spli

var sylv = new Person
{
        FirstName = input.First(),
        MiddleNames = input is [_, .. var mns, _]
        LastName = input.Last()
};
```

In this example, the Person class is instantiated with:

FirstName = “Percy”, LastName = “Kent-Smith”, MiddleNames = [“James”, “Patrick”]

I’m not sure I’ll find many uses for this, but it’ll probably get me very excited when I do. It’s a very powerful feature.

Discriminated Unions

I’m not sure whether this is something we’ll ever get in C# or not. I’m aware of at least 2 attempts to implement this concept that are available currently in Nuget:

1. Harry McIntyre’s OneOf (*https://github.com/mcintyre321/OneOf*)
2. Kim Hugener-Olsen’s Sundew.DiscriminatedUnions (*https://github.com/sundews/Sundew.DiscriminatedUnions*)

I cover Discriminated Unions and how they could be implemented in C# in an awful lot more detail in Chapter 6, so skip ahead to there if you want to see more.

In brief: they’re a way of having a type that might actually be one of several types. They’re available natively in F#, but C# doesn’t have them to date and it’s anyone’s guess whether they ever will be available.

In the meantime there are discussions happening over on GitHub (*https://github.com/dotnet/csharplang/issues/113*) and proposals in existence (*https://github.com/dotnet/csharplang/blob/main/proposals/discriminated-unions.md*).

I’m not aware of any serious plans to add them to C# 12, so for now we’ll just have to keep watching the skies!

Active Patterns

This an F# feature I can see being added to C# sooner or later. It’s an enhancement to Pattern Matching that allows functions to be executed in the left hand “pattern” side of the expression. This is an F# example:

```
let (|IsDateTime|_|) (input:string) =
    let success, value = DateTime.TryParse input
    if success then Some value else None

let tryParseDateTime input =
    match input with
    | IsDateTime dt -> Some dt
    | _ -> None
```

What F# developers are able to do, as in this example, is to provide their own custom functions to go on the left hand “pattern” side of the expression.

“IsDateTime” is the custom function here, defined on the first line. It takes a string, and returns a value if the parse worked, and what is effectively a null result if it doesn’t.

The pattern match expression “tryParseDateTime” uses IsDateTime as the pattern, if a value is returned from IsDateTime, then that case on the pattern match expression is selected and the resulting DateTime is returned.

Don’t worry too much about the intricacies of F# syntax, I’m not expecting you to learn about that here. There are other books for F#, and you could probably do worse than one or more of these:

1. Get Programming with F# by Isaac Abraham (Manning)

2. Essential F# by Ian Russell (*https://leanpub.com/essential-fsharp*)
3. F# for Fun and Profit by Scott Wlaschin (*https://fsharpforfunandprofit.com/*)

Whether either of these F# features becomes available in a later version of C# remains to be seen, but C# and F# share a common language runtime, so it's not beyond imagining that they might be ported over.

Read-only Structs

I'm not going to discuss Structs a great deal here, there are other excellent books that talk about the features of C#[2] in far more detail. What's great about them from a C# perspective is that they're passed between functions by value, not reference - i.e. a copy is passed in, leaving the original untouched. The old OO technique of passing an object into a function for it to be modified there, away from the function that instantiated it - this is anathema to a Functional Programmer. We instantiate an object based on a class, and never change it again.

Structs have been around for an awfully long time, and although they're passed by value, they can still have their properties modified, so they aren't immutable as such. At least until C# 7.2.

Now, it's possible to add a readonly modifier to a Struct definition, which will enforce all properties of the struct as readonly at design time. Any attempt to add a setter to a property will result in a Compiler Error.

Since all properties are enforced as readonly, in C# 7.2 itself, all properties need to be included in the constructor to be set. It would look like this:

```
public readonly struct Movie
{
    public string Title { get; private set; };
    public string Directory { get; private set; ]
    public IEnumerable<string> Cast { get; privat


    public Movie(string title, string directory,
    {
        this.Title = title;
        this.Directory = directory;
        this.Cast = cast;
    }
}

var bladeRunner = new Movie(
        "Blade Runner",
        "Ridley Scott",
        new []
        {
            "Harrison Ford",
            "Sean Young"
        }
);
```

This is still a little clunky, forcing us to update the constructor with every property as they're added to the struct, but it's still better than nothing.

It's also worth discussing this case, where I've added in a List to the struct:

```
public readonly struct Movie
{
    public readonly string Title;
    public readonly string Directory;
    public readonly IList<string> Cast;


    public Movie(string title, string directory,
    {
        this.Title = title;
        this.Directory = directory;
        this.Cast = cast;
    }
}

var bladeRunner = new Movie(
        "Blade Runner",
        "Ridley Scott",
        new []
```

```
            {
                "Harrison Ford",
                "Sean Young"
            }
);

bladeRunner.Cast.Add(("Edward James Olmos"));
```

This will compile, and the application will run, but an error will be thrown when the `Add()` function is called. It’s nice that the read-only nature of the struct is being enforced, but I’m not a fan of having to worry about another potential unhandled exception.

But it’s a good thing that the developer can now add the readonly modifier to clarify intent, and it will prevent any easily-avoidable mutability being added to the struct. Even if it does mean that there has to be another layer of error handling.

Init only setters

C# 9 introduced a new kind of auto-property type. We’ve already got `Get` and `Set`, but now there’s also `Init`.

If you have a class property with `Get` and `Set` attached to it, that means the property and be retrieved or replaced at any time.

If instead it has `Get` and `Init`, the it can have its value set when the object it’s part of is instantiated, but can’t then be changed again.

That means our read-only structs (and, indeed all of our classes too) can now have a slightly nicer syntax to be instantiated and then exist in a read-only state:

```
public readonly struct Movie
{
    public string Title { get; init; }
    public string Director { get; init;  }
    public IEnumerable<string> Cast { get; init
}

var bladeRunner = new Movie
   {
       Title = "Blade Runner",
       Director = "Ridley Scott",
       Cast = new []
       {
           "Harrison Ford",
           "Sean Young"
       }
   };
```

This means we don’t have to maintain a convoluted constructor (i.e. one with a parameter for literally every single property - and there could be dozens of them), along with the properties themselves, which has removed a potential source of annoying boilerplate code.

We still have the issue with exceptions being thrown when attempting to modify Lists and sub-objects, though.

Record types

In C# 9, one of my favorite features since Pattern Matching was added - record types. If you’ve not had a chance to play with these yourself yet, then do yourself a favor and do it as soon as possible. They’re fantastic.

On the face of it, they look about the same as a struct. In C# 9, a record type is based on a class, and as such is passed around by reference.

As of C# 10 and onwards that’s no longer the case, and records are treated more like structs, meaning they can be passed by value. Unlike a struct however, there is no readonly modifier, so immutability has to be enforced by the developer. This is an updated version of the Blade Runner code:

```
public record Movie
{
    public string Title { get; init; }
    public string Director { get; init;  }
    public IEnumerable<string> Cast { get; init;
}

var bladeRunner = new Movie
   {
       Title = "Blade Runner",
       Director = "Ridley Scott",
       Cast = new []
       {
           "Harrison Ford",
           "Sean Young"
       }
   };
```

It doesn't look all that different, does it? Where records come into their own though, is when you want to create a modified version. Let's imagine for a moment that in our C# 10 application, we wanted to create a new movie record for the Director's Cut of Blade Runner[3]. This is exactly the same for our purposes, except that it has a different title. To save defining data, we'll literally copy over data from the original record, but with one modification. With a read-only struct, we'd have to do something like this:

```
public readonly struct Movie
{
    public string Title { get; init; }
    public string Director { get; init;  }
```

```
     public IEnumerable<string> Cast { get; init
 }

var bladeRunner = new Movie
    {
        Title = "Blade Runner",
        Director = "Ridley Scott",
        Cast = new []
        {
            "Harrison Ford",
            "Sean Young"
        }
    };

var bladeRunnerDirectors = new Movie
{
    Title = $"{bladeRunner.Title} - The Director
    Director = bladeRunner.Director,
    Cast = bladeRunner.Cast
};
```

That’s following the Functional paradigm, and it’s not too bad, but it’s another heap of boilerplate we have to include in our applications if we want to enforce Immutability.

This becomes important if we’ve got something like a state object that needs to be updated regularly following interactions with the user, or external dependencies of some sort. That’s a lot of copying of properties we’d have to do using the read-only struct approach.

Record types gives us an absolutely amazing new keyword - `with`. This is a quick, convenient way of creating a replica of an existing record but with a modification. The updated version of the Director’s Cut code with record types looks like this:

```
 public record Movie
 {
     public string Title { get; init; }
     public string Director { get; init;  }
     public IEnumerable<string> Cast { get; init
 }

var bladeRunner = new Movie
    {
        Title = "Blade Runner",
        Director = "Ridley Scott",
        Cast = new []
        {
            "Harrison Ford",
            "Sean Young"
        }
    };

var bladeRunnerDirectors = bladeRunner with
```

```
{
    Title = $"{bladeRunner.Title} - The Director
};
```

Isn’t that cool? The sheer amount of boilerplate you can save yourself with record types is staggering.

I recently wrote a text adventure game in Functional C#. I made a central GameState record type, containing all of the progress the player has made so far. I used a massive Pattern Matching statement to work out what the player was doing this turn, and a simple **with** statement to update state by returning a modified duplicate record. It’s an elegant way to code state machines, and clarifies intent massively by cutting away so much of the uninteresting boilerplate.

A last neat feature of Records is that you can even define them simply in a single line like this:

```
public record Movie(string Title, string Directo
```

Creating instances of Movie using this style of definition can’t be done with curly braces, it has to be done with a function:

```
var bladeRunner = new Movie(
"Blade Runner",
"Ridley Scott",
new[]
{
        "Harrison Ford",
        "Sean Young"
});
```

Note that all properties have to be supplied and in order, unless you use constructor tags like this:

```
var bladeRunner = new Movie(
        Cast: new[]
         {
                "Harrison Ford",
                "Sean Young"
        },
        Director: "Ridley Scott",
        Title: "Blade Runner");
```

You *still* have to provide all of the properties, but you can put them in any order you like. For all the good that does you…

Which syntax you prefer is a matter of preference. In most circumstances they’re equivalent.

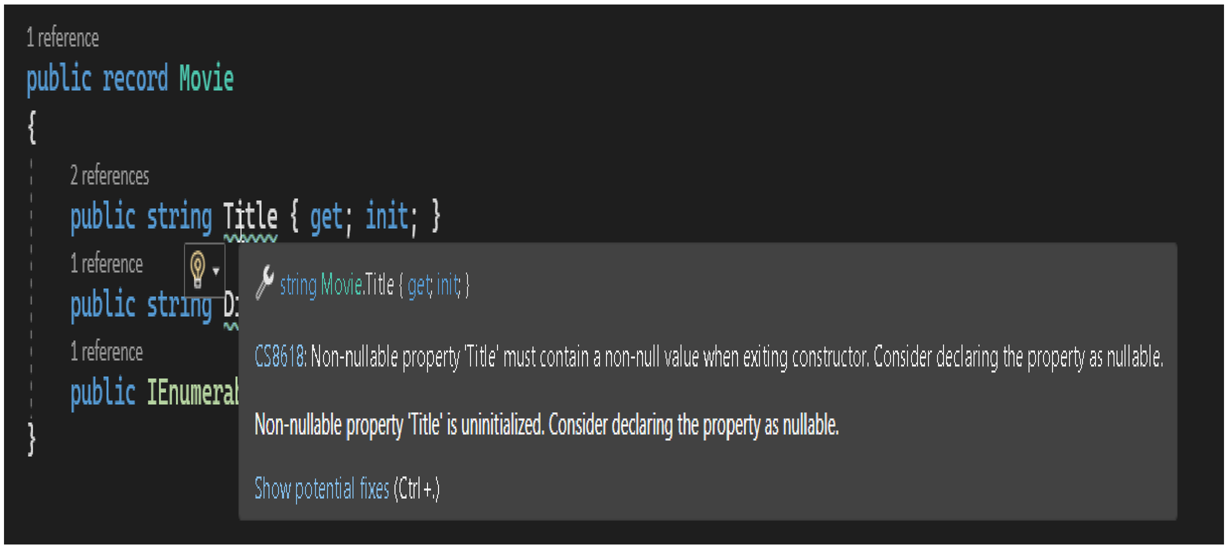
Nullable Reference Types

Despite what it sounds like, this isn’t actually a new type, like with record types. This is effectively a compiler option, which was introduced in C# 8. This option is set in the CSPROJ file, like in this extract:

```
<PropertyGroup>
  <TargetFramework>net6.0</TargetFramework>
  <Nullable>enable</Nullable>
  <IsPackable>false</IsPackable>
</PropertyGroup>
```

If you prefer using a UI, then the option can also be set in the Build section of the project’s properties.

Strictly speaking, activating the Null Reference Types feature doesn’t change the behavior of the code generated by the compiler, but it does add an extra set of warnings to the IDE and the compiler to help avoid a situation where NULL might end up assigned. Here are some that are added to my Movie record type, warning me that it’s possible for properties to end up null:



```
1 reference
public record Movie
{
    2 references
    public string Title { get; init; }
    1 reference
    public string D
    1 reference
    public IEnumerab
}
```

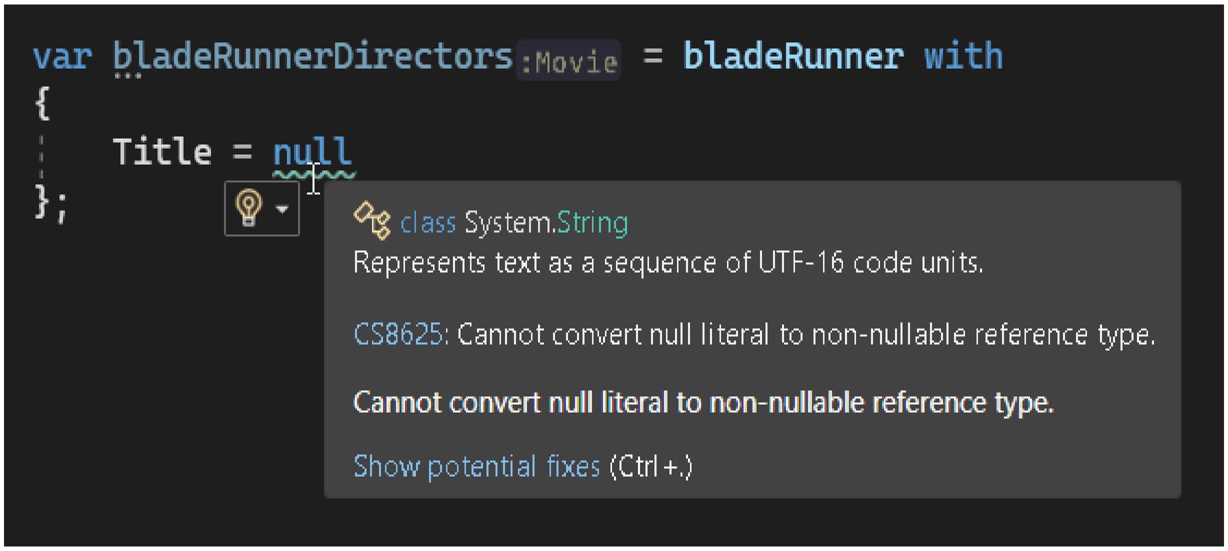
string Movie.Title { get; init; }

CS8618: Non-nullable property 'Title' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.

Non-nullable property 'Title' is uninitialized. Consider declaring the property as nullable.

Show potential fixes (Ctrl+.)

Another example occurs if I try to set the title of the Blade Runner Director's Cut to NULL:



```
var bladeRunnerDirectors:Movie = bladeRunner with
{
    Title = null
};
```

class System.String
Represents text as a sequence of UTF-16 code units.

CS8625: Cannot convert null literal to non-nullable reference type.

Cannot convert null literal to non-nullable reference type.

Show potential fixes (Ctrl+.)

Do bear in mind that these are only compiler warnings. The code will still execute without any errors at all. It's just guiding you to writing code that's less likely to contain null reference exceptions - which can only be a good thing.

Avoiding the use of a NULL value is generally a good practice, functional programming or not. NULL is the so-called “Billion dollar mistake”. It was invented by Tony Hoare in the mid-60s, and it’s been one of the leading causes of bugs in producion ever since. An object being passed into something that turned out unexpextectly to be NULL. This gives rise to a Null-Reference Exception, and you don’t need to have been in this business long before you encounter your first one of those!

Having NULL as a value adds unneeded complexity to your codebase, and introduces another source of potential errrors. This is why it’s worth paying attention to the compiler warning and keeping NULL out of your codebase wherever possible.

If there’s a perfectly good reason for a value to be NUL then you can do so by adding *?* characters to properties like this:

```
public record Movie
{
    public string? Title { get; init; }
    public string? Director { get; init;  }
    public IEnumerable<string>? Cast { get; init
}
```

The only circumstance under which I'd ever consider deliberately adding a nullable property to my codebase is where a 3rd party library requires it. Even then, I wouldn't allow the Nullable to be persisted through the rest of my code - I'd probably tuck it away somewhere where the code that parses the external data can see it, then convert it into a safer, more controlled structure for passing to other areas of the system.

The Future

As of the time of writing, C# 11 is out and well established as part of .NET 7. The plans are only starting to be put together for .NET 8 and C# 12, but it's not clear what - if anything - they'll be including for functional programmers. Given that it's a stated intent by the C# team to continue to add more Functional features with every version, it's a safe bet that there'll be at least *something* new for us to do more Functional Programming with.

Summary

In this chapter, we looked at all the features of C# that have been released since Functional Programming began to be integrated in C# 3 and 4. We looked at what they are, how they can be used, and why they're worth considering.

Broadly these fall into two categories:

- Pattern Matching, implemented in C# as an advanced form of switch statement that allows for incredibly powerful, code-saving logic to be written briefly and simply. We saw how every version of C# has contributed more pattern matching features to the developer.
- Immutability, the ability to prevent variables from being altered once instantiated. It’s highly unlikely that true Immutability will ever be made available in C# for reasons of backwards-compatability, but new features are being added to C#, such as readonly structs and record types that make it easier for a developer to work in such a way that it’s easy to pretend that immutablility exists without having to add a lot of tedious boilerplate code to the application.

In the next chapter, we’re going to take things a step further, and demonstrate some ways to use some existing features of C# in novel ways to add to your Functional Programming tool belt.

Which, frankly no bank would ever offer

You could do worse than *C# in a Nutshell*, also published by O’Reilly

Vastly superior to the theatrical cut, in my opinion.

Chapter 4. Work Smart, Not Hard with Functional Code

Everything I've covered so far has been functional programming as intended by Microsoft's C# team. You'll find these features mentioned, along with examples on the Microsoft website. In this chapter however, I want to start being a bit more creative with C#.

I don't know about you, but I like being lazy, or at least I don't like wasting my time with tedious boilerplate code. One of the many wonderful things about functional programming is how concise it is, compared to Imperative code.

In this chapter I’m going to show you ways to push the Functional envelope further than out-of-the-box C# will allow, as well as ways to implement some of the more recent versions of C# in legacy versions, and hopefully allow you to get on with your day job an awful lot quicker.

There are broadly three categories which this chapter will explore:

Funcs in Enumerables
`Func` delegates don’t seem to get used all that much, but they’re incredibly powerful features of C#. I’ll show a few ways of using them that help extend C#’s capabilities. In this case by adding them to Enumerables and operating on them with Linq expressions.

Funcs as Filters
You can also use Funcs as filters - something that sits between you and the real value you’re trying to reach. There are a few neat bits of code you can write using these principles.

Custom Enumerables
I’ve discussed IEnumerables and how cool they are before, but did you know you can break them open and implement your own customised behavior? I’ll show you how.

It’s Time to get Func-y

I’ve already talked about `Func` delegate types the introduction, but just to recap - they’re functions stored as variables. You define what parameters they take, what they return and call them like any other function. Here’s a quick example:

```
private readonly Func<Person, DateTime, string> S
    (Person p, DateTime today) => today + " : " -
```

The last generic type in the list between the two chevrons is the return value, all of the previous types are the parameters. My example above takes 2 string parameters and returns a string.

We’re going to be seeing an awful lot of Func delegates from now on, so please do make sure you’re comfortable with them before reading on.

Funcs in Enumerables

I’ve seen plenty of examples of Funcs as parameters to functions, but I’m not sure many developers realise you can put them in Enumerable, and create some interesting behaviors.

First, is the obvious one - put them in an array to act on the same data mulitple times:

```
private IEnumerable<Func<Employee, string>> desc
{
    x => "First Name = " + x.firstName,
    x => "Last Name = " + x.lastName,
    x => "MiddleNames = string.Join(" ", x.Middl
}

public string DescribeEmployee(Employee emp) =>
   string.Join(Environment.NewLine, descriptors.
```

Using this method, we can have a single original source of data (here, an Employee object) and have multiple records of the same type generated from it, and in my case I aggreated using the built-in .NET method `string.Join` to present a single, unified string to present to the end user.

There are a few advantages of this method over a simple StringBuilder.

Firstly, the array can be assembled dynamically. There could be multiple rules for each property and how it’s rendered, which could be selected from a set of local variables depending on some custom logic.

Secondly, this is an Enumerable, so by defining it this way, we’re taking advantage of a feature of Enumerables called *Lazy Evaluation*. The thing about Enumerables is that they aren’t arrays, they aren’t even data. They’re just pointers to something that will tell you how to extract the data. It might well be - and in fact usually is the case - that the source at the back of the Enumerable is a simple array, but not necessarily. An Enumerable requires a function to be executed each time the next item is accessed via a `ForEach` loop. Enumerables were developed so that it only transforms into actual data at the very last possible moment - typically when starting a `ForEach` loop iteration. Most of the time this doesn’t matter if there’s an array held in-memory somewhere feeding the Enumerable, but if there’s an expensive function or look-up to an external system powering it, then Lazy Loading can be incredibly useful to prevent unnecessary work.

The elements of an Enumerable are evaluated one at a time and only when their turn has come to be used by whatever process is performing the enumeration. For example, if we use the LINQ `Any` function to evaluate each element in an Enumerable, it will stop enumerating the first time that an element is found that matches the specified criteria, meaning the remaining elements will be left un-evaluated.

Lastly, from a maintanance perspective, this technique is easier to live with. Adding a new line to the final result is as easy as adding a new element to the array. It also acts as a restraint to future programmers, making it harder for them to try to put too much complex logic where it doesn’t belong.

A Super-Simple Validator

Let’s imagine ourselves a quick validation function. They typically look something like this:

```
public bool IsPasswordValid(string password)
{
   if(password.Length <= 6)
      return false;

   if(password.Length > 20)
       return false;

   if(!password.Any(x => Char.IsLower(x)))
      return false;

   if(!password.Any(x => Char.IsUpper(x)))
      return false;

   if(!password.Any(x => Char.IsSymbol(x)))
      return false;

   if(password.Contains("Justin", StringCompariso
      && password.Contains("Bieber", StringCompar
       return false;
```

```
    return true;
}
```

Well, for a start, that’s a **lot** of code for what is, in fact, a fairly simple set of rules. There’s a whole heap of repetitive boilerplate code that Imperative code forces us to write here. On top of that, if we want to add in another rule, that’s potentially around 4 new lines of code to add when really only 1 is especially interesting to us.

If only there were a way to compact it into just a few simple lines of code…

Well…since you asked so nicely, here you go:

```
public bool IsPasswordValid(string password) =>
    new Func<string, bool>[]
    {
        x => x.Length > 6,
        x => x.Length <= 20,
        x => x.Any(y => Char.IsLower(y)),
        x => x.Any(y => Char.IsUpper(y)),
        x => x.Any(y => Char.IsSymbol(y)),
        x => !x.Contains("Justin", StringComparis
            && !x.Contains("Bieber", StringCompar
    }.All(f => f(password));
```

Not so long now, is it? What is it I’ve done here? I’ve put all of the rules into an array of Funcs that turn a string into a bool - i.e. check a single validation rule. I’ve used a Linq statement - `.All()`. The purpose of this function is to evaluate whatever lambda expression I give it against all elements of the array it’s attached to. If a single one of these returns false, then the process is terminated early and false is returned from the `All` (As mentioned earlier, the subsequent values aren’t accesssed, so Lazy Evaluation saves us time by not evaluating them). If every single one of the items returns true, then the All also returns true.

We’ve effectively re-created the first code sample, but the boilerplate code we were forced to write - If statements and early returns - is now implicit in the structure.

This also has the advantage of once again being very easy to maintain, as a code structure. If you wanted, you could even generalize it into an extension method. Something I often do. Something like this:

```
public static bool IsValid<T>(this T @this, param
    rules.All(x => x(@this));
```

This reduces the size of the password validator yet further, and gives you a handy, generic structure to use elsewhere:

```
public bool IsPasswordValid(string password) =>
    password.IsValid(
        x => x.Length > 6,
        x => x.Length <= 20,
        x => x.Any(y => Char.IsLower(y)),
        x => x.Any(y => Char.IsUpper(y)),
        x => x.Any(y => Char.IsSymbol(y)),
        x => !x.Contains("Justin", StringComparis
            && !x.Contains("Bieber", StringCompar
    )
```

At this point, I hope you're re-considering ever writing something as long and ungainly as that first validation code sample ever again.

I think an `IsValid` check is easier to read and maintain, but if you wanted a piece of code that is much more in line with the original code sample, then a new extension method can be created using an Any instead of an All:

```
public static bool IsInvalid<T>(this T @this, par
    rules.Any(x => @this);
```

This means that the boolean logic of each array element can be reversed, as they were originally:

```
public bool IsPasswordValid(string password) =>
    !password.IsInvalid(
        x => x.Length <= 6,
        x => x.Length > 20,
        x => !x.Any(y => Char.IsLower(y)),
        x => !x.Any(y => Char.IsUpper(y)),
        x => !x.Any(y => Char.IsSymbol(y)),
        x => x.Contains("Justin", StringComparis
            && x.Contains("Bieber", StringCompari
    )
```

If you want to maintain both functions: `IsValid` and `IsInvalid` because each have their place in your codebase, it's probably worth saving yourself some coding effort, and preventing a potential maintenance task in the future by simply referencing one in the other:

```
public static bool IsValid<T>(this T @this, param
 rules.All(x => x(@this));

public static bool IsInvalid<T>(this T @this, par
 !@this.IsValid(rules);
```

Use it wisely, my young Functional Padawan Learner.

Pattern Matching for Old Versions of C#

Pattern matching is one of the very best features of C# in recent years, along with record types, but it isn't available in anything except the most recent .NET versions - See Chapter 3 for more details on native Pattern Matching in C# 7 and up.

Is there a way to allow pattern matching to happen, but without needing up upgrade to a newer version of C#?

There most certainly is. It is nowhere near as elegant as the native syntax in C# 8, but it provides a few of the same benefits.

In this example, I'll calculate how much tax someone should pay based on a grossly simplified version of the UK Income Tax rules. Please note, these really are much simpler than the real thing. I don't want to get too bogged down in the complexities of tax.

The rules I'm going to apply look like this:

- Yearly income ⇐ £12,570, then no tax is taken
- Yearly income is between £12,571 and £50,270, then take 20% tax
- Yearly income is between £50,271 and £150,000, then take 40% tax
- Yearly income is over £150,000, then take 45% tax

If you wanted to write this long-hand (i.e. non-functionally), it would look like this:

```
decimal ApplyTax(decimal income)
{
 if (income <= 12570)
  return income;
 else if (income <=50270)
         return income * 0.8M;
 else if (income <= 150000)
         return income * 0.6M;
 else
         return income * 0.55M;
}
```

Now, in C# 8 and onwards, switch expressions would compress this to a few lines. So long as you’re running at least C# 7 (i.e. .NET Framework 4.7) this is the style of pattern matching I can create:

```
var inputValue = 25000M;
var updatedValue = inputValue.Match(
 (x => x <= 12570, x => x),
 (x => x <= 50270, x => x * 0.8M),
 (x => x <= 150000, x => x * 0.6M)
).DefaultMatch(x => x * 0.55M);
```

I’m passing in an array of tuples containing 2 lambda expressions. The first determines whether the input matches against the current pattern, the second is the transformation in value that occurs if the pattern is a match.

There’s a final check to see whether the default pattern should be applied - i.e. because none of the other patterns were a match.

Despite being a fraction of the length of the original code sample, this contains all of the same functionality. My matching patterns on the left-hand side of the tuple are simple here, but they can contain expressions as complicated as you’d like, and could even be calls to whole functions containing detailed criteria to match on.

So, how did I make this work? This is an extremely simple version that provides most of the functionality required:

```
public static class ExtensionMethods
{
        public static TOutput Match<TInput, TOut
                                     this TInput
                                     params (Func
                                     Func<TInput,
        {
                var match = matches.FirstOrDefaul
                var returnValue = match.Transform
                return returnValue;
        }
}
```

I'm using the Linq method `FirstOrDefault` to first iterate through the left-hand functions to find one that returns true (i.e. one with the right criteria) then call the right-hand conversion Func to get my modified value.

This is fine, except that if *none* of the patterns match, we'll be in a bit of a fix. There will most likely be a null reference exception.

To cover this, we need to force the need to provide a default match (the equivalent of a simple `else` statement, or the _ pattern match in switch expressions).

My answer is to have the `Match` function return a placeholder object that holds either a transformed value from the Match expressions, or else executes the Default pattern lambda expression. The improved version looks like this:

```
public static MatchValueOrDefault<TInput, TOutput
  this TInput @this,
  params (Func<TInput, bool>,
  Func<TInput, TOutput>)[] predicates)
{
        var match = predicates.FirstOrDefault(x =
        var returnValue = match?.Item2(@this);
        return new MatchValueOrDefault<TInput, TO
}

public class MatchValueOrDefault<TInput, TOutput>
```

```
  {
   private readonly TOutput value;
   private readonly TInput originalValue;

   public MatchValueOrDefault(TOutput value, TInput
   {
          this.value = value;
          this.originalValue = originalValue;
   }

  public TOutput DefaultMatch(Func<TInput, TOutput>
  {
   if (EqualityComparer<TOutput>.Default.Equals(def
   {
          return defaultMatch(this.originalValue);
   }
   else
    {
          return this.value;
    }
  }
```

This method is severely limited compared to what can be accomplished in the latest versions of C#. There is no object type matching, and the syntax isn’t as elegant, but it’s still usable and could save an awful lot of boilerplate, as well as encouraging good code standards.

Versions of C# that are older still, and which don't include Tuples, can consider the use of `KeyValuePair<T,T>`, though the syntax is far from attractive.

What, you don't want to take my word? Ok, here we go. Don't say I didn't warn you…

The Extension method itself is just about the same, and just needs a small alteration to use `KeyValuePair` instead of Tuples:

```
public static MatchValueOrDefault<TInput, TOutput
  this TInput @this,
  params KeyValuePair<Func<TInput, bool>, Func<TI
{
    var match = predicates.FirstOrDefault(x => x
    var returnValue = match.Value(@this);
    return new MatchValueOrDefault<TInput, TOutpu
}
```

And here's the ugly bit. The syntax for creating `KeyValuePair` objects is pretty awful:

```
var inputValue = 25000M;
var updatedValue = inputValue.Match(
        new KeyValuePair<Func<decimal, bool>, Fur
                x => x <= 12570, x => x),
        new KeyValuePair<Func<decimal, bool>, Fur
                x => x <= 50270, x => x * 0.8M),
        new KeyValuePair<Func<decimal, bool>, Fur
                x => x <= 150000, x => x * 0.6M)
).DefaultMatch(x => x * 0.55M);
```

So you *can* still have a form of Pattern Matching in C# 4, but I'm not sure how much you're gaining by doing it. That's perhaps up to you to decide. At least I've shown you the way.

Functional Filtering

Functions don't only have to be used for turning one form of data into another, you can also use them as filters, extra layers, that sit between the developer and an original source of information or functionality.

This section looks at a few examples of how this method can be used to make your daily C# coding look simpler, and also be less error prone.

Make Dictionaries More Useful

One of my absolute favorite things in C# by far is Dictionaries. Used appropriately they can reduce a heap of ugly, boilerplate-riddled code with a few simple elegant array-like lookups. They're also very efficient to find data in, once created.

They have a problem, however, that makes it often necessary to add in a heap of boilerplate that invalidates the whole reason they’re so lovely to use. Consider the following code sample:

```
var doctorLookup = new []
{
        ( 1, "William Hartnell" ),
        ( 2, "Patrick Troughton" ),
        ( 3, "Jon Pertwee" ),
        ( 4, "Tom Baker" )
}.ToDictionary(x => x.Item1, x => x.Item2);

var fifthDoctorInfo = $"The 5th Doctor was playe
```

What’s up with this code? It falls foul of a code feature of Dictionaries that I find inexplicable, if you try looking up an entry that doesn’t exist[1], it will trigger an Exception that has to be handled!

The only safe way to handle this is to use one of several methods available in C# to check against the available keys before compiling the string, like this:

```
var doctorLookup = new []
{
        ( 1, "William Hartnell" ),
        ( 2, "Patrick Troughton" ),
        ( 3, "Jon Pertwee" ),
        ( 4, "Tom Baker" )
}.ToDictionary(x => x.Item1, x => x.Item2);

var fifthDoctorActor = doctorLookup.ContainsKey(5

var fifthDoctorInfo = $"The 5th Doctor was playe
```

Alternatively, using slightly newer versions of C#, there is a `TryGetValue` function available as well to simplify this code a little:

```
   var fifthDoctorActor = doctorLookup.TryGetValue(
```

So, can we use functional programming techniques to reduce our boilerplate code, and give us all of the useful features of Dictionaries, but without the awful tendency to explode on us? You bet'cha!

First I need a quick extension method:

```
  public static class ExtensionMethods
  {
          public static Func<TKey, TValue> ToLookup
            this IDictionary<TKey,TValue> @this)
          {
                  return x => @this.TryGetValue(x,
          }

          public static Func<TKey, TValue> ToLookup
            this IDictionary<TKey,TValue> @this,
            TValue defaultVal)
          {
                  return x => @this.ContainsKey(x)
          }
  }
```

I’ll explain further in a minute, but first, here’s how I’d use my extension methods:

```
var doctorLookup = new []
{
        ( 1, "William Hartnell" ),
        ( 2, "Patrick Troughton" ),
        ( 3, "Jon Pertwee" ),
        ( 4, "Tom Baker" )
}.ToDictionary(x => x.Item1, x => x.Item2)
        .ToLookup("An Unknown Actor");

var fifthDoctorInfo = $"The 5th Doctor was playe
// output = "The 5th Doctor was played by An Unk
```

Notice the difference?

If you look carefully, I’m now using rounded function brackets, rather than square array/dictionary brackets to access values from the Dictionary. That’s because it’s technically not a dictionary any more! It’s a function.

If you look at my extension methods, they return functions, but they’re functions that keep the original Dictionary object in scope for as long as they exist. Basically, they’re like a filter layer sitting between the Dictionary and the rest of the codebase. The functions make a decision on whether use of the Dictionary is safe or not.

It means we can use a Dictionary, but the Exception that occurs when a key isn’t found will no longer be thrown, and we can either have the default for the type (usually Null) returned, or supply our own default value. Simples.

The only down side to this method is that it’s no longer a dictionary, in effect. This means you can’t modify it any further, or perform any LINQ operations on it. If you are a situation though, where you’re sure you won’t need to, then this is something you can use.

Parsing Values

Another common cause of noisy, boilerplate code is parsing values from `string` to other forms. Something like this for parsing in a hypothetical settings object, in the event we were working in .NET Framework, and the appsettings.JSON and `IOption<T>` features aren’t available:

```
public Settings GetSettings()
{
        var settings = new Settings();

        var retriesString = ConfigurationManager
        var retriesHasValue = int.TryParse(retrie
        if(retriesHasValue)
                settings.NumberOfRetries = retrie
        else
                settings.NumberOfRetries = 5;

        var pollingHourStr = ConfigurationManager
        var pollingHourHasValue = int.TryParse(po
        if(pollingHourHasValue)
                settings.HourToStartPollingAt = p
        else
                settings.HourToStartPollingAt = 0

        var alertEmailStr = ConfigurationManager
        if(string.IsNullOrWhiteSpace(alertEmailSt
                settings.AlertEmailAddress = "tes
        else
                settings.AlertEmailAddress = aea

        var serverNameString = ConfigurationManag
        if(string.IsNullOrWhiteSpace(serverNameSt
                settings.ServerName = "TestServer
        else
```

```
                  settings.ServerName = sn.ToStrin

          return settings;
}
```

That’s a lot of code to do something simple, isn’t it? There’s a lot of boilerplate code noise there that obscures the intention of the code to all but those familiar with these sorts of operations. Also, if a new setting were to be added, it would take 5 or 6 lines of new code for each and every one. That’s quite a waste.

Instead, we can do things a little more Functionally and hide the structure away somewhere, leaving just the intent of the code visible for us to see.

As usual, here’s an extension method to take care of business for me:

```
public static class ExtensionMethods
{
        public static int ToIntOrDefault(this ob
                int.TryParse(@this?.ToString() ?
                      ? parsedValue
                      : defaultVal;

        public static string ToStringOrDefault(t
                string.IsNullOrWhiteSpace(@this?
                ? defaultVal
```

```
                    : @this.ToString();
}
```

This takes away all of the repetitive code from the first example, and allows us to move to a more readable, result-driven code sample, like this:

```
public Settings GetSettings() =>
        new Settings
        {
                NumberOfRetries = ConfigurationMa
                                         .ToIntOr
                HourToStartPollingAt = Configurat

                AlertEmailAddress = Configuration
                                  .ToStringOrDefa
                ServerName = ConfigurationManager



        };
```

It’s easy now to see at a glance what the code does, what the default values are, and how you’d add more settings with a single line of code.

Any other settings value types besides `int` and `string` would need another extension method creating, but that’s no great hardship.

Custom Enumerations

Most of us have likely used Enumerables when coding, but did you know that there's an engine under the surface that you can access and use to create all sorts of interesting custom behaviors?

With a custom iterator, we can drastically reduce the number of lines of code needed when more complicated behavior is needed when looping through data. First though, it's necessary to understand just how an Enumerable works beneath the surface.

There's a class sitting beneath the surface of the Enumerable, the engine which drives the Enumeration, which is what allows you to use a ForEach to loop through values. It's called the Enumerator.

The Enumerator has basically two features:

- Current: Get the current item out of the enumerable. This may be called as many times as you want, provided you don't try moving to the next item. If you try getting the Current value before first calling MoveNext, an exception is thrown.
- MoveNext: Move from the current item, and try to see whether there is another to be selected or not. Returns `True` if another value is found, `False` if we've reached the end of the Enumerable, or there were no

elements in the first place. The first time this is called, it points the Enumerator at the first element in the Enumerable.

Query Adjacent Elements

A relatively simple example to start with. Let’s imagine that I want to run through an Enumerable of Integers, to see whether it contains any numbers that are consecutive.

An Imperative solution would likely look something like this:

```
public IEnumerable<int> GenerateRandomNumbers()
{
        var rnd = new Random();
        var returnValue = new List<int>();
        for (var i = 0; i < 100; i++)
        {
                returnValue.Add(rnd.Next(1, 100)
        }
        return returnValue;
}

public bool ContainsConsecutiveNumbers(IEnumerabl
{
        // OK, you caught me out OrderBy isn't st
        // there's no way I'm going to write out
        // here just to prove a point!
        var sortedData = data.OrderBy(x => x).To

        for (var i = 0; i < sortedData.Length - 
        {
                if ((sortedData[i] + 1) == sorte
                        return true;
        }

        return false;
}

var result = ContainsConsecutiveNumbers(Generate
Console.WriteLine(result);
```

To make this code functional, as is often the case, we need an extension method. Something that takes the Enumerable, extracts its Enumerator and controls the customised behavior.

To avoid use of an imperative-style loop, I'm using recursion here. In brief - recursion is a way of implementing an indefinite[2] loop by having a function call itself repeatedly.

I'll revisit the concept of recursion in a later chapter. For now though, I'm just going to use the standard, simple version of recursion

```
public static bool Any<T>(this IEnumerable<T> @th
{
        using var enumerator = @this.GetEnumerato
        var hasElements = enumerator.MoveNext();
        return hasElements && Any(enumerator, eva
}

private static bool Any<T>(IEnumerator<T> enumera
                Func<T, T, bool> evaluator,
                T previousElement)
{
        var moreItems = enumerator.MoveNext();
        return moreItems && (evaluator(previousEl
                ? true
                : Any(enumerator, evaluator, enur

}
```

So, what’s happening here? It’s kind’ve like juggling, in a way. I start by extracting the Enumerator, and moving to the first item.

Inside the private function, I accept the enumerator (now pointing to the first item), the “are we done” evaluator function and a copy of that same first item.

What I do then is immediately move to the next item, and run the evaluator function, passing in the first item, and the new “current”, so they can be compared.

At this point either we find out we’ve run out of items or the evaluator returns true, in which case we can terminate the iteration. If MoveNext returned true then we check if the `previousValue` and `Current` match our requirement (as specified by `evaluator`). If they do then we finish and return true, otherwise we make a recursive call to check the rest of the values.

This is the updated version of the code to find consecutive numbers:

```
public IEnumerable<int> GenerateRandomNumbers()
{
 var rnd = new Random();

 var returnValue = Enumerable.Repeat(0, 100)
  .Select(x => rnd.Next(1, 100));
 return returnValue;
}

public bool ContainsConsecutiveNumbers(IEnumerabl
{
 var sortedData = data.OrderBy(x => x).ToArray()
 var result = sortedData.Any((prev, curr) => cur
 return result;
}
```

It would also be easy enough to create an `All` method based on the same logic, like so:

```
  public static bool All<T>(this IEnumerator<T> enu
  {
          var moreItems = enumerator.MoveNext();
          return moreItems
                  ? evaluator(previousElement, enur
                                  ? All(enumerator,
                                  : false
                  : true;
  }

  public static bool All<T>(this IEnumerable<T> @th
  {
          using var enumerator = @this.GetEnumerato
          var hasElements = enumerator.MoveNext();
          return hasElements
           ? All(enumerator, evaluator, enumerator
           : true;
  }
```

The only difference are the conditions for deciding whether or not to continue, and whether we need to return early. With an `All`, the point is to check every pair of values, and only return out of the loop early if one is found not to meet the criteria.

Iterate Until a Condition is Met

This is basically a replacement for a while loop, so that there's another statement we don't necessarily need.

For my example, I want to imagine what the turn system might be like for a text-based adventure game. For younger readers - this is what we had in the old days, before graphics[3]. You used to have to write what you wanted to do and the game would write what happened. Kind of like a book, except you wrote what happened yourself.

The basic structure of one of those games was something like:

- Write a description of the current location
- Receive user input
- Execute requested command

Here's how Imperative code might handle that situation:

```
var gameState = new State
{
  IsAlive = true,
  HitPoints = 100
};

while(gameState.IsAlive)
{
  var message = this.ComposeMessageToUser(gameSta
  var userInput = this.InteractWithUser(message)
  this.UpdateState(gameState, userInput);

  if(gameState.HitPoints <= 0)
    gameState.IsAlive = false;
}
```

In principle what we want is a Linq-style Aggregate function, but one that doesn’t loop through all of the elements of an array, then finish. Instead we want it to loop continuously until our end condition (the player is dead) is met. I’m simplifying a little here, obviously our player in a proper game could *win* as well. But my example game is like life, and life’s not fair!

The extension method for this is another place where we’d benefit from tail recursion optimised calls, and I’ll be looking into options for that in a later chapter, but for now I’ll just use simple recursion - which may become an issue if there are a lot of turns, but for now it’d stop me from introducing too many ideas too soon.

```
public static class ExtensionMethods
{
        public static T AggregateUntil<T>(
          this T @this,
          Func<T,bool> endCondition,
          Func<T,T> update) =>
                endCondition(@this)
                         ? @this
                         : AggregateUntil(update(
}
```

Using this, I can do away with the `while` loop entirely and transform the entire turn sequence into a single function, like so:

```
var gameState = new State
{
  IsAlive = true,
  HitPoints = 100
};

var endState = gameState.AggregateUntil(
       x => x.HitPoints <= 0,
       x => {
         var message = this.ComposeMessageToUse
         var userInput = this.InteractWithUser(
            return this.UpdateState(x, userInput);
        });
```

This isn’t perfect, but it’s functional now. There are far better ways of handling the multiple steps of the update to the game’s state, and there’s the issue of how to handle user interaction in a functional manner, too. There’s a section coming up on that in chapter [X].

Conclusion

In this chapter, we looked at ways to use Funcs, Enumerables and Extension methods to extend C# to make it easier to write Functional-style code, and to get around a few existing limitations of the language.

I’m certain that I’m barely scratching the surface with these techniques, and that there are plenty more out there to be discovered and used.

In our next chapter, we’ll be looking at Higher-order functions, and some structures that can be used to take advantage of them to create yet more useful functionality.

Incidentally, it was Peter Davison.

but hopefully not infinite!

go and check out the epic adventure game Zork if you’d like to see this for yourself. Try not to get eaten by a Grue!

Part II. Into the Belly of the Functional

Hail, bold Adventurer!

I see you’ve survived the first part of your journey. Now you can consider yourself promoted from functional Apprentice to Journeyman. This next road is longer, twistier and far, far stranger.

Be not afraid though, because if you are open to it, there are wonders awaiting you ahead.

This is where we stop thinking entirely in terms of out-of-the-box C# and start looking a little more into functional theory.

I hope you aren’t expecting formal definitions, or talk on list theory, because you won’t find it here. I’m going to take you up the gentle slope, a step at a time. Always with an eye to what’s actually useful to you in your daily coding.

Come along. The horses are getting restless. It’s time to saddle up!

Chapter 5. Higher-Order Functions

Welcome back my friends to the show that never ends.

This chapter, we're looking at uses for higher-order functions. I'm going to look at novel ways to use them in C# to save yourself effort, and to make code that is less likely to fail.

But, what *are* Higher-order functions?

Higher-order Functions is a slightly odd name for something very simple. In fact you've likely been using them for some time if you've spent much time working with LINQ. They come in two flavors, here’s the first:

```
var liberatorCrew = new []
{
 "Roj Blake",
 "Kerr Avon",
 "Vila Restal",
 "Jenna Stannis",
 "Cally",
 "Olag Gan",
 "Zen"
};
var filteredList = liberatorCrew.Where(x => x.Fi
```

Passed into the `Where` function there is an arrow expression - which is just a shorthand for writing out an unnamed function. The long-hand version would look like this:

```
function bool IsGreaterThanM(char c)
{
 return c > 'M';
}
```

So here, the function has been passed around as the parameter to another function, to be executed elsewhere inside it.

This is another example of the use of higher-order functions:

```
public Func<int, int> MakeAddFunc(int x) => y =>
```

Notice here that there are two arrows, not one. We're taking an integer *x* and from that returning a new function. In that new function references to *x* will be filled in with whatever was provided when MakeAddFunc was called originally.

For example:

```
var addTenFunction = MakeAddFunc(10);
var answer = addTenFunction(5);

// answer is 15
```

By passing 10 into `MakeAddFunc` in the example above, I created a new function whose function is simply to add 10 to whatever addtional integer you pass into it.

In short a higher-order function is a function with one or more of the following properties:

- Accepts a function as a parameter
- Returns a function as its return type

In C# this is all typically done with either a `Func` (for functions with a return type) or `Action` (for functions that return void) delegate types.

It’s a fairly simple idea, and even easier to implement - but the effect they can have on your codebase is incredible.

In this chapter I’m going to walk through ways of using Higher-Order Functions to improve your daily coding.

I’ll also be looking quite a bit into a next-level usage of Higher-Order functions called Combinators. These enable passing around functions in a way that creates a more complex - and useful - behavior. They’re called that incidentally, because they originate from a mathematical technique called Combinatory Logic. You won’t need to worry about ever hearing that term again, or about any references to advanced maths - I’m not going there. It’s just in case you were curious…

A Problem Report

To get started, we’ll look at a bit of problem code. Let’s imagine that your company have asked you for a function to take a data store of some kind (an XML file, a JSON file, who knows. Doesn’t matter), summarise how many there are of each possible value, then transmit that data on to somewhere else. On top of that, they want a separate message to be sent in the event that no data was found at all. I run a really, really loose ship, so let’s keep things fun, and imagine you work for the Evil Galactic Empire™ and you are cataloguing Rebel Alliance ships on your radar.

The code might look something like this:

```
public void SendEnemyShipWeaponrySummary()
{
 try
 {
   var enemyShips = this.DataStore.GetEnemyShips
   var summaryNumbers = enemyShips.GroupBy(x =>
                                   .Select(x =>
   var report = new Report
   {
       Title = "Enemy Ship Type",
       Rows = summaryNumbers.Select(X => new Rep
       {
           ColumnOne = X.Type,
           ColumnTwo = X.Count.ToString()
       })
   };

   if (!report.Rows.Any())
       this.CommunicationSystem.SendNoDataWarnin
   else
       this.CommunicationSystem.SendReport(repor
 }
 catch (Exception e)
```

```
    {
     this.Logger.LogError(e,
     $"An error occurred in {nameof(SendEnemyShipWe
    }
   }
```

This is fine, isn't it? Isn't it? Well, think about this scenario. You're sitting at your desk, eating your daily pot noodle[1], when you notice that - Jurassic Park style - there is a rhythmic ripple appearing in your coffee. This signals the arrival of your worst nightmare. Your boss! Let's imagine that your boss is - thinking totally at random here - a tall, deep-voiced gentleman in a black cape and with appauling asthma. He also really hates it when people displease him. *Really* hates it.

He's happy with the first function you created. For this you can breath a sigh of relief. But now he wants a second function. This one is going to create another summary, but this time of the level of weaponry in each ship. Whether they are unarmed, lightly armed, heavily armed or capable of destroying planets. That sort of thing.

Easy, you think. The boss will be so impressed with how quickly I do this. So, you do what seems easiest `Ctrl+C`, then `Ctrl+V` to copy & paste the original, change the name, change the property you're summarising, and you end up with something like this:

```
public void GenerateEnemyShipWeaponrySummary()
{
 try
 {
   var enemyShips = this.DataStore.GetEnemyShips
   var summaryNumbers = enemyShips.GroupBy(x =>
                                        .Select(x =>
   var report = new Report
   {
       Title = "Enemy Ship Weaponry Level",
       Rows = summaryNumbers.Select(X => new Rep
       {
           ColumnOne = X.Type,
           ColumnTwo = X.Count.ToString()
       })
   };

   if (!report.Rows.Any())
       this.CommunicationSystem.SendNoDataWarnin
   else
       this.CommunicationSystem.SendReport(repo
 }
 catch (Exception e)
 {
  this.Logger.LogError(e,
  $"An error occurred in {nameof(GenerateEnemySh
 }
}
```

Five seconds of work, and a day or two of leaning on your figurative shovel with the odd complaint out-loud of how hard the work is here, all while you secretly work on today’s Wordle. Job done, and slaps on the back all round, right? Right?

Well….There are a couple of problems with this approach.

First, let’s think about unit testing. As good, upstanding code citizens, we unit test all of our code. Imagine we’d unit tested the snot out of that first function. When we copied & pasted the second in, what was the level of unit test coverage at that point?

I’ll give you a clue - it was between zero and zero. You could copy and paste the tests too, and that would be fine, but that’s now an awful lot more code that we’re copying and pasting every time.

This isn’t an approach that scales up well. What if our boss wanted another function after this one, and another, and another. What if we ended being asked for 50 functions? Or 100?!? That’s a lot of code. You’d end up with something thousands of lines long, not something I’d be keen to support.

It gets worse when you consider something that happened to me near the beginnning of my career. I was working for an organisation that had a desktop application that carried out a series of complex calculations for each customer, based on a few input parameters. Each year, the rules changed, but the old rule bases had to be replicated because it might be necessary to see what would have been calculated in a previous year.

So, the folks that had been developing the app before I joined the team had copied a whole chunk of code every year. Made a few little changes, added a link somewhere to the new version, and voilà. Job done.

I was tasked with making these annual changes one year, so off I went, young, innocent and raring to make a difference to the world. When I was making my changes, I noticed something odd. There was a weird error with a field that wasn't anything to do with my changes. I fixed the bug, but then a thought occurred to me that made my heart sink…

I checked every previous version of the codebase for each previous year and found that nearly all of them had the same bug. It had been introduced about 10 years ago, and every devleoper since then had replicated the bug precisely. So, I had to fix it 10 times over, increasing the testing effort by an order of magnitude.

With this in mind, ask yourself - did copying and pasting really save you any time? I routinely work on apps that stay in existence for decades, and which show no sign of being put out to pasture any time soon.

When I decide where to make time-saving measures for coding work, I try and look over the whole life of the application, and try to keep in mind what the consequences might be for a decision a decade on.

To return to the subject at hand, how would I have used Higher-order functions to solve this problem? Well, are you sitting comfortably? Then I'll begin…

Thunks

A bundle of code that carries a stored calculation, which can be executed later on request is properly known as a *Thunk*. Same as the sound a plank of wood makes when it smacks you in the side of the head. There's an argument to be had as to whether that hurts your head more or less than reading this book!

Here in C#, `Func` delegates are again the way that we'd implement this. We can write functions that take `Func` delegates as parameter values, to allow for certain calculations in our function to be left effectively blank, and which can be filled in from the outside world, via an arrow function.

Although there is a serious, proper, mathematical term for this technique, I like calling them doughnut functions, because it's more descriptive. They're like normal functions, but with a hole in the middle! A hole I'd ask someone else to fill in with the necessary functionality.

This is one potential way to refactor the problem report function:

```
public void SendEnemyShipWeaponrySummary() =>
  GenerateSummary(x => x.Type, "Enemy Ship Type S

public void GenerateEnemyShipWeaponryLevelSummary
  GenerateSummary(x => x.WeaponryLevel, "Enemy Sh

private void GenerateSummary(Func<EnemyShip, stri
{
  try
  {
    var enemyShips = this.DataStore.GetEnemyShips
    var summaryNumbers = enemyShips.GroupBy(summa
                                    .Select(x =>
    var report = new Report
    {
      Title = reportName,
      Rows = summaryNumbers.Select(X => new Repor
      {
          ColumnOne = X.Type,
          ColumnTwo = X.Count.ToString()
      })
  };

  if (!report.Rows.Any())
   this.CommunicationSystem.SendNoDataWarning();
  else
   this.CommunicationSystem.SendReport(report);
  }
```

```
    catch (Exception e)
    {
      this.Logger.LogError(e,
      $"An error occurred in {nameof(GenerateSumma
    }
  }
```

In this, revised version, we’ve gained a few advantages.

Firstly, the number of additional lines per new report is just one! That’s a much tidier codebase, and easier to read. The code is kept very close to the intent of the new function - i.e. be the same as the first, but with a few changes.

Secondly, after unit testing function 1, when we create function 2, the unit test level is still close to 100%. The only difference functionally is the report name, and the field to be summarised.

Lastly, any enhancements or bug fixes to the base function will be shared between all report functions simultaniously. That’s a lot of benefit for relatively little effort. There’s also a very high degree of confidence that if one report function tests well, that all of the others will be the same.

One could actually walk away from this version happy. But, if it were me, I’d actually consider going a step further and exposing the private version with its `Func` parameters on the interface for whatever wants to use it.

Something like this:

```
public interface IGenerateReports
{
 void GenerateSummary(Func<EnemyShip, string> sur
}
```

The implementation would be the private function from the previous code sample made public instead. This way there's no need to ever modify the interface or implementing class again, at least not if all that's wanted is an additional report for a different field.

This makes the business of creating reports something that can be done entirely arbitrarily by whatever code module consumes this class. It saves a lot of the burdon of maintaining the report set from developers like ourselves, and puts it more in the hands of the teams that care about the reports themselves. Imagine the sheer number of Requests for Change that will now never need to come to a development team.

If you wanted to be really wild, you could expose further `Func` parameters as `Func<ReportLine,string>` to allow users of the report class to define custom formatting. You could also use `Action` parameters to allow for bespoke logging or event handling. This is just in my silly, made-up reporting class. The possibilities for the use of higher-order functions in this way are endless.

Despite being a functional programming feature, this is keeping us squarely in line with the *O* of the SOLID Principles of Object-Oriented design[2] - the Open/Closed principle, which states a module should be open to extension, but closed to modification.

It’s surprising how well OO and functional programming can complement each other in C#. I often think it’s important for developers to make sure they are adept at both paradigms, so they know how to use them together effectively.

Chaining Functions

Allow me to introduce you to the best friend you never knew you needed - the Map function. This function is also commonly referred to as Chain and Pipe, but for the sake of consistency, we’ll call it Map throughout this book. I’m afraid that there’s a tendency for a lot of functional structures to have many names in use, depending on the programming language and implementation. I’ll try and point out whenever this is the case.

Now, I’m British, and there’s a cliché about British people that we like talking about the weather. It’s entirely true. Our country is one that’s been known to go through 4 seasons in a single day, so the weather is a constant source of fascination to us.

I used to work for an American company, once upon a time, and when I did, the topic of conversation with my colleagues over video call would often turn inevitably to the subject of the weather. They'd tell me that the temperature outside was around 100 degrees. I work in celcius, so to me this sounds rather suspiciously like the boiling point of water. Given that my colleagues were not screamning as their blood boiled away into steam, I suspected something else was at work. It was, of course, that they were working in Fahrenheit, so I had to convert this to something I understood with the following formula:

- subtract 32
- Then, multiply by 5
- Then, divide by 9

Which gives a temperature in Celcius of around 38 degrees, which is warm and toasty, but for the most part safe for human life.

How could I code this process in exactly that multi-step operation, then finish by returning a formatted string? I *could* stick it all together into a single line like this:

```
public string FahrenheitToCelcius(decimal tempInF
  Math.Round(((tempInF-32) *5 / 9), 2) + "°C";
```

Not very readable though, is it? Honestly, I probably wouldn’t make too much fuss about that in production code, but I’m demonstrating a technique, and I don’t want to get bogged down, so bear with me.

The multi-step way to write this out is like this:

```
string FahrenheitToCelcius(decimal tempInF)
{
 var a = tempInF - 32;
 var b = a * 5;
 var c = b / 9;
 var d = Math.Round(c, 2);
 var returnValue = d + "°C";
 return returnValue;
}
```

This is much more readable, and easier to maintain, but it still has an issue. We’re creating variables that are essential intended to be used a single time and then thrown away. In this little function it’s not terribly relevant, but what if this were a gigantic thousand-line function? What if instead of a little decimal variable like these, there was a large, complex object instead? All the way down at line 1000, that variable - which is never intended to be used again - is still in scope, and holding up memory. It’s also a little messy to create a variable you aren’t planning to make any use of beyond the next line. This is where Map comes in.

Map is somewhat like the LINQ `Select` function, except instead of operating on each element of an Enumerable, it operates on an object. Any object. You pass it a Lambda arrow function just the same as with `Select` except that your *x* parameter refers to the base object. If you applied it to an Enumerable, the *x* parameter would refer to the entire enumerable, not indivudual elements thereof.

Here’s what my modified Fahrenheit conversion would look:

```
public string FahrenheitToCelcius(decimal tempIn
        tempInF.Map(x => x - 32)
                        .Map(x => x * 5)
                        .Map(x => x / 9)
                        .Map(x => Math.Round(x, 
                        .Map(x => x + "°C");
```

Same exact functionality, same friendly, multi-stage operation, but no throw-away variables. Each of those arrow functions is executed, then once completed, their contents are subject for Garbage Collection. The decimal *x* that is multiplied by 5 is subject for disposal when the next arrow function takes a copy of its result and divides that by 9.

Here’s how you implement Map:

```
public static class MapExtensionMethods
{
        public static TOut Map<TIn, TOut>(this TI
                f(@this);
}
```

It’s tiny, isn’t it? Despite that, I use this partiular method quite a lot. Whenever I want to do a multi-step transformaton of data. It makes it easier to convert whole function bodies into simple arrow functions, like my Map-based FahrenheitToCelcius function, above.

There are far more advanced versions of this method, which includes things like error handling, and which I’ll be getting onto in Chapter 7. For now though, this is a fantastic little toy that you can start playing with right away. Uncle Simon’s early Christmas gift to you. Ho, ho, ho.

There is a simpler implementation of Map possible, if you don’t want to change types with each transformation. This is cleaner and more concise, if it suits your needs.

It could be implemented like this:

```
public static T Map<T>(this T @this, params Func
 transformations.Aggregate(@this, (agg, x) => x(
```

Using that, the basic Fahrenheit to Celcius transformation would be something like this:

```
public decimal FahrenheitToCelcius(decimal tempI
        tempInF.Map(
         x => x - 32,
         x => x * 5,
         x => x / 9
         x => Math.Round(x, 2);
```

This might be worth using to save a little bit of boilerplate in simpler cases, like the temperature conversion. See Chapter 8 later on Currying for some ideas on how to make this look even better.

Fork Combinator

I’ve also heard this one called “Converge”. I like “Fork” though, it’s more descriptive of how exactly it works. A Fork combinator is used to taka a single value, then process it in multiple ways, simultaniously, then join up all of those separate strands into a single, final value. It can be used to simplify some fairly complex multi-step calculations into a single line of code.

The process is going to run roughly like this:

- Start with a single value
- Feed it into a set of “prong” functions - each of which act on the original input in isolation to produce some sort of output
- A “join” function takes the result of the prongs and merges it into a final result.

Here are a few examples of how I might use it.

If you want to specify the number of arguments in your function definition - rather than having an unspecified number of prongs from an array, then it’s possible to use a Fork to calculate an average value:

```
var numbers = new [] { 4, 8, 15, 16, 23, 42 }
var average = numbers.Fork(
 x => x.Sum(),
 x => x.Count(),
 (s, c) => s / c
);
// average = 18
```

or here’s a blast from the past, a Fork you can use to calculate the Hypotenuse of a triangle:

```
var triangle = new Triangle(100, 200);
var hypotenuse = triangle.Fork(
 x => Math.Pow(x.A, 2),
 x => Math.Pow(x.B, 2)
```

```
   x => Math.Pow(x.B, 2),
   (a2, b2) => Math.Sqrt(a2 + b2)
  );
```

The implementation looks like this:

```
public static class ext
{
        public static TOut Fork<TIn, T1, T2, TOut
          this TIn @this,
          Func<TIn, T1> f1,
          Func<TIn, T2> f2,
          Func<T1,T2,TOut> fout)
        {
                var p1 = f1(@this);
                var p2 = f2(@this);
                var result = fout(p1, p2);
                return result;
        }
}
```

Note that having two generic types, one for each prong, means that any combination of types can be returned by those functions.

You could easily go out and write versions of this for any number of parameters beyond two as well, but each additional parameter you want to consider would require an additional extension method.

If you wanted to go further, and have an unlimited number of “prongs”, then provided you are OK with having the same intermediate type generated by each, that’s easily done:

```
public static class ForkExtensionMethods
{
 public static TEnd Fork<TStart, TMiddle, TEnd>(
  this TStart @this,
  Func<TMiddle, TEnd> joinFunction,
  params Func<TStart, TMiddle>[] prongs
 )
{
 var intermediateValues = prongs.Select(x => x(@t
 var returnValue = joinFunction(intermediateValue
 return returnValue;
}
```

We could use this, for instance, to create a text description based on an object:

```
var personData = this.personRepository.GetPerson(
var description = personData.Fork(
 prongs => string.Join(Environment.NewLine, prong
 x => "My name is " + x.FirstName + " " + x.LastN
 x => "I am " + x.Age + " years old.",
 x => "I live in " + x.Address.Town
)

// This might, for example, produce:
//
// My name is Jean Valjean
// I am 30 years old
// I live in Montreuil-sur-mer
```

With this Fork example, it’s easy enough to add as many more lines of description as we want, but maintaining the same level of complexity, and readability.

Alt Combinator

I’ve also seen this referred to as “Or”, “Alternate” and “Alternation”. It’s used to bind together a set of functions to achieve the same end, but which should be tried one after the other until one of them returns a value.

Think of it as working like “Try method A, if that doesn’t work, try method B, if that doesn’t work, try method C, if that doesn’t work I suppose we’re out of luck”.

Let’s try and imagine a scenario where we might want to find something by trying multiple methods:

```
var jamesBond = "007"
 .Alt(x => this.hotelService.ScanGuestsForSpies(x
 x => this.airportService.CheckPassengersForSpies
 x => this.barService.CheckGutterForDrunkSpies(x)

 if(jamesBond != null)
  this.deathTrapService.CauseHorribleDeath(jamesB
```

- So long as one of those three methods returns a value corresponding to a hard-drinking, boderline-misogynist, thuggish employee of the British Government, then the jamesBond variable won’t be null. Whichever function returned a value first is the last function to be run.

So how do we implement this function before we find our enemy has scarpered? Like this:

```
public static TOut Alt<TIn, TOut>(this TIn @this,
        args.Select(x => x(@this))
        .First(x => x != null);
```

Remember here that the LINQ `Select` function operates on a lazy-loading principle, so even though I appear to be converting the whole of the `Func` array into concrete types, I’m not, because the `First` function will prevent any elements from being executed after one of them has returned a non-null value. Isn’t LINQ great?

Compose

A common feature of functional languages is the ability to build up a higher-order function from a collection of smaller functions. Any process that involves combinging functions is called *Composing*.

There are JavaScript libraries like RamdaJS[3] that have terrific composing features available, but C#’s strong typing actually works against it in this instance.

There are a few methods for composing functions in C#. The first is the most simple, just using basic Map functions, as described earlier in this chapter:

```
var input = 100M;
var f = (decimal x) => x.Map(x => x - 32)
  .Map(x => x * 5)
  .Map(x => x / 9)
  .Map(x => Math.Round(x, 2))
  .Map(x => $"{x} degrees");
var output = f(input);
// output = "37.78 degrees"
```

f here is a composed higher-order function. There are 5 functions (e.g. x ⇒ x - 32, those steps of the calculation) used to create it, which are described as anonymous lambda expressions. They combine, like lego bricks, to form a larger, more complex behavior.

A valid question at this point - What's the point of composing functions?

The answer is that you don't necesarily have to do the whole thing all in once. You could build it in pieces, and then ultimately create many functions using the same base pieces.

Imagine now that I want to also hold a `Func` delegate that represented the opposite conversion - we'd end up with two functions like this:

```
var input = 100M;
var fahrenheitToCelcius = (decimal x) => x.Map(x
  .Map(x => x * 5)
  .Map(x => x / 9)
  .Map(x => Math.Round(x, 2))
  .Map(x => $"{x} degrees");
var output = fahrenheitToCelcius(input);
Console.WriteLine(output);.
// 37.78 degrees

var input2 = 37.78M;
var celciusToFahrenheit =       (decimal x) =>
 x.Map(x => x * 9)
```

```
   .Map(x => x / 5)
   .Map(x => x + 32)
   .Map(x => Math.Round(x, 2))
   .Map(x => $"{x} degrees");
  var output2 = celciusToFahrenheit(input2);
  Console.WriteLine(output2);
  // 100.00 degrees
```

The last two lines of each function are actually identical. Isn't it a bit wasteful to repeat them each time? We can actually eliminate the repetition using a Compose function:

```
  var formatDecimal = (decimal x) => x
   .Map(x => Math.Round(x, 2))
   .Map(x => $"{x} degrees");

  var input = 100M;
  var celciusToFahrenheit = (decimal x) => x.Map(x
    .Map(x => x * 5)
    .Map(x => x / 9);
  var fToCFormatted = celciusToFahrenheit.Compose(f
  var output = fToCFormatted(input);
  Console.WriteLine(output);

  var input2 = 37.78M;
  var celciusToFahrenheit =       (decimal x) =>
   x.Map(x => x * 9)
   .Map(x => x / 5)
   .Map(x => x + 32);
  var cToFFormatted = celciusToFahrenheit.Compose(f
  var output2 = cToFFormatted(input2);
  Console.WriteLine(output2);
```

Functionally, these new versions using Compose are identical to the previous versions exclusively using Map.

The Compose function performs nearly the same task as Map, with the subtle difference that we're ultimately producing a `Func` delegate at the end, not a final value. This is the code that performs the Compose process:

```
  public static class ComposeExtensionMethods
  {
          public static Func<TIn, NewTOut> Compose<
           this Func<TIn, OldTOut> @this,
           Func<OldTOut, NewTOut> f) =>
                  x => f(@this(x));
  }
```

By using the Compose, we've eliminated some unnecessary replication. Any improvements to the Format process will be shared by both `Func` delegate objects simultaniously.

There is a limitation, however. In C#, extension methods can't be attached to lambda expressions or to functions directly. We can attach an extension to a lambda expression if it's referenced as a `Func` or `Action` delegate, but for that to happen it first needs to be assigned to a variable where it will be automatically set as a delegate type for us. This is why its necessary in the examples above to assigned the chains of `Map` functions to a variable before calling `Compose` - otherwise it would be possible to simply call `Compose` at the end of the `Map` chain and save ourselves a variable assignment.

This process is not unlike reusing code via inheritance in Object-Oriented programming, except it's done at the individual line level, and requires quite significantly less boilerplate to achieve. It also keeps these similar, related pieces of code together, rather than them having to necessarily be spread out over separate classes and files.

Transduce

A Transducer is a way of combining list-based operations, like Select and Where, with some form of aggregation to perform multiple transformations to a list of values, before finally collapsing it down into a single, final value.

While Compose is a useful feature, it has some limitations. It effectively only ever takes the place of a Map function - i.e. it acts on the object as a whole, and it can't perform LINQ operations on Enumerables. You *could* Compose an array and put Select and Where operations inside each, but honestly that looks pretty messy:

```
var numbers = new [] { 4, 8, 15, 16, 23, 42 };
var add5 = (IEnumerable<int> x) => x.Select(y =>
var Add5MultiplyBy10 = add5.Compose(x => x.Select

var numbersGreaterThan100 = Add5MultiplyBy10.Comp

var composeMessage = numbersGreaterThan100.Compos
Console.WriteLine("Output = " + composeMessage(nu
// Output = 130,200,210,280,470
```

If you're happy with that, the by all means use it. There's nothing wrong with it per se, aside from being rather inelegant.

There is another structure that we can use though - Transduce. A Transduce operation acts on an array and represents all of the stages of a functional flow:

- Filter (i.e. `.Where`) - Reduce the number of elements
- Transform (i.e. `.Select`) - Convert them to a new form
- Aggregate (i.e. Erm..actually is *is* Aggregate) - whittle down the collection of many items to a single item using these rules

There are many ways this could be implemented in C#, but here’s one possibility:

```
public static TFinalOut Transduce<TIn, TFilterOut
        this IEnumerable<TIn> @this,
        Func<IEnumerable<TIn>, IEnumerable<TFilte
        Func<IEnumerable<TFilterOut>, TFinalOut>
                aggregator(transformer(@this));
```

This extension method takes a transformer method - which can be any combination of `Select` and `Where` the user defines to transform the Enumerable ultimately from one form and size, to another. The method also takes an aggregator, which converts the output of the transformer into a single value.

This is how the compose function I defined above could be implemented with this version of the Transduce method:

```
var numbers = new [] { 4, 8, 15, 16, 23, 42 };

// N.B - I could make this a single line with bra
// I find this more readable, and it's functional
// to lazy evaluation of Enumerables
var transformer = (IEnumerable<int> x) => x
        .Select(y => y + 5)
        .Select(y => y * 10)
        .Where(y => y > 100);

var aggregator = (IEnumerable<int> x) => string.J

var output = numbers.Transduce(transformer, aggre
Console.WriteLine("Output = " + output);
// Output = 130, 200, 210, 280, 470
```

Alternatively, if you’d prefer to handle everything as `Func` delegates, so that you can reuse the Transducer function, it could be written in this way:

```
var numbers = new [] { 4, 8, 15, 16, 23, 42 };
var transformer = (IEnumerable<int> x) => x
        .Select(y => y + 5)
        .Select(y => y * 10)
        .Where(y => y > 100);

var aggregator = (IEnumerable<int> x) => string.J

var transducer = transformer.ToTransducer(aggrega
var output2 = transducer(numbers);
Console.WriteLine("Output = " + output2);
```

This is the updated extension method:

```
public static class TransducerExtensionMethod
{
        public static Func<IEnumerable<TIn>, New
                this Func<IEnumerable<TIn>,
                IEnumerable<OldTOut>> @this,
                Func<IEnumerable<OldTOut>, NewTO
                        x => aggregator(@this(x)
}
```

We’ve now generated a `Func` delegate variable that can be used as a function on as many arrays of integers as we want, and that single `Func` will perfom any number of transformations and filters required, then aggregate the array down to a single, final value.

Tap

A common concern I hear raised about chains of functions is that it’s impossible to perform logging within them - excepting that you make one of the links in the chain a reference to a separate function that does have logging calls within it.

There is a technique in functional programming that can be used to inspect the contents of a function chain at any point - a Tap function.

A Tap function is a bit like a wire tap[4] in old detective films. Something that allows a stream of information to be monitored and acted on, but without disrupting or altering it.

The way to implement a Tap is like this:

```
public static class Extensions
{
 public static T Tap<T>(this T @this, Action<T> a
 {
  action(@this);
  return @this;
}
```

An `Action` delegate is effectively like a void returning function. In this instance it accepts a single parameter - a generic type, T. The Tap function passes the current value of the object in the chain into the Action, where logging can take place, then returns an unmodified copy of that same object.

You could use it like this:

```
var input = 100M;
var fahrenheitToCelcius = (decimal x) => x.Map(x
  .Map(x => x * 5)
  .Map(x => x / 9)
```

```
    .Tap(x => this.logger.LogInformation("the un-ro
    .Map(x => Math.Round(x, 2))
    .Map(x => $"{x} degrees");
  var output = fahrenheitToCelcius(input);
  Console.WriteLine(output);
  // 37.78 degrees
```

In this new version of the Fahrenheit to Celcius functional chain, I'm now Tapping into it after the basic calculation is completed, but before I start rounding and formatting to string.

I added a call to a logger in the Tap, but you could switch that for a `Console.WriteLine` or whatever else you'd like.

Try/Catch

There are several more advanced structures in functional programming for handling errors. If you just want something quick and easy you can quickly implement in a few lines of code, but which has its limitations, keep reading. Otherwise, try having a look ahead at the next chapter on Discriminated Unions, and the chapter after on advanced functional structures. There's plenty to be found there on handling errors without side effects.

For now though, let's see what we can do with a few simple lines of code…

In theory, in the middle of functional-style code, there shouldn't be any errors possible. If everything is done in line with the functional principles of side-effect free code, immutable variables, etc. then you should be safe. On the fringes though, there are always interactions that might be considered unsafe.

Let's imagine you have a scenario in which you want to run a lookup in an external system with an integer Id. This external system could be a database, a web api, a flat file on a network share, anything at all. The thing all of these possibilities have in common though, is that any of them can fail for many reasons, few, if any, of which are the fault of the devleoper.

There could be network issues, hardware issues on the local or remote computers, inadvertent human intervention. The list goes on…

This is how you'd usually deal with that situation in Object Oriented code:

```
pubic IEnumerable<Snack> GetSnackByType(int typeI
{
 try
 {
  var returnValue = this.DataStore.GetSnackByType
  return returnValue;
 }
 catch(Exception e)
 {
  this.logger.LogError(e, $"There aren't any pork
  return Enumerable.Empty<Snack>()
 }
}
```

There are two things I dislike about this code block. The first is how much boilerplate we have to bulk out the code with. There’s a lot of industrial-strength coding we have to add to protect ourselves from problems that we didn’t cause.

The other issue is with try/catch blocks themselves. It breaks the order of operations, moving execution of the program from where we were to some potentially hard-to-find location. In this case, it’s a nice, simple, compact little function and the location of the Catch is easy to determine. I’ve worked in codebases though where the Catch was several layers of functions higher than the place the fault occurred. Bugs were common in that codebase because assumptions were made about certains lines of code being reached when they weren’t due to the strange positioning of the Try/Catch block.

I probably wouldn’t honestly have too many issues with the code block above in production, but left unchecked, bad coding practices can leak in. There’s nothing in the code that’s preventing future coders from introducing multi-level nested functions in here.

I think the best solution is to use an approach that removes all of the boilerplate, and makes it hard, or even impossible, to introduce bad code structure later.

Something like this:

```
pubic IEnumerable<Snack> GetSnackByType(int typeI
{
 var result = typeId.MapWithTryCatch(this.DataSto
   ?? Enumerable.Empty<Snack>();
 return result;
}
```

What I'm doing is running a Map function with an embedded Try/Catch. The new Map function either returns a value if everything worked, or `null` if there was a failure.

The extension method looks like this:

```
public static class Extensions
{
 public static TOut MapWithTryCatch<TIn,TOut>(thi
 {
  try
  {
   return f(@this);
```

```
    }
    catch()
    {
     return default;
    }
   }
  }
```

This isn't quite a perfect solution though. What about error logging? This is committing the cardinal sin of swallowing error messages unlogged.

There are a few ways you could think about solving this. Any of these are equally fine, so proceed as your fancy takes you.

One option is to instead have an extension method that takes an ILogger instance to return a `Func` delegate containing the Try/Catch functionality. Something like this:

```
  public static class TryCatchExtensionMethods
  {
   public static TOut CreateTryCatch<TIn,TOut>(this
   {
    Func<TIn,TOut> f =>
    {
     try
     {
      return f(@this);
     }
     catch(Exception e)
     {
      logger.LogError(e, "An error occurred");
      return default;
     }
    }
   }
  }
```

The usage would be pretty similar:

```
  public IEnumerable<Snack> GetSnackByType(int type
  {
   var tryCatch = typeId.CreateTryCatch(this.logge
   var result = tryCatch(this.DataStore.GetSnackBy
     ?? Enumerable.Empty<Snack>();
   return result;
  }
```

Only a single additional line of boilerplate added, and now logging is being done. Sadly there isn’t anything specific we can add in the message besides the error itself. The extension method doesn’t know where it’s called from, or the context of the error, which is perfect for re-using the method all over the codebase.

If you don't want the try/catch being aware of the `ILogger` interface, or you want to provide a custom error message every time, then we need to look at something a little more complicated to handle error messaging.

One option is to return a meta-data object which contains the return value of the function that's being executed, and a bit of data about whether things worked, whether there were errors and what they were. Something like this:

```
public class ExecutionResult<T>
{
 public T Result { get; init; }
 public Exception Error { get; init; }
}

public static class Extensions
{
 public static ExtensionResult<TOut> MapWithTryCa
 {
  try
  {
   var result = f(@this);
   return new ExecutionResult<TOut>
   {
    Result = result
```

```
     };
    }
    catch(Exception e)
    {
     return new ExecutionResult<TOut>
     {
      Error = e
     };
    }
  }
 }
```

I don’t really like this approach. It’s breaking one of the SOLID principles of Object-Oriented design - the Interface Segregation Principle. Well sort of. Techically that applies to Interfaces, but I try to apply it everywhere. Even if I do write functional code. The idea is that we shouldn’t be forced to include something in a class or interface that we don’t actually need. Here, we’re forcing a successful run to include an `Exception` property that it’ll never need, and likewise, a failure run will have to include the Result property it’ll never need.

There are other ways you could do this, but I’m making it simple, and either returning a version of the `ExecutionResult` class with the result, or else a `default` value of Result and the Exception returned.

This means I can call it like this:

```
pubic IEnumerable<Snack> GetSnackByType(int typeI
{
 var result = typeId.MapWithTryCatch(this.DataSto
 if(result.Value == null)
 {
  this.Logger.LogException(result.Error, "We ran
  return Enumerable.Empty<Snack>();
 }

 return result.Result;
}
```

The unnecessary fields aside, there's another issue with this approach - the onus is now on the developer using the Try/Catch function to add additional boilerplate to check for errors.

Skip ahead to the next chapter for an alternative way of handling this sort of return value in a more purely functional manner. For now though, here's a slightly cleaner way of handling it.

First, I'll add in another extension method. One that attaches to the ExecutionResult object this time:

```
public static T OnError<T>(this ExecutionResult<T
 {
if (@this.Error != null)
```

```
  errorHandler(@this.Error);
  return @this.Result;
   }
```

What I’m doing here is first checking whether there’s an error. If there is, then execute the user-defined `Action` - which will presumably be a logging operation. It finishes by unwrapping the ExecutionResult into just its actual returned data object.

All of that means you can now handle the Try/Catch like this:

```
  public IEnumerable<Snack> GetSnackByTypeId(int ty
          typeId.MapWithTryCatch(DataStore.GetSnack
                  .OnError(e => this.Logger.LogErro
```

It’s far from a perfect solution, but without moving on another level in functional theory, it’s workable and elegant enough that it’s not setting off my internal perfectionist. It also forces the user to consider error handling when using this, which can only be a good thing!

Handling Nulls

Aren’t null reference exceptions annoying? If you want someone to blame, it’s a guy called Tony Hoare who invented the concept of Null back in the

60s. Actually, let’s not blame anyone. I’m sure he’s a lovely person, beloved by everyone that knows him. In any case, we can hopefully all agree that null reference exceptions are an absolute pain in the preverbial.

So, is there a functional way to deal with them? If you’ve read this far, you probably know that the answer will be a resounding “yes!”[5].

The *Unless* function takes in a boolean condition and an `Action` delegate, and only executes the `Action` if the boolean is false - i.e. the `Action` is always executed *unless* the condition is true.

The most common usage for something like this is - you guessed it - checking for null.

Here’s an example of exactly the sort of code I’m trying to replace. This is a rarely-seen bit of source code for a Dalek[6]:

```
public void BusinessAsUsual()
{
 var enemies = this.scanner.FindLifeforms('all')
 foreach(var e in enemies)
 {
  this.Gun.Blast(e.Coordinates.Longitude, e.Coor
  this.Speech.ScreamAt(e, "EXTERMINATE");
 }
}
```

This is all well and good, and probably leaves a lot of people killed by a psychotic mutant in a mobile pepper-pot shaped tank. But, what if the Coordinates object was null for some reason? That's right - null reference exception.

This is where we make this functional, and introduce an Unless function to prevent the exception from occuring. This is what unless looks like:

```
public static class UnlessExtensionMethods
{
 public void Unless<T>(this T @this, Func<bool> c
 {
  if(!condition(@this)
  {
   f(@this);
  }
 }
}
```

It has to be a void, unfortunately. If we swapped the `Action` for a `Func`, then it's fine to return the result of the `Func` from the extension method. What about when condition is true though, and we don't execute? What do I return then? There isn't really an answer to that question.

This is how I'd use it to make my new, super-duper, even more deadly functional Dalek:

```
public void BusinessAsUsual()
{
 var enemies = this.scanner.FindLifeforms('all')

 foreach(var e in enemies)
 {
  e.unless(
   x => x.Coordinates == null,
   x => this.Gun.Blast(e.Coordinates.Longitude, e
  )

 // May as well do this anyway, since we're here
  this.Speech.ScreamAt(e, "EXTERMINATE");
 }
}
```

Using this, a null Coordinates object won’t result in an exception, the gun simply won’t be fired.

There are more ways to prevent null exceptions coming over the next few chapters - ways that require more advanced coding and a little theory, but which are much more thorough in the way they work. Stay tuned.

Update an Enumerable

I’m going to finish off this section with a useful. It involves updating an element in an Enumerable without changing any data at all!

The thing to remember about enumerables is that they are designed to make use of “lazy evaluation” - i.e. they don’t actually convert from a set of functions pointing at a data source to actual data until the last possible moment. Quite often, the use of `Select` functions doesn’t trigger an evaluation, so we can use them to effectively create filters sitting between the data source and the place in the code in which enumeration of the data will actually take place.

Here’s an example of altering an Enumerable, so that the item at position x is replaced:

```
var sourceData = new []
{
 "Hello", "Doctor", "Yesterday", "Today", "Tomor
}

var updatedData = sourceData.ReplaceAt(1, "Darkne
var finalString = string.Join(" ", updatedData);
// Hello Darkness, my old friend Yesterday Today
```

What I’ve done is call a function to replace the element at position 1 (i.e. “Doctor”) with a new value. Despite having two variables, nothing is actually done to the source data at all. The variable SourceData remains the same after this code snippet has come to the end. Further to that, no replacement is actually made until calling `string.Join`, because that's the very moment at which concrete values are required.

This is how it's done:

```
public static class Extensions
{
  public static IEnumerable<T> ReplaceAt(this IE
    int loc,
    T replacement) =>
    @this.Select((x, i) => i == loc ? replacement
}
```

This Enumerable, returned here, actually points at the original Enumerable and gets its values from there, but with one crucial difference. If the index of the element ever equals the user-defined value (1, the second element, in our example). All other values are passed through, unaltered.

If you were so inclined, you could provide a function to perform the update - giving the user the ability to base the new version of the data item on the old that is being replaced.

This is how you'd achieve that:

```
public static class Extensions
{
  public static IEnumerable<T> ReplaceAt(this IEr
    int loc,
    Func<T, T> replacement) =>
    @this.Select((x, i) => i == loc ? replacement
}
```

Easy enough to use too:

```
var sourceData = new []
{
 "Hello", "Doctor", "Yesterday", "Today", "Tomorr
}

var updatedData = sourceData.ReplaceAt(1, x => x
var finalString = string.Join(" ", updatedData);
// Hello Doctor Who Yesterday Today Tomorrow Cont
```

It’s also possible that we don’t know the Id of the element we want to update - in fact there could be multiple items to update. This is an alternative Enumerable update function based on providing a T to Bool converting `Func` to identify the records that should be updated.

This example is based on board games - one of my favorite hobbies - much to the annoyance of my ever-patient wife! In this scenario there is a Tag property on the BoardGame object, which contains meta data tags describing the game (“family”, “co-op”, “complex”, stuff like that) which will be used by a search engine app. It’s been decided that another tag should be added to games suitable for 1 player - “solo”.

```
var sourceData = this.DataStore.GetBoardGames();

var updatedData = sourceData.ReplaceWhen(
                x => x.NumberOfPlayersAllowed.Co
                x => x with { Tags = x.Tags.Appe
this.DataStore.Save(updatedData);
```

The implementation is a variation on code we’ve already covered:

```
public static class ReplaceWhenExtensions
{
 public static IEnumerable<T> ReplaceWhen<T>(thi
  Func<T, bool> shouldReplace,
  Func<T, T> replacement) =>
  @this.Select(x => shouldReplace(x) ? replaceme
}
```

This function can be used to replace the need for many instances of If-statements, and reduce them down to simpler, more predictable operations.

Conclusion

In this chapter we looked at various ways to use the concept of higher-order functions to develop ways to provide rich functionality to our codebase, avoiding the need for Object-Oriented style statements.

Do get in touch if you have any ideas of your own for higher-order function uses. You never know, it might end up in a future edition of this book!

In the next chapter, we'll be looking at Discriminated Unions, and how this functional concept can help to better model concepts in your codebase, and remove the need for a lot of defensive code typically needed with non-functional projects. Enjoy!

Ideally the hotest, spiciest flavor you can find. Flames should be shooting from your mouth as you eat!

Read more about it here: *https://en.wikipedia.org/wiki/SOLID* - or if you prefer video then here's one presented by yours truly: *https://www.youtube.com/watch?v=0vJb_B47J6U*

See it for yourself here: *https://ramdajs.com/*

I'd guess that's where they get their name

Also, congratulations for making it this far. Although it probably didn't take you anywhere so much time as it did me!!

For the non-initiated, these are the main baddies in the British SF TV series Doctor Who. See them in action here: *https://www.youtube.com/watch?v=d77jOE2Cjx8*

Chapter 6. Discriminated Unions

Discriminated unions (DUs) are a way of defining a type (or class in the OO world) that is actually one of a set of different types. Which type an instance of a DU actually is at any given moment has to be checked before use.

F# has DUs available natively, and it's a feature used extensively by F# developers. Despite sharing a common runtime with C#, and the feature being there for us *in theory*, there are only plans in place to introduce them into C# at some point - but it's not certain how or when. What we can do in the meantime is roughly simulate them with abstract classes, and that’s the technique I’m going to talk about in this chapter.

This chapter is our very first dabble into some of the more advanced areas of functional programming. Earlier chapters in the book were more focused on how you, the developer can work smart, not hard. We’ve also looked at ways to reduce boilerplate, and to make code more robust and maintainable.

Discriminated unions[1] are a programming structure that will do all of this too, but are more than a simple extension method, or a single-line fix to remove a little bit of boilerplate. DUs are closer in concept to a design pattern - in that they have a structure, and some logic that needs to be implemented around it.

Holiday Time

Let’s imagine an old-school Object-Oriented problem where we’re creating a system for package holidays. You know - the sort where the travel agency arrange your travel, accomodation, etc. all in one. I’ll leave you to imagine what lovely destination you’re off too. Personally, I’m quite fond of the Greek islands.

```
public class Holiday
{
 public int Id { get; set; }
```

```
  public Location Destination { get; set; }
  public Location DepartureAirport { get; set; }
  public DateTime StartDate { get; set; }
  public int DurationOfStay { get; set; }
 }

 public class HolidayWithMeals : Holiday
 {
  public int NumberOfMeals { get; set; }
 }
```

Now imagine we were creating, say, an account page for our customers[2], and we want to list everything they've bought so far. Not all that difficult, really. We can use some relatively-new `is` statement to build the necessary string. Here's one way we could do it:

```
 public string formatHoliday(Holiday h) =>
  "From: " + h.DepartureAirport.Name + Environment
  "To: " + h.Destination.Name + Environment.NewLin
  "Duration: " + h.DurationOfStay + " Day(s)" +
  (
   h is HolidayWithMeals hm
    ? Environment.NewLine + "Number of Meals: " +
    : string.Empty
  );
```

If I wanted to quickly improve this with a few functional ideas, I could consider introducing a Fork combinator (see previous chapter), The basic type is Holiday, the sub-type is a Holiday with a Meal. Essentially the same thing, but with an extra field or two.

What if…there was a project started up in the company. Now, they're going to start offering other types of service, separate from holidays. They're going to also start providing day trips that don't involve hotels, flights, or anything else of that sort. Entrance into Tower Bridge[3] in London, perhaps. Or a quick jaunt up the Eiffel Tower in Paris. Whatever you fancy. World's your oyster.

The object would look something like this:

```
public class DayTrip
{
 public int Id { get; set; }
 public DateTime DateOfTrip { get; set; }
 public Location Attraction { get; set; }
 public bool CoachTripRequired { get; set; }
}
```

The point is though, that if we want to represent this new scenario with inheritance from a Holiday object, it doesn't work. An approach I've seen some people follow is to merge all of the fields together, along with a boolean to indicate which fields are the ones you should be looking at.

Something like this:

```
public class CustomerOffering
{
  public int Id { get; set; }
  public Location Destination { get; set; }
  public Location DepartureAirport { get; set; }
  public DateTime StartDate { get; set; }
  public int DurationOfStay { get; set; }
  public bool CoachTripRequired { get; set; }
  public bool IsDayTrip { get; set; }
}
```

This is a poor idea for several reasons. For one, you're breaking the Interface Segregation principle. Whichever type it really is, you're forcing it to hold fields that are irrelevant to it. We've also doubled up the concepts of "Destination" and "Attraction", as well as "DateOfTrip and "StartDate" here, to avoid duplication, but it means that we've lost some of the terminology that makes code dealing with day trips meaningful.

The other option is to maintain them as entirely separate types of object with no relationship between them at all. Doing that though, we lose the ability to have a nice, concise, simple loop through every object. We wouldn't be able to list everything in a single table in date order. There would have to be multiple tables.

None of the possibilities seem all that good. But this is where DUs come charging to the rescue. In the next section, I’ll show you how to use them to provide an optimum solution to this problem.

Holidays with Discriminated Unions

In F#, you can create a union type for our customer offering example, like this:

```
type CustomerOffering =
 | Holiday
 | HolidayWithMeals
 | DayTrip
```

What that means is that you can instantiate a new instance of CustomerOffering, but there are three separate types it could be, each potentially with their own entirely different properties.

This is the nearest we can get to this approach in C#:

```
public abstract class CustomerOffering
{
 public int Id { Get; set; }
}
```

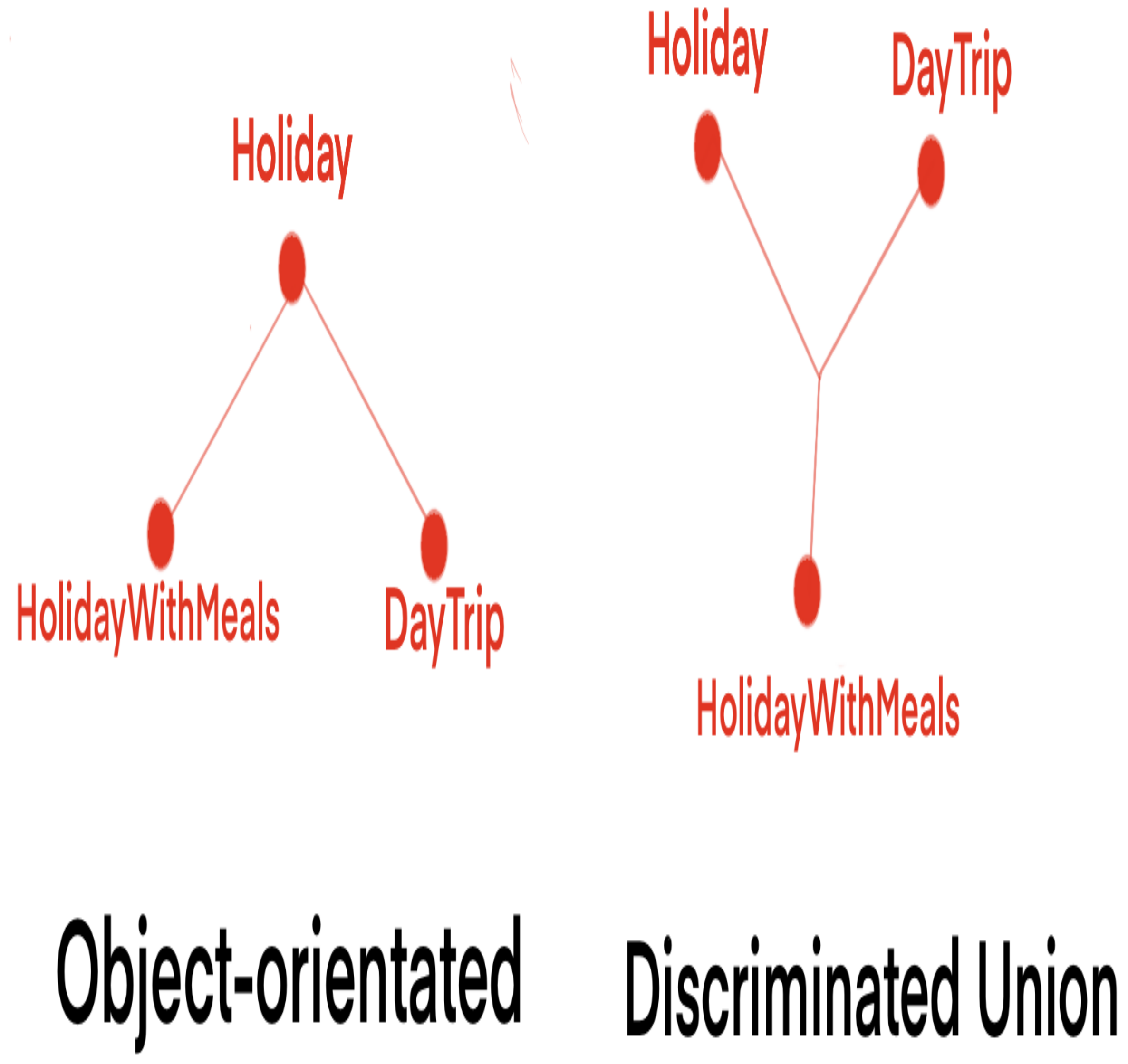
```
public class Holiday : CustomerOffering
{
 public Location Destination { get; set; }
 public Location DepartureAirport { get; set; }
 public DateTime StartDate { get; set; }
 public int DurationOfStay { get; set; }
}

public class HolidayWithMeals : Holiday
{
 public int NumberOfMeals { get; set; }
}

public class DayTrip : CustomerOffering
{
 public DateTime DateOfTrip { get; set; }
 public Location Attraction { get; set; }
 public bool CoachTripRequired { get; set; }
}
```

On the face of it, it doesn’t seem entirely different to the first version of this set of classes, but there’s an important difference. The base is abstract - you can’t actually create a CustomerOffering class. Instead of being a family tree of classes with one parent at the top that all others conform to, all of the sub-classes are different, but equal in the hiararchy.

Here’s a class hiararchy diagram, which makes the difference between the two approaches clearer:



The DayTrip class is in no way forced to conform to any concept that makes sense to the Holiday class. DayTrip is completely its own thing. It means it can use property names that correspond exactly to its own business logic, rather than having to retro-fit a few of the properties from Holiday. In other words - DayTrip isn't an **extension** of Holiday, it's an **alternative** to it.

This also means you can have a single array of all CustomerOfferings, even though they're wildly different. No need for separate data sources.

We'd handle an array of CustomerOffering objects in code using a pattern matching statement:

```
public string formatCustomerOffering(CustomerOffe
 c switch
 {
  HolidayWithMeals hm => this.formatHolidayWithMe
  Holiday h => this.formatHoliday(h),
  DayTrip dt => this.formatDayTrip(tp)
 };
```

This simplifies the code everywhere the discriminated union is received, and gives rise to more descriptive code, and code that more accurately describes all of the possible outcomes of a function.

Schrödinger’s Union

If you want an analogy of how these things work, think of poor old Schrödinger’s Cat. This was a thought experiment proposed by an Austrian physisist Erwin Schrödinger to highlight a paradox in quantum mechanics. The idea was that given a box containing a cat[4] and a radioactive isotope that had a 50-50 chance of decaying, which would kill the cat. The point was that, according to quantum physics[5] until someone opens the box to check on the cat, both states - alive and dead - exist at the same time. Meaning that the cat is both alive and dead at the same time.

This also means that if Herr Schrödinger were to send his cat/isotope box in the post to a friend[6] they have a box that could contain one of two states inside, and until they open it, they don’t know which. Of course, the postal service being what it is, chances are the cat would dead upon arrival *either way*. This is why you really shouldn’t try this one at home. Trust me I’m not a Doctor, nor do I play one on TV.

That’s kind’ve how a discriminated union is. A single returned value, but which may exist in two or more states. You don’t know which until you examine it.

If a class doesn’t care which state, you can even pass it along to its next destination unopened.

Schrödinger's cat as code might look like this:

```
public abstract class SchrödingersCat { }

public class AliveCat : SchrödingersCat { }

public class DeadCat : SchrödingersCat { }
```

I'm hoping you're now clear on what exactly Discriminated Unions actually **are**. I'm going to spend the rest of this chapter demonstrating a few examples of what they are **for**.

Naming Conventions

Let's imagine a code module for writing out people's names from the individual components. If you have a traditional British name, like my own, then this is fairly straightforward. A class to write a name like mine would look something like this:

```
public class BritishName
{
 public string FirstName { get; set; }
 public IEnumerable<string> MiddleNames { get; se
 public string LastName { get; set; }
 public string Honorific { get; set; }
}

var simonsName = new BritishName
{
 Honorific = "Mr.",
 FirstName = "Simon",
 MiddleNames = new [] { "John" },
 LastName = "Painter
};
```

The code to render it would be as simple as this:

```
public string formatName(BritishName bn) =>
 bn.Honorific + " " bn.FirstName + " " + string..
 " " + bn.LastName;
// Results in "Mr Simon John Painter"
```

All done, right? Well, this works for traditional British names, but what about Chinese names? They aren't written in the same order as British names. Chinese names are written in the order <family name> <given name>, and many Chinese people take a "courtesy name" - a western-style name, which is used professionally.

Let's take the example of the legendary actor, directory, writer, stunt-man, singer & all-round awesome Human being - Jackie Chan. His real name is

Fang Shilong. In that set of names, his family name (i.e. Surname) is Fang. His personal name (often in English called the First name, or Christian Name) is Shilong. Jackie is a courtesy name he's used since he was very young. This style of name doesn't work whatsoever with the formatName function I created, above.

I *could* mangle the data a bit to make it work. Something like this:

```
var jackie = new BritishName
{
 Honorific = "Xiānsheng", // equivalent of "Mr."
 FirstName = "Fang",
 LastName = "Shilong"
}
// results in "xiānsheng Fang Shilong"
```

So fine, this correctly writes his two official names in the correct order. What about his courtesy name though? There's nothing to write that out. Also, The Chinese equivalent of "Mr" - Xiānsheng [7] - actually goes **after** the name, so this is really pretty shoddy - even if we try re-purposing the existing fields.

We could add an awful lot of `if` statements into the code to check for the nationality of the person being described, but that approach would rapidly turn into a nightmare if we tried to scale it up to include more than 2 nationalities.

Once again, a better approach would be to use a discriminated union to represent the radically different data structures in a form that mirrors the reality of the thing they're trying to represent.

```
public abstract class Name { }

public class BritishName : Name
{
 public string FirstName { get; set; }
 public IEnumerable<string> MiddleNames { get; se
 public string LastName { get; set; }
 public string Honorific { get; set; }
}

public class ChineseName : Name
{
 public string FamilyName { get; set; }
 public string GivenName { get; set; }
 public string Honorific { get; set; }
 public string CourtesyName { get; set; }
}
```

In my imaginary scenario there are probably separate data sources for each name type - each with their own schema. Maybe a Web API for each country?

Using this union, we can actually create an array of names containing both me and Jackie Chan[8]

```
var names = new Name[]
{
 new BritishName
 {
  Honorific = "Mr.",
  FirstName = "Simon",
  MiddleNames = new [] { "John" },
  LastName = "Painter"
 },
 new ChineseName
 {
  Honorific = "Xiānsheng",
  FamilyName = "Fang",
  GivenName = "Shilong",
  CourtestyName = "Jackie"
 }
}
```

I coud then extend out my formatting function with a pattern matching expression:

```
public string formatName(Name n) =>
 n switch
 {
  BritishName bn => bn.Honorific + " " bn.FirstNa
     + string.Join(" ", bn.MiddleNames) + " " + b
  ChineseName cn => cn.FamilyName + " " + cn.Give
     cn.Honorific + " \"" + cn.CourtesyName + "\'
 };

var output = string.Join(Environment.NewLine, nar
// output =
// Mr. Simon John Painter
// Fang Shilong Xiānsheng "Jackie"
```

This same principle can be applied to any style of naming for anywhere in the world, and the names given to fields will always be meaningful to that country, as well as always being correctly styled without re-purposing existing fields.

Database Lookup

The sort of area of a system I'd often consider using Discriminated Unions in C# is as return types from functions defined on an interface.

One area I'm especially likely to use this technique is in lookup functions to data sources. Let's imagine for a moment you wanted to find someone's details in a system of some kind somewhere. The function is going to take an integer Id value, and return a Person record.

At least that's what you'd often find people doing. Something like this:

```
public Person  GetPerson(int id)
{
 // Fill in some code here.  Whatever data
 // store you want to use.  Except mini-disc.
}
```

But if you think about it, returning a Person object is only *one* of the possible return states of the function.

What if an Id is entered for a person that doesn't exist? You *could* return `Null` I suppose, but that's not descriptive of what actually happened. What if there were a handled `Exception` that resulted in nothing being returned? The `Null` doesn't tell you *why* it was returned.

The other possibility is an `Exception` being raised. It might well not be the fault of your code, but nevertheless it could happen if there are network issues, or whatever. What would you return in this case?

Rather than returning an unexplained `Null` and forcing other parts of the codebase to handle it, or an alternative return type object with metadata fields in it containing Exceptions, etc. we could instead create a discriminated union:

```
public abstract class PersonLookupResult
{
 public int Id { get; set; }
}

public class PersonFound : PersonLookupResult
{
 public Person Person { get; set; }
}

public class PersonNotFound : PersonLookupResult
{

}

public class ErrorWhileSearchingPerson : PersonLo
{
 public Exception Error { get; set; }
}
```

All of this means we can now return a single class from our GetPersonById function which tells the code utilizing the class that one of these three states has been returned, but which it is has already been determined. There’s no need for the returned object to have logic applied to it, to determine whether it worked or not, and the states are completely descriptive of each case that needs to be handled.

The function would look something like this:

```
public PersonLookupResult  GetPerson(int id)
{
 try
 {
  var personFromDb = this.Db.Person.Lookup(id);
  return personFromDb == null
   ? new PersonNotFound { Id = id }
   : new PersonFound
    {
     Person = personFromDb,
     Id = id
    };
 }
 catch(Exception e)
 {
  return new ErrorWhileSearchingPerson
  {
   Id = id,
   Error = e
  }
 }
}
```

And consuming it is once again a matter of using a pattern matching expression to determine what to do:

```
  public string DescribePerson(int id)
  {
   var p = this.PersonRepository.GetPerson(id);
   return p switch
   {
    PersonFound pf => "Their name is " + pf.Name,
    PersonNotFound _ => "Person not found",
    ErrorWhileSearchingPerson e => "An error occur
   };
  }
```

Sending Email

The last example was fine for cases where you’re expecting a value back, but what about cases where there’s no return value? Let’s imagine I’d written some code to send an email to a customer, or family member I can’t be bothered to write a message to myself[9].

I don’t expect anything back, but I might like to know if an error has occured, so this time there are only two states I’m especially concerned with.

This is how I’d accomplish it:

```
public abstract class EmailSendResult
{

}

public class EmailSuccess : EmailSendResult
{

}

public class EmailFailure : EmailSendResult
{
 pubic Exception Error { get; set; }
}
```

Use of this class in code might look like this:

```
public EmailSendResult SendEmail(string recipient
{
 try
 {
  this.AzureEmailUtility.SendEmail(recipient, mes
  return new EmailSuccess();
 }
 catch(Exception e)
 {
  return new EmailFailure
  {
   Error = e
  };
 }
}
```

Usage of the function elsewhere in the codebase would look like this:

```
var result = this.EmailTool.SendEmail("Season's (

var messageToWriteToConsole = result switch
{
 EmailFailure ef => "An error occurred sending th
 EmailSuccess _ => "Email send successful",
 _ => "Unknow Response"
};
```

```
this.Console.WriteLine(messageToWriteToConsole);
```

This, once again, means I can return an error message and failure state from the function, but without anything anywhere depending on properties it doesn’t need.

Console Input

Some time ago I came up with the mad idea to try out my functional programming skills by converting an old text-based game written in HP Timeshare BASIC to functional-style C#.

The game was called Oregon Trail, and dated all the way back to 1975. Hard as it is to believe, even older than I am! Older even, than Star Wars. In fact, it even predates monitors, and had to effectively be played on something that looked like a typewriter. In those days, when the code said “print” - it meant it!

One of the most crucial things the game code had to do was to periodically take input from the user. Most of the time an integer was required - either to select a command from a list, or to enter an amount of goods to purchase. Other times, it was important to receive text, and to confirm what it was the user typed - such as in the hunting mini-game, where the user was required to type “BANG” as quickly as possible to simulate attempting to accurately hit a target.

I *could* have simply had a module in the codebase that returned raw user input from the console. This would mean that every place in the entire codebase that required an integer value would be required to carry out a check, then a parse to integer, before getting on with whatever logic was actually required.

A smarter idea is to use a discriminated union to represent the different states the logic of the game recognises from user input, and keep the necessary int check code in a single place.

Something like this:

```
public abstract class UserInput
{

}


public class TextInput : UserInput
{
 public string Input { get; set; }
}

public class IntegerInput : UserInput
{
```

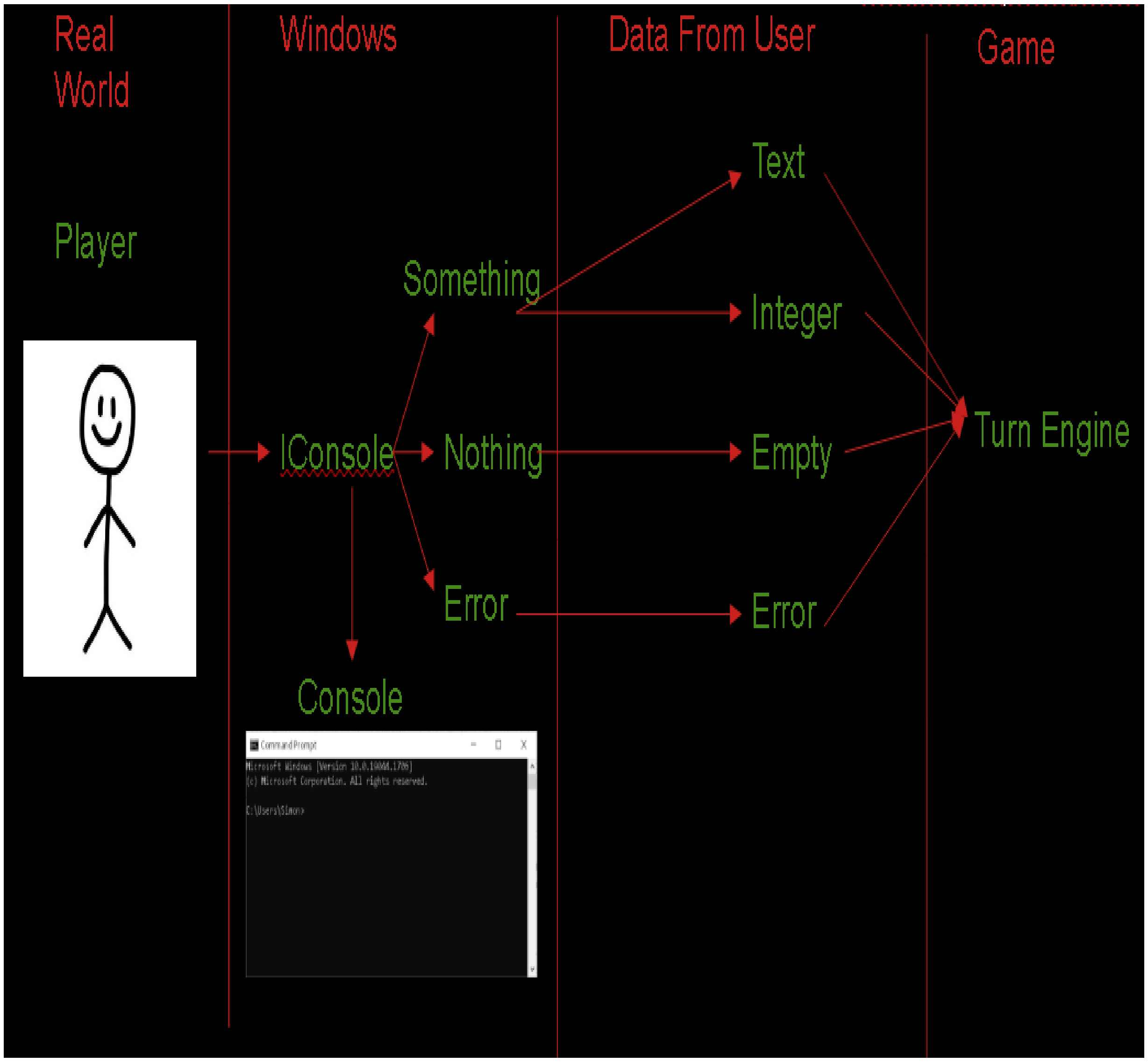
```
{
 public int Input { get; set; }
}

public class NoInput : UserInput
{
}

public class ErrorFromConsole : UserInput
{
 public Exception Error { get; set; }
}
```

I’m not honestly sure what errors are possible from the console, but I don’t think it’s wise to rule it out, especially as it’s something beyond the control of our application code.

The idea here is that I’m gradually shifting from the impure area beyond the codebase, into the pure, controlled area within it. Like a multi-stage airlock.



```
Microsoft Windows [Version 10.0.19044.1706]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Simon>
```

Speaking of the console being beyond our control… If we want to keep our codebase as functional as possible, then it’s best to hide it behind an interface, so that we can inject mocks during test, and push back the non-pure area of our code a little further.

Something like this:

```
public interface IConsole
{
 UserInput ReadInput(string userPromptMessage);
}

public class ConsoleShim : IConsole
{
 public UserInput ReadInput(string userPromptMess
 {
  try
  {
   Console.WriteLine(userPromptMessage);
   var input = Console.ReadLine();
   return new TextInput
   {
    Input = input
   };
  }
  catch(Exception e)
  {
   return new ErrorFromConsole
   {
    Error = e
   };
  }
 }
}
```

That was the most basic representation possible of an interaction with the user. That's because that's an area of the system with side effects, and I want to keep that as small as possible.

After that, I created another layer, but this time there was actually some logic applied to the text received from the player:

```
public class UserInteraction
{
 private readonly IConsole _console;
 public UserInteraction(IConsole console)
 {
  this._console = console;
 }

public UserInput GetInputFromUser(string message
{
        var input = this._console.ReadInput(messa
        var returnValue = input switch
        {
                TextInput x when string.IsNullOrW
                 new NoInput(),
                TextInput x when int.TryParse(x.I
                 new IntegerInput
                 {
                         Input = int.Parse(x.Inpu
                 },
                TextInput x => new TextInput
                {
                        Input = x.Input
                }
        };

        return returnValue;
}
}
```

This means that if I want to prompt the user for input, and guarantee that they gave me an interger, it’s now very easy to code:

```
  public int GetPlayerSpendOnOxen()
  {
   var input = this.UserInteraction.GetInputFromUse
   var returnValue = input switch
   {
    IntegerInput ii => ii.Input,
    _ => {
     this.UserInteraction.WriteMessage("Try again"
     return GetPlayerSpendOnOxen();
    }
   };

   return returnValue;
  }
```

What I’m doing in this code block is prompting the player for input. Then, I check whether it’s the integer I expected - based on the check already done on it via a discriminated union. If it’s an integer, great. Job’s a good ‘un, return that interger.

If not, then the player needs to be prompted to try again, and I call this function again, recursively. I could add more detail in about capturing and logging any errors received, but I think this demonstrates the principle soundly enough.

Note also, that there isn’t a need for a Try/Catch in this function. That is already handled by the lower-level function.

There are many, many places this code checking for integer are needed in my Oregon Trail conversion. Imagine how much code I’ve saved myself by wrapping the integer check into the structure of the return object!

Generic Unions

All of the discriminated unions so far are entirely situation specific. Before wrapping up this chapter, I want to discuss a few options for creating entirely generic, reusable versions of the same idea.

Firstly, let me re-iterate - we can’t have discriminated unions you can simply declare easily, on the fly, like the folks in F# can. It’s just not a thing we can do. Sorry. The best we can do is emulate it as closely as possible, with some sort of boilerplate trade-off.

Here are a couple of functional structures you can use. There are, incidentally, more advanced ways to use these coming up in the next chapter. Stay tuned for that.

Maybe

If your intention with using a Discriminated Union is to represent that data might not have been found by a function, then the Maybe structure might be the one for you.

Implementations look like this:

```
public abstract class Maybe<T>
{
}

public class Something<T> : Maybe<T>
{
 public Something(T value)
```

```
   {
    this.Value = value;
   }

   public T Value { get; init; }
  }

  public class Nothing<T> : Maybe<T>
  {

  }
```

You’re basically using the Maybe abstract as a wrapper around another class, the actual class your function reutrns, but by wrapping it in this manner, you are signalling to the outside world that there may not necessarily be anything returned.

Here’s how you might use it for a function that returns a single object:

```
  public Maybe<DoctorWho> GetDoctor(int doctorNumbe
  {
   try
   {
     using var conn = this._connectionFactory.Make(
    // Dapper query to the db
    var data = conn.QuerySingleOrDefault<Doctor>(
     "SELECT * FROM [dbo].[Doctors] WHERE DocNum =
     new { docNum = doctorNumber });
     return data == null
      ? new Nothing<DoctorWho>();
      : new Something<DoctorWho>(data);
   }
  catch(Exception e)
  {
   this.logger.LogError(e, "An error occurred gett
   return new Nothing<DoctorWho>();
  }

  }
```

You’d use that something like this:

```
// William Hartnell.  He's the best!
var doc = this.DoctorRepository.GetDoctor(1);
var message = doc switch
{
 Something<DoctorWho> s => "Played by " + s.Value
 Nothing<DoctorWho> _ => "Unknown Doctor"
};
```

This doesn’t handle error situations especially well. A Nothing state at least prevents unhandled exceptions from occurring, and we are logging, but nothing useful has been passed back to the end user.

Result

An alternative to a Maybe is a Result, which represents the possibility that a function might throw an error, instead of returning anything. It might look like this:

```
public abstract class Result<T>
{
}

public class Success : Result<T>
{
 public Success<T>(T value)
 {
  this.Value = value;
 }

 public T Value { get; init; }
}

public class Failure<T> : Result<T>
{
 public Failure(Exception e)
 {
  this.Error = e;
 }

 public Exception Error { get; init; }
```

```
    public Exception Error { get; init; }
  }
```

Now, the Result version of the "Get Doctor" function looks like this:

```
  public Result<DoctorWho> GetDoctor(int doctorNumb
  {
   try
   {
     using var conn = this._connectionFactory.Make
    // Dapper query to the db
    var data = conn.QuerySingleOrDefault<Doctor>(
     "SELECT * FROM [dbo].[Doctors] WHERE DocNum =
     new { docNum = doctorNumber });
     return new Success<DoctorWho>(data);
   }
  catch(Exception e)
  {
   this.logger.LogError(e, "An error occurred gett
   return new Failure<DoctorWho>(e);
  }

  }
```

And you might consider using it, something like this:

```
// Sylvester McCoy.  He's the best too!
var doc = this.DoctorRepository.GetDoctor(7);
var message = doc switch
{
 Success<DoctorWho> s when s.Value == null => "Ur
 Success<DoctorWho> s2 => "Played by " + s2.Value
 Failure<DoctorWho> e => "An error occurred: " e
};
```

Now I'm covering the error scenario in one of the possible states of the Discriminated Union, but the burden of null checking falls to the receiving function.

Maybe vs Result

A perfectly valid question at this point - which is better to use? Maybe or Result?

The Maybe gives a state that informs the user that no data has been found, removing the need for null checks, but effectively silently swallows errors. It's better than an unhandled exception, but it could result in unreported bugs.

The Result handles errors elegantly, but puts a burden on the receiving function to check for null.

My personal preference? This might not be strictly within the standard definition of these structures, but I combine them into one. I usually have a 3-state Maybe - Something, Nothing, Error. That handles just about anything the codebase can throw at me.

This would be my personal solution to the problem:

```
public abstract class Maybe<T>
{
}

public class Something<T> : Maybe<T>
{
 public Something(T value)
 {
  this.Value = value;
 }

 public T Value { get; init; }
}

public class Nothing<T> : Maybe<T>
{

}

public class Error<T> : Maybe<T>
```

```
{
 public Error(Exception e)
 {
  this.CapturedError = e;
 }

 public Exception CapturedError { get; init; }
}
```

And I’d use it like this:

```
public Maybe<DoctorWho> GetDoctor(int doctorNumbe
{
 try
 {
   using var conn = this._connectionFactory.Make
  // Dapper query to the db
  var data = conn.QuerySingleOrDefault<Doctor>(
   "SELECT * FROM [dbo].[Doctors] WHERE DocNum =
   new { docNum = doctorNumber });
   return data == null
    ? new Nothing<DoctorWho>();
    : new Something<DoctorWho>(data);
 }
catch(Exception e)
{
 this.logger.LogError(e, "An error occurred gett
    return new Error<DoctorWho>(e);
  }

  }
```

This means the receiving function can now handle all three states elegantly with a pattern matching expression:

```
  // Peter Capaldi.  The other, other best Doctor!
  var doc = this.DoctorRepository.GetDoctor(12);
  var message = doc switch
  {
   Nothing<DoctorWho> _ => "Unknown Doctor!",
   Something<DoctorWho> s => "Played by " + s.Value
   Error<DoctorWho> e => "An error occurred: " e.E
  };
```

I find this allows me to provide a full set of responses to any given scenario, when returning from a function that requires a connection to the cold, dark, hungry-wolf-filled world beyond my program, and easily allows a more informative response back to the end user.

Before we finish on this topic, here’s how I’d use that same structure to handle a return type of Enumerable:

```
public Maybe<IEnumerable<DoctorWho>> GetAllDocto
{
 try
 {
   using var conn = this._connectionFactory.Make
  // Dapper query to the db
  var data = conn.Query<Doctor>(
   "SELECT * FROM [dbo].[Doctors]");
   return data == null || !data.Any()
    ? new Nothing<IEnumerable<DoctorWho>>();
    : new Something<IEnumerable<DoctorWho>>(data
 }
catch(Exception e)
{
 this.logger.LogError(e, "An error occurred gett
 return new Error<IEnumerable<DoctorWho>>(e);
}

}
```

This allows me to handle the response from the function like this:

```
// Great chaps.  All of them!
var doc = this.DoctorRepository.GetAllDoctors();
var message = doc switch
{
 Nothing<IEnumerable<DoctorWho>> _ => "No Doctor
```

```
   Something<IEnumerable<DoctorWho>> s => "The Doct
    string.Join(Environment.NewLine, s.Value.Select
   Error<IEnumerable<DoctorWho>> e => "An error occ
  };
```

Once again, nice and elegant, and everything has been considered. This is an approach I use all the time in my everyday coding, and I hope after reading this chapter that you will too!

Either

Something and Result - in one form or another - now generically handle the idea of returning from a function where there’s some uncertainty as to how it might behave. What about scenarios where you might want to return two or more entirely different types?

This is where the Either type comes in. The syntax isn’t the nicest, but it does work.

```
public abstract class Either<T1, T2>
{

}

public class Left<T1, T2> : Either<T1, T2>
{
 public Left(T1 value)
 {
  Value = value;
 }

 public T1 Value { get; init; }
}

public class Right<T1, T2> : Either<T1, T2>
{
 public Right(T2 value)
 {
  Value = value;
 }

 public T2 Value { get; init; }
}
```

I could use it to create a type that might be left or right, like this:

```
   public Either<string, int> QuestionOrAnswer() =>
    new Random().Next(1, 6) >= 4
     ? new Left<string, int>("What do you get if you
     : new Right<string, int>(42);

   var data = QuestionOrAnswer();
   var output = data switch
   {
    Left<string, int> l => "The ultimate question wa
    Right<string, int> r => "The ultimate answer was
  };
```

You could of course expand this out to have three or more different possible types. I'm not entirely sure what you'd call each of them, but it's certainly possible. There's just a lot of awkward boilerplate, in that you have to include all of the references to the generic types in a lot of places. At least it works, though…

Conclusion

In this chapter we discussed Discriminated Unions. What exactly they are, how they are used, and just how incredibly powerful they are as a code feature.

Discriminated Unions can be used to massively cut down on boilerplate code, and make use of a datatype that descriptively represents all possible states of the system, in a way that strongly encourages the receiving function to handle them appropriately.

Discriminated Unions can't be implemented quite as easily as in F#, or other functional languages, but there are possibilities in C#, at least.

In the next chapter, I’ll be looking into some more advanced functional concepts, which will take Discriminated Unions up to the next level!

let me please re-assure everyone that despite being called “discriminated unions” they bear no connection to anyone’s view of love and/or marriage or to worker’s organizations.

Didn’t I tell you? We’re in the travel business now, you and I! Togther we’ll flog cheap holidays to unsuspecting punters until we retire rich and contented. That, or carry on doing what we’re doing now. Either way.

It’s not London Bridge, that famous one you’re thinking of. London Bridge is elsewhere. In Arizona, in fact. No, really. Look it up

N.B - no-one has ever done this. I’m not aware of a single cat ever being sacrificed in the name of quantum mechanics

Somehow. I’ve never really understood this part of it.

Wow. What a horrible birthday present that would be. Thanks, Schrödinger!

“先生” - It literally means “one who was born earlier”. Interestingly, if you were to write the same letters in Japanese, it would be pronounced “Sensei”. I’m a nerd - I love stuff like this!

Sadly the closest I'll ever get to him for real. Do watch some of his Hong-Kong films, if you haven't already! I'd start with the Police Story series.

Just kidding, folks, honest! Please don't take me off your Christmas card lists!

Chapter 7. Functional Flow

Calls to external systems, whether they be databases, web APIs or whatever are an absolute pain, aren't they? Before making use of your data - the most important part of the function you're writing, you have to:

1. Catch and handle any exceptions. Maybe the network was glitching, or the DB server offline?
2. Check what's come back from the database is not NULL
3. Check that there's an actual, reasonable set of data, even if it isn't null.

That’s a lot of tedious boilerplate, and all of that gets in the way of your actual business logic.

Use of the Maybe Discriminated Union from the previous chapter will help somewhat with returning something other than Null for records not found or errors encountered, but there’s still boilerplate required even then.

What if I were to tell you there were a way you could never have to see another unhandled Exception again? Not only that, but you’d never even need to use a Try/Catch block again. As for Null checks? Forget ‘em. You won’t ever have to do that again either.

Don’t believe me? Well, strap in, I’m going to introduce you to one of my favorite features of functional programming. Something I use all the time in my day job, and I’m hoping that after reading this chapter, so too might you.

Maybe, Revisited

Mentioning that Maybe discriminated union from the last chapter - I’d like to revisit it now, but this time I’m going to show you how it can be even more useful than you could ever have imagined.

What I’m going to do is add in a version of the Map extension method I used a couple of chapters ago. If you remember the `Map` combinator back in Chapter 5 “Chaining Functions” works similarly to the LINQ Select method, except it acts on the entire source object, not individual elements from it.

This time though, I’m going to add a bit of logic inside the Map, something that will determine which actual type is going to come out. I’m giving it a different name this time - Bind[1]

```
public static Maybe<TOut> Bind<TIn, TOut>(this Ma
{
        try
        {
                Maybe<TOut> updatedValue = @this
                {
                        Something<TIn> s when !E
                                new Something<TO
                        Something<TIn> _ => new N
                        Nothing<TIn> _ => new No
                        Error<TIn> e => new Erro
                        _ => new Error<TOut>(new
                };
                return updatedValue;
        }
        catch (Exception e)
        {
                return new Error<TOut>(e);
```

```
          }
  }
```

So, what’s happening here? One of a few possible things:

- The current value of `This` - i.e. the current object being held by the Maybe is a `Something` - i.e. an actual value, and the value held is non-default[2], in which case the supplied function is run, and whatever came out of it is returned in a new `Something`.
- The current value of `This` is a `Something`, but the value inside it is default (Null, in most cases), in which case instead of a *Something*, we now return a `Nothing`
- The current value of `This` is a Nothing, in which case, return another Nothing. No point doing anything else.
- The current value of `This` is an Error. Again, no point doing anything except passing it on.

What’s the point of all this? Well, imagine the following procedural code:

```
public string MakeGreeting(int employeeId)
{
 try
 {
  var e = this.empRepo.GetById(employeeId);
  if(e != null)
```

```
    {
     return "Hello " + e.Salutation + " " + e.Name
    }

    return "Employee not found";
   }

   catch(Exception e)
   {
     return "An error occurred: " + e.Message;
   }
  }
```

When you look at it, the actual purpose of this code is incredibly simple. Fetch an employee. If there are no issues, say hello to them. But, since Null and unhandled exceptions are a thing, we have to write so much defensive code. Null checks and `Try/Catch` blocks. Not just here, but all over our codebase.

What's worse is that we make it the problem of the code calling this function to know what to do with it. How do we signal that there was an error, or that the employee wasn't found? In my example I just return a string for whatever application we've written to display blindly. The other option would be to return some sort of return object with meta data attached (bool DataFound, bool ExceptionOccurred, Exception CapturedException - that sort of thing).

Using a Maybe and Bind function though, none of that is necessary. The code could be re-written like this:

```
public Maybe<string> MakeGreeting(int employeeId
 new Something(employeeId)
  .Bind(x => this.empRepo.GetById(x))
  .Bind(x => "Hello " + x.Salutation + " " + x.Na
```

Think about the possible results for each bind I listed above.

If the Employee Repository returned a Null value, the next Bind call would identify a Something with a default (i.e. Null) value, and not execute the function that constructs a greeting string, instead it would return a Nothing.

If an error occurred in the repository (maybe a network connection issue, something impossible to predict or prevent) then the error would simply be passed on, instead of executing the function.

The ultimate point I'm making is that the arrow function that assembles a greeting will only ever execute if the previous step a) returned an actual value and b) didn't throw an unhandled exception.

This means that the small function written above with use of the Bind method is functionally identical to the previous version, covered with defensive code.

It gets better…

We’re not returning a string any more, we’re returning a Maybe<string>. This is a descriminated union that can be used to inform whatever is calling our function what the result of execution was, whether it worked, etc. That can be used in the outside world to decide how to handle the resulting value, or it can be used in subsequent chains of Bind calls.

Either like this:

```
public Interface IUserInterface
{
 void WriteMessage(string s);
}

// Bit of magic here because it doesn't matter
this.UserInterface = Factory.MakeUserInterface()

var message = makeGreetingResult switch
{
 Something s => s.Value,
 Nothing _ => "Hi, but I've never heard of you."
 Error _ => "An error occurred, try again"
};

this.UserInterface.WriteMessage(message);
```

Or alternatively you could adapt the UserInterface module so that it takes a Maybe as a parameter:

```
public Interface IUserInterface
{
 void WriteMessage(Maybe<string> s);
}

// Bit of magic here because it doesn't matter
this.UserInterface = Factory.MakeUserInterface()


var logonMessage = MakeGreeting(employeeId)
 .Bind(x => x + Environment.NewLine + MakeUserInt
this.UserInterface.WriteMessage(logonMessage);
```

Swapping out a concrete value in an Interface for a Maybe<T> is a sign to the class that consumes it that there is no certainty that the operation will work, and forces the consuming class to consider each of the possibilities and how to handle them. It also puts the onus on deciding how to respond entirely in the hands of the consuming class. There's no need for the class returning the Maybe to have any interest in what happens next.

The best description I've ever encountered for this style of programming was in a talk and related articles by Scott Wlashin called Railway Oriented Programming [3]

Wlashin desribes the process as being like a railway line with a series of points. Each set of points is a Bind call. The train starts on the Something line, and every time a function passed to a Bind is executed, the train either carries on to the next set of points, or switches to the Nothing path, and simply glides along to the station at the end of the line without doing any more work.

It's a beautiful, elegant way of writing code, and cuts out so much boilerplate it isn't even funny.

If only there were a handy technical term for this structure. Oh wait, there is! It's called a Monad!

I said way back at the beginning that they might pop up somewhere. If anyone ever says to you that Monads are complicated, I hope you can see now that they're wrong.

Monads are like a wrapper around a value of some kind. Like an envelope, or a burrito. They hold the value, but make no comment about what it's actually set to.

What they do is give you the ability to hang functions off them which provide a safe environment to do perform operations without having to worry about negative consequences - such as null reference exceptions.

The Bind function is like a relay race - each call does some sort of operation, then passes its value to the next runner. It also handles errors and nulls for you, so you don’t need to worry about writing so much defensive code.

If you like, imagine it being like an explosion proof box. You have a package you want to open, but you don’t know whether it’ll be something safe like a letter[4] or could it be something explosive, just waiting to take you down when you lift the lid. If you pop it inside the Monad container, it can open safely or explode, but the Monad will keep you safe from the consequences.

That’s really all there is too it. Well, mostly.

In the rest of this chapter I’ll consider what else we can do with Monads, and what other sorts of Monads there are. Don’t worry though, the “hard” part is over now, if you’ve passed this point, and you’re still with me, then the rest of this book is going to be a piece of cake[5].

Maybe and Debugging

A comment I hear sometimes hear regarding strings of Bind statements is that it’s harder to use debug tools in Visual Studio to step through the changes. Especially when you have scenarios like this:

```
var returnValue = idValue.ToMaybe()
```

```
    .Bind(transformationOne)
    .Bind(transformationTwo)
    .Bind(transformationThree);
```

It is actually possible in most versions of Visual Studio, but you'd need make sure you keep mashing the "Step-in" key to enter the nested arrow functions inside the Bind call. It's still hardly the best for working out what's happening if a value isn't being calculated correctly. It's even worse when you consider Step-in will enter the Maybe's Bind function and need a few more steps to be taken before seeing the result of the arrow function.

I tend to write my Binds one to a line, each storing a variable containing their individual output:

```
var idMaybe = idValue.ToMaybe();
var transOne = idMaybe.Bind(x => transformationOn
var transTwo = transOne.Bind(x => transformationT
var returnValue = transTwo.Bind(x => transformati
```

Functionally these two samples are identical, it's just that we're capturing each output separately, rather than immediately feeding it into another function and discarding them.

The second sample is easier to diagnose issues with though, as you can inspect each intermediate value. Made easier due to the functional programming technique of never modifying a variable once set - this means that every intermediate step in the process is set in stone to work through what exactly happened, as well as how and where things went wrong in the event of a bug.

These intermediate values will remain in scope for the entirety of the life of whatever larger function they're part of, though. So, if it's an especially large function, and one of the intermediate values is hefty, it might be worth merging them so that the large intermediate vaue is de-scoped as early as possible.

This decision is mostly a matter of personal style, and one or two codebase constraints. Whichever you choose is fine.

Map vs Bind

Strictly speaking, I'm not implementing the Bind function in accordance with the functional paradigm.

There *should* be two functions attached to the Maybe: Map and Bind. They're nearly the same, but with a small and subtle difference.

The Map function is like the one I described in the previous section - it attaches to a Maybe<T1> and needs a function that gives you the value of type T1 from inside the Maybe and requires you to turn it into a type T2.

An actual Bind needs you to pass in a function that returns a Maybe of the new type - i.e. Maybe<T2>. It still returns the same result as the Map function,

```
public static Maybe<TOut> Map<TIn, TOut>(this May

public static Maybe<TOut> Bind<TIn, TOut>(this Ma
```

If, for example, you had a function that calls a database and returns a type of `Maybe<IEnumerable<Customer>>` to represent a list of customers that may or may not have been found - then you'd call that with a Bind function.

Any subsequent chained functions to change the `Enumerable` of customers into some other form would be done with a Map call, since those changes are data-to-data, not data-to-maybe.

Here's how you might go about implementing a proper Bind:

```
public static Maybe<TOut> Bind<TIn, TOut>(this Ma
{
        try
        {
                var returnValue = @this switch
                {
```

```
                                Something<TIn> s => f(s.\
                                _ => new Nothing<TOut>()
                        };
                        return returnValue;
                }
                catch (Exception _)
                {
                        return new Nothing<TOut>();
                }
        }
```

And an example of how to use it:

```
public Interface CustomerDataRepo
{
 Maybe<Customer> GetCustomerById(int customerId),
}

public string DescribeCustomer(int customerId) =>
 new Something<int>(customerId)
  .Bind(x => this.customerDataRepo.GetCustomerByI
  .Map(x => "Hello " + x.Name);
```

Use this new Bind function and rename the previous one Map, and you’re conforming to the functional paradigm a little closer.

In production code however, I don’t personally do this. I just use a function called Bind for both purposes.

Why, you might ask?

It’s mostly to prevent confusion, in all honesty. There is a Map function that’s native to JavaScript but which operates like a C# `Select` on individual elements of arrays. In C# there’s also a Map function in Jimmy Bogard’s AutoMapper library[6], which is used to convert an array of objects from one type to another.

With both of these instances of a Map function already in use in many C# codebases, I thought adding another Map function into the mix might confuse other folks looking at my code. For this reason, I use Bind for all purposes, as there isn’t already a Bind function anywhere in C# or JavaScript - except for in libraries implementing the functional paradigm.

You can please yourself whether you use the more strictly accurate version with both Map and Bind, or the route that - in my opinion - is less confusing and more pragmatic, which is to simply use multiple instances of the Bind function to serve every purpose.

I’m going to carry on assuming that second option for the rest of this book.

Maybe and the Primitives

This sounds like the title of an amazing pulp adventure novel that was never written. Probably involving our heroine - Captain Maybe, swinging to the rescue somewhere in a lost civilisation populated by aggressive cave dwelling nasties.

In fact, a primitive type in C# is one of a set of built-in types that *don't* default to Null. This is a list of them:

- bool
- byte
- sbyte
- char
- decimal
- double
- float
- int
- uint
- nint
- nuint
- long
- ulong[7]
- short
- ushort

The point here is that if I were to use any of these on my `Bind` function from the previous sections, and set their value to 0, it would fall foul of the check against `default`, because most of these default to 0[8]

Here's an example of a unit test that would fail (I'm using XUnit with FluentAssertions for a friendlier, human-readable assert style):

```
[Fact]
public Task primitive_types_should_not_default_to
{
        var input = new Something<int>(0);
        var output = input.Bind(x => x + 10);
        (output as Something<int>).Value.Should()
}
```

This test stores an integer with the value of 0 in a Maybe, then attempts to `Bind` it into a value 10 higher - i.e. it should equal 10. In the existing code, the switch operation inside `Bind` would consider the value 0 to be default, and switch the return type from `Something<int>` to `Nothing<int>` and the function to add 10 wouldn't be carried out, meaning that *output* in my unit test would be switched into a null, and the test would fail with a null reference exception.

Arguably though, this isn't correct behavior, 0 is a valid value of integer.

It’s easily fixed with an additional line in the `Bind` function, however:

```
public static Maybe<TOut> Bind<TIn, TOut>(this Ma
{
        try
        {
                Maybe<TOut> updatedValue = @this
                {
                        Something<TIn> s when !Ec
                                new Something<TOu
                                // This is the ne
                        Something<TIn> s when s.(
                        Something<TIn> _ => new N
                        Nothing<TIn> _ => new Not
                        Error<TIn> e => new Error
                        _ => new Error<TOut>(new
                };
                return updatedValue;
        }
        catch (Exception e)
        {
                return new Error<TOut>(e);
        }
}
```

The new line checks the first generic argument of the `Maybe<T>` - i.e. the “real” type of `T`. All of the types I listed at the beginning of this section would have a value of `IsPrimitive` set to `true`.

If I were to re-run my unit test with this modified `Bind` function, then the 0-valued `int` still wouldn't match on the check against not being `default`, but the next line would match, because int is a primitive.

This does now mean that all primitives are *incapable* of being a `Nothing<T>`. Whether that's right or wrong is a matter for you to assess for yourself.

You might want to consider it a `Nothing<T>` if *T* is a `bool`, for example. If that's the case, another case would need to be added to the switch between the first two lines to handle the specific case of *T* being `bool`.

It might also be important to a calculation that it be possible for a boolean `false` to be passed into a function to perform a calculation. As I said, it's a question you can best answer yourself.

One way to avoid this situation altogether is to always pass a nullable class around as *T* so that you can be sure that you're getting the correct behavior when trying to decide whether what you're looking at is `Something` or `Nothing`.

Maybe and Logging

Another thing worth considering for the use of Monads in a professional environment is the all-important developer's tool - Logging. It's often crucially important to log information about the status of the progress of a function. Not just errors, but also all sorts of important information.

It's possible to do something like this, of course:

```
var idMaybe = idValue.ToMaybe();
var transOne = idMaybe.Bind(x => transformationOn
if(transOne is Something<MyClass> s)
{
 this.Logger.LogInformation("Processing item " +
}
else if (transOne is Nothing<MyClass>)
{
 this.Logger.LogWarning("No record found for " +
}
else if (transOne is Error<MyClass> e)
{
 this.Logger.LogError(e, "An error occurred for '
}
```

This is likely to balloon out of hand if you do much of this, though. Especially if there are many binds in the process that all require logging.

It might be possible to leave out the error log until the very end, or even all the way out in the Controller, or whatever else it was that ultimately originated this request. The error message would be passed from hand to hand untouched. But that still leaves ocassional log messages for Information or Warning purposes.

I prefer to add extension methods to the Maybe to provide a set of event handler functions:

```
public static class MaybeLoggingExtensions
{
 public static Maybe<T> OnSomething(this Maybe<T>
 {
  if(@this is Something<T>)
  {
   a(@this);
  }

  return @this;
 }

 public static Maybe<T> OnNothing(this Maybe<T> @
 {
  if(@this is Nothing<T> _)
  {
   a();
  }
```

```
      return @this;
    }
  }

    public static Maybe<T> OnError(this Maybe<T> @th
    {
      if(@this is Error<T> e)
      {
        a(e.CapturedError);
      }

      return @this;
    }
```

The way I’d use this then, is more like this:

```
  var idMaybe  idValue.ToMaybe();
  var transOne = idMaybe.Bind(x => transformationOn
    .OnSomething(x => this.Logger.LogInformation("Pr
    .OnNothing(() => this.Logger.LogWarning("No reco
    .OnError(e => this.Logger.LogError(e, "An error 
```

This is fairly usable, although it does have a drawback.

The OnNothing and OnError states will proliferate from Bind to Bind unmodified, so if you have a long list of Bind calls with OnNothing or OnError handler functions, they'll all fire every time. Like this:

```
var idMaybe  idValue.ToMaybe();
var transOne = idMaybe.Bind(x => transformationOr
 .OnNothing(() => this.Logger.LogWarning("Nothing
var transTwo = transOne.Bind(x => transformation
 .OnNothing(() => this.Logger.LogWarning("Nothing
var returnValue = transTwo.Bind(x => transformat
  .OnNothing(() => this.Logger.LogWarning("Nothin
```

In the code sample above, all three OnNothings will fire, and three Warning logs will be written. You may want that, or you may not. It might not be all that interesting after the first nothing.

I **do** have a solution for this issue, but it means quite a lot more coding.

Create a new instance of Nothing and Error that descent from the originals:

```
public class UnhandledNothing<T> : Nothing<T>
{
}

public class UnhandledError<T> : Error<T>
{
}
```

```
}
```

We’d also need to modify the Bind function so that these are the types that are returned when switching from the Something path to one of these.

```
public static Maybe<TOut> Bind<TIn, TOut>(this Ma
{
        try
        {
                Maybe<TOut> updatedValue = @this
                {
                        Something<TIn> s when !Eq
                                new Something<TOu
                        Something<TIn> s when s.G
                        Something<TIn> _ => new U
                        Nothing<TIn> _ => new Not
                        UnhandledNothing<TIn> _ =
                        Error<TIn> e => new Error
                        UnhandledError<TIn> e => 
                        _ => new Error<TOut>(new 
                };
                return updatedValue;
        }
        catch (Exception e)
        {
                return new UnhandledError<TOut>(e
        }
}
```

Then finally, the handler functions need to be updated:

```
public static class MaybeLoggingExtensions
{

 public static Maybe<T> OnNothing(this Maybe<T> @
 {
  if(@this is UnhandledNothing<T> _)
  {
   a();
   return new Nothing<T>();
  }

  return @this;
 }
}

 public static Maybe<T> OnError(this Maybe<T> @th
 {
  if(@this is UnhandledError<T> e)
  {
   a(e.CapturedError);
   return new Error<T>(e.CapturedError);
  }
```

```
    return @this;
   }
```

What all of this means is that when a switch happens from Something to one of the other states, the Maybe switches to a state that not only signals that either Nothing or an Exception occurred, but also that nothing has yet handled that state.

Once one of the handler functions is called, and an unhandled state is found, then the callback is triggered to log, or whatever, and a new object is returned that maintains the same state type, but this time indicating that isn't no longer unhandled.

This means that in my example above with multiple Bind calls with OnNothing functions attached, only the first OnNothing will actually be triggered, the rest will be ignored.

There's nothing stopping you still using a pattern matching statement to examine the type of the Maybe to perform some action or other once the Maybe reaches its final destination, elsewhere in the codebase.

Maybe and Async

So, I know what you're going to ask me next. Look, I'm going to have to let you down gently. I'm already married. Oh? My mistake. Async and

Monads. Yeah, OK. Moving on…

How do you handle calls inside a Monad to processes that are async? Honestly, it’s not all that hard. Leave the Maybe Bind functions you’ve already written, and add these to the codebase as well:

```
public static async Task<Maybe<TOut>> BindAsync<
{
        try
        {
                Maybe<TOut> updatedValue = @this
                {
                        Something<TIn> s when Equ
                                new Something<TOu
                        Something<TIn> _ => new N
                        Nothing<TIn> _ => new Not
                        Error<TIn> e => new Erro
                        _ => new Error<TOut>(new
                };
                return updatedValue;
        }
        catch (Exception e)
        {
                return new Error<TOut>(e);
        }
}
```

All I’ve done here is wrap another layer around the value we’re passing around.

The first layer is the Maybe - representing that the operation we’re trying may not have worked

The second layer is the Task - representing that an `async` operation needs to be carried out first, before we can get to the Maybe.

Using this out in your wider codebase is probably best done a line at a time so you can avoid mixing up the async and non-async versions in the same chain of Bind calls, otherwise you could end up with a `Task<T>` being passed around as a type, rather than the actual type, `T`. Also, it means you can separate each Async call out and use an await statement to get the real value to pass along to the next Bind operation.

Nested Maybes

There is a problem scenario with the Maybes I’ve shown so far. This was something I only really came to realise existed once I’d changed an awful lot my Interfaces to have Maybe<T> as the return type for anything involving an external interaction.

Here’s a scenario to consider. I’ve created a couple of different data loaders of some description. They could be databases, web APIs, or whatever. Doesn’t matter:

```
public interface DataLoaderOne
{
        Maybe<string> GetStringOne();
}

public interface DataLoaderTwo
{
        Maybe<string> GetStringTwo(string stringO
}

public interface DataLoaderThree
{
        Maybe<string> GetStringThree(string strin
}
```

In some other part of the code I want to orchestrate calls to each of these interfaces in turn using a Bind call.

Note that the point of the `Maybe<string>` return type is that I can reference them through `Bind` functions and if any of the dataLoader calls fail, the subsequent steps *wont*' be executed and I'll get a `Nothing<string` or `Error<string>` at the end to examine.

Something like this:

```
            var finalString = dataLoaderOne.GetString
                    .Bind(x => dataLoaderTwo.GetStri
                    .Bind(x => dataLoaderThree.GetSt
```

What I find though is that this code won’t compile. Why do you suppose that is?

There are three function calls at work here, and all three return the type `Maybe<string>`. Look at what happens a line at a time:

1. First GetStringOne returns a `Maybe<string>`. So far, so good.
2. Next the Bind call attaches to the `Maybe<string>` and unpacks it to a string to pass to GetStringTwo, whose return type is popped into a new `Maybe` for safe keeping.
3. The next bind call unwraps the return type of that last bind so that it’s only the return type of GetStringTwo - but GetStringTwo didn’t return a `string`, it returned `Maybe<string>`. So, on this second Bind call, x is actually equal to `Maybe<string>` which can’t be passed into GetStringThree!

I *could* solve this by directly accessing the value out of the Maybe stored in x, but first I’d need to cast it to a Something. What if it weren’t a something, though? What if an error had occurred in GetStringOne talking to the database? What if no string could be found?

I basically need a way to unpack a nested Maybe, but *only* in the event that it returns a Something containing a real value, in all other cases, we need to match its unhappy path (Nothing or Error).

The way I'd do it is with another Bind function to sit alongside the other two we've already created, but this one specifically handles the issue of nested Maybes.

I'd do it like this:

```
public static Maybe<TOut> Bind<TIn, TOut>(
        this Maybe<Maybe<TIn>> @this, Func<TIn, 
{
        try
        {
                var returnValue = @this switch
                {
                        Something<Maybe<TIn>> s 
                        Error<Maybe<TIn>> e => ne
                        Nothing<Maybe<TIn>> => ne
                        _ => new Error<TOut>(new

                };
                return returnValue;


        }
```

```
            catch (Exception e)
            {
                    return new Error<TOut>(e);
            }
  }
```

What we’re doing here is taking the nested bind (`Maybe<Maybe<string>>`) and calling Bind on it, which unwraps the first layer, leaving us simply with `Maybe<string>` inside the Bind callback function, from there we can just do the exact same logic on the Maybe as in previous Bind functions.

This would need to be done for the `async` version as well:

```
  public static async Task<Maybe<TOut>> BindAsync<T
  {
   try
   {
            var returnValue = await @this.Bind(async
            {
                      var updatedValue = @this switch
                      {
                       Something<TIn> s when
                        EqualityComparer<TIn>.Default
                         new Something<TOut>(await f
                       Something<TIn> _ => new Nothin
                      Nothing<TIn> _ => new Nothing
                      Error<TIn> e => new Error<TOut
                     }
                     return updatedValue;
          });
    return returnValue;
   }
   catch(Exception e)
   {
    return new Error<TOut>(e);
   }
  }
```

If you want another way to think about this process. Think of the `SelectMany` function in LINQ. If you feed it an array of arrays - i.e. a multi-dimensional array, then you get back a single-dimension, flat array. This handling of nested `Maybe` objects now allows us to do the same thing with Monads. This is actually one of the “Laws” of Monads - the properties that anything that calls itself a Monad is expected to follow.

In fact, that leads me neatly onto the next topic. What exactly are Laws, what are they for, and how do we make sure our C# Monads follow them too…

The Laws

Strictly speaking, to be considered a true Monad, there are a set of rules (known as *laws*) that you have to conform to. I'll briefly talk through each of these laws, so you'll know for yourself whether you're looking at a real Monad or not.

Left Identity Law

This states that a Monad, given a function as a parameter to its Bind method will return something that is entirely the equivalent of just running the function directly with no side-effects.

Here's some C# code to demonstrate that:

```
Func<int, int> MultiplyByTwo = x => x * 2;
var input = 100;
var runFunctionOutput = MultiplyByTwo(input);

var monadOutput = new Something<int>(input).Bind

// runFunctionOutput and monadOutput.Value shoul
// be identical - 200 - to conform to the Left I
```

Right Identity Law

Before I explain this one, I need to move back a few steps…

Firstly, I need to explain Functors. These are functions that convert a thing, or list of things from one form into another. `Map`, `Bind` and `Select` are all examples of Functors.

The very simplest functor that exists is the Identity functor. The identity is a function that given an input, returns it back again unaltered, and with no side-effects. It can have its uses, when you’re composing functions together.

The only reason I’m interested in it here and now is that it’s the basis of the second Monad law - the Right Identity Law. This means a Monad, when given an Identity Functor in its Bind function, will return the original value back with no side effects.

I could test the Maybe I created in the last chapter like this:

```
Func<int,int> identityInt = (int x) => x;
var input = 200;
var result = new Something<int>(input).Bind(ident
// result = 200
```

All this means is that the Maybe takes a function that doesn’t result in an error or `Null`, executes it, then returns exactly whatever comes out of it, and nothing else.

The basic gist of both of these first two laws is simply that the Monad can’t interfere in any way with the data coming in or going out, or in the execution of the funtion provided as a parameter to the Bind method. A Monad is simply a pipe down which functions and data flow.

Associativity law

The first two laws should be fairly trivial, and my Maybe implementation fulfills them both. The last law, the Associativity Law, is a bit harder to explain.

It basically means that it doesn’t matter how the Monads are nested, you always end up with a single Monad containing a value at the end.

Here’s a simple C# example:

```
var input = 100;
var op1 = (int x) => x * 2;
var op2 = (int x) => x + 100;

var versionOne = new Something<int>(input)
 .Bind(op1)
 .Bind(op2);

 // versionOne.Value = 100 * 2 + 100 = 300
```

```
   var versionTwo = new Something<int>(input)
    .Bind(x => new Something<int>(x).Bind(op1)).Bi

    // If we don't implement something to fulfill t
    // associativity law, we'll end up with a type
    // Something<Something<int>>, where we want th
    // the exact same type and value as versionOne
```

Look back a few sections to my description of how to deal with nested Maybes“Nested Maybes” and you’ll see how this is implemented.

With any luck, now we’ve looked at the three Monad laws, I’ve proven that my Maybe is a proper, no-holds barred, honest to dog Monad.

In the next section, I’ll show you another Monad you might use to do away with the need for storing variables that need to be shared around.

Reader

Let’s imagine for a moment that we’re putting together a report of some kind. It does a series of pulls of data from an SQL Server database.

First we need to grab the record for a given user.

Next, using that record, we get their most recent order from our entirely imaginary bookshop[9].

Finally, we turn the most recent Order record into a list of items from the order and return a few details from them in a report.

We want to take advantage of a monad-style Bind operation, so how do we ensure the data from each step is passed along with the database connection object? That's no problem, we can throw together a Tuple and simply pass along both objects.

```
public string MakeOrderReport(string userName) =
 (
  Conn: this.connFactory.MakeDbConnection(),
  userid
 )
  .Bind(x => (
   x.Conn,
   Customer: this.customerRepo.GetCustomer(x.Con
  )
  .Bind(x => (
   x.Conn,
   Order: this.orderRepo.GetCustomerOrders(x.Con
  ),
  .Bind(x => this.Order.Items.First())
  .Bind(x => string.Join("\r\n", x));
```

This is a workable solution, but it's a bit ugly. There's a few repeated steps that exist only to alow the connection object to be persisted between Bind operations. It’s harming the readability of the function.

It’s also not pure, if you think about it. The function is having to create a database connection, which is a form of side-effect.

There is another functional structure we can use to solve all of these problems - the Reader Monad. It’s functional programming’s answer to dependency injection, but on the functionl-level, rather than into a class.

In the case of the function above, it’s the IDbConnection that we want to inject in, so it can be instantiated elsewhere, leaving the MakeOrderReport pure - i.e. free of any side effects.

Here’s an incredibly simple use of a Reader:

```
var reader = new Reader<int, string>(e => e.ToSt
var result = reader.Run(100);
```

What we’ve defined here is a Reader which takes a function that it stores, but doesn’t execute. The function has an “environment"variable type as its parameter - this is the currently unknown value we’re going to inject in the future, and it returns a value based on that parameter, in our case an integer.

The “int → string” function is stored in the Reader on the first line, then on the second line we call the “Run” function which provides the missing environment variable value, here 100. Since the environment has finally been provided, the Reader can therefore use it to return a real value.

Since this is a Monad, that also means we should have Bind functions to provide a flow. This is how they'd be used:

```
var reader = new Reader<int, int>(e => e * 100)
 .Bind(x => x / 50)
 .Bind(x => x.ToString());

var result = reader.Run(100);
```

Note that the type of the "reader" variable is `Reader<int, string>`. That's because every Bind call places a wrapper around the previous function which has the same alternate return type, but a different parameter.

In the first line with the parameter `e => e * 100`, that function wil be exuecute later, afer the Run And, so on…

This is a more realistic use of the Reader:

```
public Customre GetCustomerData(string userName,
 new Reader(this.customerRepo.GetCustomer(userNar
 .Run(db);
```

Alternatively, you can actually simply return the Reader, and allow the outside world to continue using Bind calls to modify it further before the Run function turns it into a proper value.

```
public Reader<IdbConnection, User> GetCustomerDat
  new Reader(this.customerRepo.GetCustomer(userNa
```

This way, the same function can be called many times, with the Reader's Bind function used to convert it to the actual type I want.

For example, if I wanted to get the customer's order data:

```
var dbConn = this.DbConnectionFactory.GetConnecti

var orders = this._customerRepo.GetCustomerData('
 .Bind(X => x.OrderData.ToArray())
 .Run(dbConn);
```

Think of it as you're creating a box that can only be opened by inserting a variable of the correct type.

The use of the Reader also means it's easy to inject a mock IDbConnection into these functions and write unit tests based on them.

Depending on how you want to structure your code, you could even consider exposing Readers on interfaces. It doesn't have to be a depenency like a DbConnection that you pass in, it could be an Id value for a database table, or anything you like. Something like this, perhaps:

```
public interface IDataStore
{
 Reader<int,Customer> GetCustomerData();
 Reader<Guid,Product> GetProductData();
 Reader<int,IEnumerable<Order>> GetCustomerOrders
}
```

There are all sorts of ways you could use this, it's all a matter of what suits you, and what you're trying to do.

In the next section, I'll show a variation on this idea - the State Monad.

State

In principle, a State Monad is very similar to a Reader. A container is defined that requires some form of state object to convert itself into a proper final piece of data. Binds are usable to provide additional data transformations, but nothing will happen until the State is provided.

What makes it difference is two things:

1. Instead of an “Environment” type, it’s known as the “state” type
2. There are two items, not one being passed between Bind operations.

In a Reader, the original Environment type is only seen at the beginning of the chain of Binds. With a State Monad, it persists all the way through to the end. The State type, and whatever the current value is set to are stored in a Tuple which is passed from one step to the next. Both the value and the State can be replaced with new values each time. The Value can change types, but the State is a single type throughout the whole process that can have its vaules updated, if required.

You can also arbitrarily fetch or replace the State object in the State Monad at any time using functions.

My implementation doesn’t strictly adhere to the way you’ll see it in languages like Haskell, but I’d argue that implementations of that kind are a pain in C#, and I’m not convinced that there’s any point to doing it. The version I’m showing you here could well have some use in daily C# coding.

```
public class State<TS, TV>
{
        public TS CurrentState { get; init; }
        public TV CurrentValue { get; init; }
        public State(TS s, TV v)
        {
                CurrentValue = v;
                CurrentState = s;
                CurrentState = s;
        }
}
```

The State monad doesn’t have multiple states, so there’s no need for a base abstract class. It’s just a simple class with two properties - a Value and a State (i.e. the thing we’ll pass along through every instance).

The logic has to be implemented in extension methods:

```
public static class StateMonadExtensions
{

public static State<TS, TV> ToState<TS, TV>(this
        new(@this, value);

public static State<TS, TV> Update<TS, TV>(
         this State<TS, TV> @this,
         Func<TS, TS> f
        ) => new(f(@this.CurrentState), @this.Cu
}
```

As usual, not a lot of code to implement, but plenty of interesting effects.

This is the way I’d use it:

```
public IEnumerable<Order> MakeOrderReport(string
 this.connFactory.MakeDbConnection().ToState(use
 .Bind((s, x) => this.customerRepo.GetCustomer(s,
 .Bind((s, x) => this.orderRepo.GetCustomreOrder
```

The idea here is that the state object is being passed along the chain as *s*, and the result of the last Bind is passed as *x*. Based on both of those values, you can determine what the next value should be.

This just leaves out the ability to update the current state. I'd do that, with this extension method:

```
public static State<TS, TV>Update<TS,TV>(
 this State<TS,TV> @this,
 Func<TS, TS> f
) => new(@this.CurrentState, f(@this.CurrentStat
```

Here's a simple example for the sake of illustration:

```
var result = 10.ToState(10)
        .Bind((s, x) => s * x)
        .Bind((s, x) => x - s) // s=10, x = 90
        .Update(s => s - 5) // s=5, x = 90
        .Bind((s, x) => x / 5); // s=5, x = 18
```

Using this method, you can have arrow functions with a few bits of state that will flow from one Bind operation to the next, and which you can even update when needed. It prevents you from being either forced to turn your neat arrow function into a full-function with curly braces, or passing a large, ungainly Tuple containing the readonly data through each Bind.

This implementation has left off the form you"ll find in Haskell, where the initial State value is only passed in when the complete chain of Binds has been defined, but I'd argue that in a C# context this version is more useful, and certainly an awful lot easier to code!

Maybe a State?

You may notice in the previous code sample, there is no functionality to use the Bind function like a Maybe, to capture error conditions and null results coming back in one or other of several possible states. Is it possible to merge the Maybe and the Reader into a single monad that both persists a State object **and** handles errors?

Yes, and there are several ways you could accomplish it, depending on how exactly you're planning to use it. I'll show you my preferred solution. First, I'd adjust the State class so that instead of a value, it stores a Maybe containing the value.

```
public class State<TS, TV>
{
        public TS CurrentState { get; init; }
        public Maybe<TV> CurrentValue { get; init
        public State(TS s, TV v)
        {
                CurrentValue = new Something<TV>(
                CurrentState = s;
        }
}
}
```

Then Id adjust the Bind function to take into account the Maybe, but without changing the signature of the function:

```
public static State<TS, TNew> Bind<TS, TOld, TNew
        this State<TS, TOld> @this, Func<TS, TOld
        new(@this.CurrentState, @this.CurrentValu
```

The usage is just about exactly the same, except that Value is now of type Maybe<T>, instead of simply T. That only really affects the return value from the container function, though:

```
public Maybe<IEnumerable<order>> MakeOrderReport(
 this.connFactory.MakeDbConnection().ToState(user
 .Bind((s, x) => this.customerRepo.GetCustomer(s,
 .Bind((s, x) => this.orderRepo.GetCustomerOrders
```

Whether you want to merge the concepts of the Maybe and the State Monad in this way, or would rather keep them separately is entirely down to you.

If you do follow this approach, you'd just need to make sure to use a Switch expression to translate the Maybe into a single, concrete value at some point.

A last thing to bear in mind too - The CurrentValue object of the State Monad doesn't have to be data, it can be a `Func` delegate too, allowing you to have a bit of functionality ported betweeen Bind calls.

In the next section I'm going to look at what *else* might be a Monad you've already been using in C#.

Examples You're already using

Believe it or not, you've already most likely been using Monads for a while already if you've been working with C# for any amount of time. Let's take a look at a few examples.

Enumerable

If an Enumerable isn't a Monad, it's as close as it gets, at least once we invoke LINQ - which as we already know is developed based on functional programming concepts.

The Enumerable `Select` method operates on individual elements within an enumerable, but it still obeys the Left Identity law:

```
var op = x => x * 2;
var input = new [] { 100 };

var enumerableResult = input.Select(op);
var directResult = new [] { op(input.First()) };
// both equal the same value - { 200 }
```

and the Right Identity Law:

```
var op = x => x;
var input = new [] { 100 };

var enumerableResult = input.Select(op);
var directResult = new [] { op(input.First()) };
// both equal the same value - { 100 }
```

That just leaves the Associativity law - which is still a necessity to be considered a true Monad. Does Enumerable follow that? It does, of course.

By use of SelectMany.

Consider this:

```
var createEnumerable = (int x) => Enumerable.Rang
var input = new [] { 100, 200 }
var output = input.SelectMany(createEnumerable);
// output = single dimension array with 300 eleme
```

There we have it, nested Enumerables outputted as a single Enumerable. That's the Associativity law. Ipso facto, QED, etc. Yeah. Enumerables are Monads.

Task

What about Tasks? Are they monads too? I bet you a beer of your choice[10] that they're absolutely monads, and I can prove it.

Let's run through the laws once again.

The Left identity law, a function call with a Task should match calling the function call directly. That's slightly tricky to prove, in that an async method always returns a type of `Task` or `Task<T>` - which is the same as `Maybe<T>` in many ways, if you think about it. It's a wrapper around a type that might or might not resolve to actual data. But, if we move back a level of abstraction, I think we can still demonstrate that the law is obeyed:

```
public Func<int> op = x => x * 2;
public async Task<int> asyncOp(int x) => await Ta

var taskResult = await asyncOp(100);
var nonTaskResult = op(100);
// The result is the same - 200
```

I'm not saying that this is C# code I'd necessarily be proud of, but it does at least prove the point, that whether I call `op` through an async wrapper method or not, the result is the same. That's the Left Identity Law confirmed. How about the right Identity Law? Honestly, that's roughly the same code again:

```
// Notice the function is simply returning x back
public Func<int> op = x => x;
public async Task<int> asyncOp(int x) => await Ta

var taskResult = await asyncOp(100);
var nonTaskResult = op(100);
// The result is the same as the initial input -
```

That’s the Identity laws settled. What about the equally-important Associativity law? Believe it or not, there is a way to demonstrate this with Tasks.

```
async Task<int> op1(int x) => await Task.FromResu
async Task<int> pp2() => await Task.FromResult(10
var result = await op1(await pp2());
// result = 1,000
```

Here we have a `Task<int>` being passed into another `Task<int>` as a parameter, but with nested calls to `await`, it’s all possible to flatten out to a simple `int` - which is the actual type of result.

Hopefully I’ve earned my beer? Mine’s a pint[11], please. A European-style half-litre is fine as well.

Other Structures

Honestly and truly - if you’re OK with my version of the Maybe monad, and you aren’t bothered about going further, then feel free to skip ahead to the next chapter. You can easily achieve most of what you’re likely to want to achieve with Maybe alone. I’m going to describe a few other kinds of Monad that exist out there in the wider functional programming language world, which you *might* want to consider implementing in C#.

These Monads might be of interest if squeezing out the last few vestiges of non-functional code from C# is your intention. They might also be of interest from a theoretical perspective. It's entirely up to you though, if you want to take the Monad concept further and continue to implement these.

Now, strictly speaking the version of the Maybe monad I've been building up over this chapter and the previous chapter was a mix of two different monads.

A true Maybe monad has only two states - Something (or Just) and Nothing (or Empty). That's it. The monad for handling error states is the Either (aka Result) monad. That has two states - Left and Right.

Right is the "happy" path, where every function passed into the Bind command worked, and all is right with the world.

Left is the "unhappy" path where an error of some kind occurred, and the error is contained in the Left.

The Left and Right naming convention is presumably coming from the recurring concept in many cultures that the Left hand is evil and the Right hand is good. This is even captured in bits of our language - the Latin word for "left" is literally "Sinister". In these enlightened days however, we no longer drive out left-handed folks[12] from our homes or whatever it was they used to do.

I won’t spell out that implementation here, you can basically achieve it by taking my version of the Maybe and removing the “Nothing” class.

Similarly, you can make a true Maybe by removing the Error class - although I’d argue that by combining the two into a single entity, you’ve got something that can handle just about any situation you’re likely to encounter when interacting with external resources.

Is my approach pure and fully correct classical functional theory? No. Is it useful in production code? 100% yes.

There are many more monads beyond the Maybe and Either, and if you move into a programming language like Haskell you’ll likely make regular use of them. Here are a few examples:

- Identity - a monad that simply returns back whatever value you feed in. These have their uses when getting into deeper functional theory in a more purely functional language, but don’t really have an application here in C#.
- State - used to run a series of operations on a value. Not entirely unlike a `Bind` method, but there’s also a state object that’s passed along as well, that is used as an additional object to the calculations. In C#, we’re just as well using a LINQ `Aggregate` function, or a Maybe `Bind` function with a tuple or something similar to pass the necessary state objects through.

- IO - Used to allow interactions with external resources without introducing impure functions. In C#, we can follow the Inversion of Control pattern (i.e. Dependency Injection) to allow us to get around any issues for testing, etc.
- Environment- Known as the Reader monad in Haskell. Often used for processes like Logging, to wrap away the side-effects. This is useful if you're trying to ensure your language is enforcing the strict rules of functional programming, but I don't see any benefit in C#.

As you can see from the list above, there are many other monads available in the world of functional programming, but I'd argue that most or all of them don't provide any real benefit to us. At the end of the day, C# is a hybrid functional/object-oriented language. It has been extended to allow support for functional programming concepts, but it's never going to be a purely functional language, and there's no benefit to trying to treat it as such.

I strongly recommend experimenting with the Maybe/Either monad, but beyond that, I honestly wouldn't bother, unless you're curious to see just how far you can push the idea of functional programming in C#[13]. It's not something for your production environment, though.

In the final section, I'll provide a complete worked example of how to use monads in an application.

A Worked Example

OK, here we go. Let's put it all together in one great, epic heap of monady-functional goodness. We've already talked holidays in our examples this chapter, so this time I'm going to focus on how we're actually going to get to the airport - this will need a series of look-ups, data transformations and all of which would normally require error handling and branching logic if we were to follow a more conventional, Object-oriented approach. I hope you'll agree that using functional programming techniques and monads, it looks quite considerably more elegant.

First-off, we need our interfaces. I'm not actually going to code each and every single dependency our code has, so I'm just going to define the interfaces we'll need:

```
public interface IMappingSystem
{
 Maybe<Address> GetAddress(Location l);
}

public interface IRoutePlanner
{
 Task<Maybe<Route>> DetermineRoute(Address a, Ad
}

public interface ITrafficMonitor
```

```
{
 Maybe<TrafficAdvice> GetAdvice(Route r);
}

public interface IPricingCalculator
{
 decimal PriceRoute(Route r);
}
```

Having done that, I’ll write the code to consume them all. The specific scenario I’m imagining is this - It’s the near future. Driverless cars have become a thing. To the extent that most people no longer own personal vehicles, and simply use an app on their phones to have a car brought out of the cloud of automated vehicles, direct to their homes[14].

The process is going to be something like this:

1. The initial inputs are the starting location & destination, provided by the user.
2. Each of these locations need to be looked up in a mapping system and converted to proper addresses
3. The user’s account needs to be fetched from the internal data store
4. The route needs to be checked with a traffic service
5. A pricing service has to be called to determine a charge for the journey
6. The price is returned to the user.

In code, that process might look something like this:

```
public Maybe<decimal> DeterminePrice(Location fro
{
 var addresses = this.mapping.GetAddress(from).Bi
   (From: x,
    To: this.mapping.GetAddress(to)));

 var route = await addresses.BindAsync (async x =

 var trafficInfo = route.Bind(x => this.trafficAc
 var hasRoadWorks = trafficInfo is Something<Traf
  s.Value.RoadworksOnRoute;

 var price = route.Bind(x => this.pricing.PriceRo
 var finalPrice = route.Bind(x => hasRoadWorks ? 

 return finalPrice;
}
```

Far easier this was, isn't it? Before I end this chapter, I want to unpick a few details of what's happening in the code sample, above.

Firstly, there's no error handling anywhere. Any of those external dependencies could result in an error being throwm, or no details being located in their respective data stores. The monad `Bind` function handles all of that logic. If - for example - the router was unable to determine a route (maybe a network error occurred), then the Maybe would be set to an `Error<Route>` at that point, and none of the subsequent operations would be executed. The final return type would be `Error<decimal>`, because the `Error` class is re-created at each step, but the actual `Exception` is passed between instances. The outside world is responsible for doing something with the final returned value, until that .

If we followed the OO approach to this code, then the function would most likely be 2-3 times longer, in order to include `Try/Catch` blocks and checks against each object to confirm that they're valid.

I've used Tuples in cases where I want to build up a set of inputs. In the case of the Address objects, this means that in the event that the first address isn't found, the lookup for the second won't be attempted. It also means that both of the inputs required by the second function to be run are available in a single location, which we can then access with another `Bind` call (assuming there's a real value returned by the address lookup).

The final few steps don't actualy involve calls to external dependencies, but by continuing to use the `Bind` function, anything inside its parameter lambda expression can be written with the assumption that there is a real value available, because if there weren't, the lambda wouldn't be executed.

And there we have it, a pretty much fully functional bit of C# code. I hope it was to your liking.

Conclusion

In this chapter, we explored the dreaded functional programming concept that has been known to make grown developers tremble in their inexpensive footware. All being well, this shouldn’t be a mystery to you any longer.

I’ve shown you how:

- to massively reduce the amount of required code
- to introduce an implicit error handling system.

All using the Maybe Monad, along with how to make one yourself.

In the next chapter, I’ll be taking a brief look at the concept of currying. I’ll see you on the next page.

I don’t know why that name gets used. I really don’t. It’s pretty common though, and something of a standard. Just bear with me.

I would say non-null, but integers default to 0 and booleans to false

See here to read the article, it’s worth your while:
https://fsharpforfunandprofit.com/rop/

or living Schrödinger’s cat. Actually, *is* that the safe option? I’ve owned cats, I know what they’re like!

Cheesecake preferably. Paul Hollywood would be very disappointed to learn I don’t like many cakes, but New York style Cheesecake is most certainly one of them!

You can read about that here: *https://automapper.org/* It can be used for quickly and easily converting between types, if that’s something you do a lot in your code

Not a kind of tea! Nor is it a porcine character from Dragonball

Except bool, which defaults to false and char which defaults to *\0*

I’m the imaginary owner, you can be the imaginary person that helps customers find what they want. Aren’t I generous?

I like dark ales, and european-style lagers. That strong stuff they make in Minnesota is terrific too

That’s 568ml. I’m aware other countries have other definitions of the word

My brother among them. Hi, Mark - you’ve been mentioned in a programming book! I wonder whether you’ll ever know about it?

C# rather than .NET in general - F# is .NET as well.

This all sounds great to me, I don’t much like driving & I get terribly car sick if I’m a passenger. I for one welcome our driverless car overlords.

Chapter 8. Currying and Partial Application

Currying and Partial Application are two more functional concepts that come straight out of old maths papers. The former has absolutely nothing to do with Indian food, delicious though it is[1] in fact it’s named after the pre-eminent American mathematician Haskell Brooks Curry, after whom no fewer than three programming languages are named[2].

Currying came from Haskell Curry’s work on combinatory logic, which served as one of the bases for modern functional programming. Rather than give a dry formal definition, I’ll explain by example. This is a bit of vaguely C#-like pseudocode for an add function:

```
public interface ICurriedFunctions
{
 decimal Add(decimal a, decimal b);
}

var curry = // some logic for obtaining an implen

var answer = curry.Add(100, 200);
```

In this example, we’d expect answer to simply be 300 (i.e. 100+200), which is indeed what it would be.

What if I were only to provide a single parameter, however? Like this:

```
public interface ICurriedFunctions
{
 decimal Add(decimal a, decimal b);
}

var curry = // some logic for obtaining an implen

var answer = curry.Add(100); // What could it be
```

In this scenario, if this were a hypothetical curried function, what do you think you'd have returned to you in *answer*?

There's a rule of thumb I've devised when working in functional programming - if there's a question, the answer is likely to be "functions". Which is the case here.

If this were a curried function, then the the *answer* variable would be a function. It would be a modified version of the original `Add` function, but with the first parameter now set in stone as the value 100 - effectively making it a new function, one that adds 100 to whatever you provide.

You might use it like this:

```
public interface ICurriedFunctions
{
 decimal Add(decimal a, decimal b);
}

var curry = // some logic for obtaining an implem

var add100 = curry.Add(100); // Func<decimal,deci

var answerA = add100(200); // 300 -> 200+100
var answerB = add100(0); // 100 -> 0+100
var answerC = add100(900); // 1000 -> 900+100
```

It’s basically a way to start with a function that has a number of parameters, and from it, create mutiple, more specific versions of that function. One single base function can become many different functions. You *could* compare it to the OO concept of inheritance, if you like? In reality it’s *nothing* at all like inheritance. There is only actually a single function with any logic behind it - the rest are effectively pointers to that base function holding parameters, ready to feed into it.

What exactly is the point of currying, though? How do you use it?

Let me explain…

Currying and large functions

In the “Add” example I gave above, we’ve only got a single pair of parameters, and so there are only two possibilities for what we could possibly do with them when Currying is possible:

1. Supply the first parameter, get back a function
2. Supply both parameters and get back a value

How would currying handle a function with greater than 2 base parameters? For this I’ll use an example of a simple CSV parser - i.e. something that takes a CSV text file, breaks it into records by line, then uses some delimeter (typically a comma) to break it up again into individual properties within the record.

Let’s imagine I’d written a parser function to load in a batch of book data:

```
// Input in the format:
//
//title,author,publicationDate
//The Hitch-Hiker's Guide to the Galaxy,Douglas A
//Dimension of Miracles,Robert Sheckley,1968
//The Stainless Steel Rat,Harry Harrison,1957
//The Unorthodox Engineers,Colin Kapp,1979

public IEnumerable<Book> ParseBooks(string fileNa
 File.ReadAllText(fileName)
   .Split("\r\n")
   .Skip(1) // Skip the header
   .Select(x => x.split(",").ToArray())
   .Select(x => new Book
   {
    Title = x[0],
    Author = x[1],
    PublicationDate = x[2]
   });

var bookData = parseBooks("books.csv");
```

This is all well and good, except that the next two sets of books have different formats. Books2.csv uses pipes instead of commas to separate fields, and Books3.csv comes from a Linux environment and has “\n” line endings instead of the Windows style “\r\n”.

We could get around this by creating 3 different functions that are near replicas of each other. I’m not keen on unnecessary replication though, since it adds too many problems for future developers that want to maintain the codebase.

A more reasonable solution is to add in parameters for everything that could possibly change. Like this:

```
public IEnumerable<Book> ParseBooks(
 string lineBreak,
 bool skipHeader,
 string fieldDelimiter,
 string fileName
) =>
 File.ReadAllText(fileName)
  .Split(lineBreak)
  .Skip(skipHeader ? 1 : 0)
  .Select(x => x.split(fieldDelimiter).ToArray()
  .Select(x => new Book
  {
   Title = x[0],
   Author = x[1],
   PublicationDate = x[2]
  });

var bookData = ParseBooks(Environment.NewLine, tr
```

Now, If I wanted to follow the non-functional approach to the use of this function, I’d have to fill in every parameter every possible style of CSV file, like this:

```
  var bookData1 = ParseBooks(Environment.NewLine, t
  var bookData2 = ParseBooks(Environment.NewLine, t
  var bookData3 = ParseBooks("\n", false, ",", "boo
```

What Currying actually means is to supply the parameters one at a time. Any calls to a curried function results either in a new function with one fewer parameters, or else a concrete value if all parameters for the base function have been supplied.

The calls with the full set of supplied parameters, from the previous code sample, could be replaced like this:

```
  // First some magic that curries the parseBooks f
  // I'll look into implementation details later, I
  // understand the theory for now.
```

```
var curriedParseBooks = ParseBooks.Curry();

// these two have 3 parameters - string, string,
var parseSkipHeader = curriedParseBooks(true);
var parseNoHeader = curriedParseBooks(false);

// 2 parameters
var parseSkipHeaderEnvNl = parseSkipHeader(Envir
var parseNoHeaderLinux = parseNoHeader("\n");

// 1 parameter each
var parseSkipHeaderEnvNlCommarDel = parseSkipHea
var parseSkipHeaderEnvNlPipeDel = parseSkipHeade
var parseNoHeaderLinuxCommarDel = parseNoHeaderL


// Actual data, Enumerables of Book data
var bookData1 = parseSkipHeaderEnvNlCommarDel("b
var bookData2 = parseSkipHeaderEnvNlPipeDel("boo
var bookData3 = parseNoHeaderLinuxCommarDel("boo
```

The point is that Currying turns a function with X parameters, into a sequence of X functions, each of which has a single parameter - the last one returning the final result.

You could even write the function calls above like this - if you really, really wanted to!:

```
var bookData1 = parseBooks(true)(Environment.NewL
var bookData2 = parseBooks(true)(Environment.NewL
var bookData3 = parseBooks(true)("\n")(",")("book
```

The point of the first example of currying is that we’re gradually building up a hyper-specific version of the function that only takes a file name as a parameter. In addition to that, we’re storing all of the intermediate versions for potential re-use in building up other functions.

What we’re effectively doing here is building up functions like a wall made of lego bricks, where each brick is a function. Or, if you want to think about it another way, there’s a family tree of functions, with each choice made at each stage causing a branch in the family:

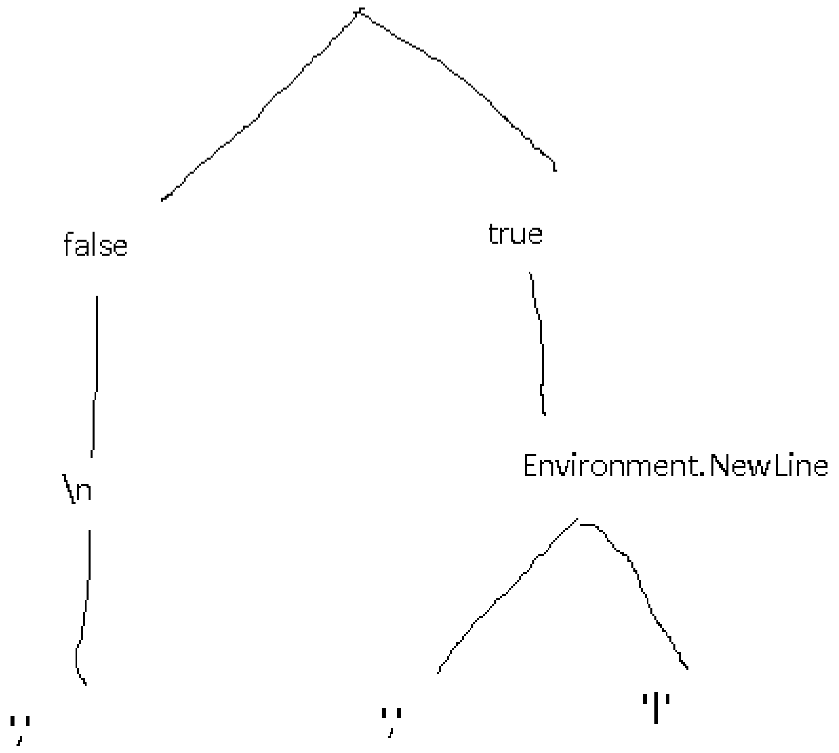


Figure 8-1. A The family tree of the parseBooks functions

Another example that might have uses in production is splitting up a logging function into multiple, more specific functions:

```
// For the sake of this exercise, the parameters
// an enum (log type - warning, error, info, etc
// containing a message to store in the log file
var logger = getLoggerFunction()
var curriedLogger = logger.Curry();

var logInfo = curriedLogger(LogLevel.Info);
var logWarning = curriedLogger(LogLevel.Warning)
var logError = curriedLogger(LogLevel.Error);

// You'd use them then, like this:

logInfo("This currying lark works a treat!");
```

There are a few useful features of this approach:

- We’ve actually only created one single function at the end of the day, but from it, managed to create at least 3 usable variations which can be passed around, requiring only a filename to be usable. That’s taking code re-use to an extra level!
- There are also all of the intermediate functions available too. These can either be used directly, or as a starting point for creating additional new functions.

There’s another use for currying in C# as well. I’ll discuss that in the next section.

Currying and Higher-Order functions

What if I wanted to use currying to create a few functions to convert between celsius and fahrenheit. What I’d do is start with curried versions of each of the basic arithmetic operations, like this:

```
// once again, the Currying process is just magic
// Keep reading for the implementation

var add = ((x,y) => x + y).Curry();
var subtract = ((x,y) => y - x).Curry();
var multiply = ((x,y) => x * y).Curry();
var divide = ((x,y) => y / x).Curry();
```

Using this, along with the map function from a previous chapter, we can create a fairly consise set of function definitions:

```
var celsiusToFahrenheit = x =>
 x.Map(multiply(9))
 .Map(divide(5))
 .Map(add(32));

var fahrenheitToCelsius = x=>
 x.Map(subtract(32))
  .Map(multiply(5))
  .Map(divide(9));
```

Whether you find any of this useful if largely dependent on use case - what you’re actually trying to achieve and whether currying fits in with it.

It’s available for you in C# now, as you can see. If, that is, we can find a way to implement it in C#…

Currying in .NET

So, the big question: More functional-based languages can do this natively with **all** functions in your codebase, can we do anything like this in .NET?

The short answer is no-ish.

The longer answer is yes, sort of. It's not as elegant as in a functional language (e.g. F#) where this is all available out of the box. We either need to hard-code it, create a static class, or else hack around with the language a bit and jump through a few hoops.

The hard-coded method assumes that you will only ever use the function in a curried manner, like this:

```
var Add = (decimal x) => (decimal y) => x + y;
var Subtract = (decimal x) => (decimal y) => y -
var Multiply = (decimal x) => (decimal y) => x *
var Divide = (decimal x) => (decimal y) => y / x
```

Note that there are two sets of arrows in each function, meaning that we've defined one `Func` delegate that returns another -i.e. the actual type is `Func<decimal,Func<decimal,decimal>>`. So long as you're using C# 10 or later, then you'll be able to take advantage of implicit typing with the `var` keyword, like in the example above. Older versions of C#

may need to implicity state the type of the delegates in the code sample above.

The second option is to create a static class that can be referenced from anywhere in the codebase. You can call it what you'd like, but I'm going with `F` for Functional.

```
public static class CurryingExtensions
{
  public static Func<T1, Func<T2, TOut>> Curry<T1
   Func<T1, T2, TOut> functionToCurry) =>
    (T1 x) => (T2 y) => functionToCurry(x, y);

  public static Func<T1, Func<T2, Func<T3, TOut>>
   Func<T1, T2, T3, TOut> functionToCurry) =>
    (T1 x) => (T2 y) => (T3 z) => functionToCurry

  public static Func<T1, Func<T2, Func<T3, Func<T
   Func<T1, T2, T3, T4, TOut> functionToCurry) =>
    (T1 x) => (T2 y) => (T3 z) => (T4 a) => funct
}
```

This effecively places layers of `Func` delegates between calls to the end function that's being curried, and the areas of code that use it that way.

The down side to this method is that we'll have to create a curry method for every possible number of parameters. My example covers functions with 2, 3 or 4 parameters. Functions with more than that would need another Curry method constructed, based on the same formula.

The other issue is that Visual Studio is unable to implicitly determine the type for the function being passed in, so it's necessary to define the function to be curried within the call to F.Curry, declaring the type of each parameter, like this:

```
var Add = F.Curry((decimal x, decimal y) => x + 
var Subtract = F.Curry((decimal x, decimal y) =>
var Multiply = F.Curry((decimal x, decimal y) =>
var Divide = F.Curry((decimal y, decimal y) => y
```

The final option - and my preferred option - is to use extension methods to cut down somewhat on the boilerplate code necessary. The definitions would look like this for 2, 3 and 4 parameter functions:

```
public static class Ext
{
        public static Func<T1,Func<T2, T3>> Curr
         this Func<T1,T2,T3> @this) =>
                (T1 x) => (T2 y) => @this(x, y)
```

```
            public static Func<T1,Func<T2,Func<T3,T4
             this Func<T1,T2,T3,T4> @this) =>
                     (T1 x) => (T2 y) => (T3 z) => @t

            public static Func<T1,Func<T2,Func<T3,Fur
             this Func<T1,T2,T3,T4,T5> @this) =>
                     (T1 x) => (T2 y) => (T3 z) => (
}
```

That's a fairly ugly block of code, isn't it? Good news is you can just shove that somewhere deep down at the back of your codebase, and largely forget it exists.

The usage would be like this:

```
// specifically define the function on one line
// it has to be stored as a `Func` delegate, rath
// Lambda expression
var Add = (decimal x, decimal y) => x + y;
var CurriedAdd = Add.Curry();

var add10 = CurriedAdd(10);
var answer = add10(100);
// answer = 110
```

So that’s currying. The eagle-eyed among you may have noticed that this chapter is called “Currying **and** Partial Application”.

What on earth is Partial Application? Well…since you asked so very nicely…

Partial Application

Partial application works along a very similar line to Currying, but there’s a subtle difference. The two terms are often even used - incorrectly - in exchange for each other.

Currying deals **exclusively** with converting a function with a set of parameters into a series of successive function calls, each with a single paramater (the technical term is a *unary* function).

Partial appliction on the other hand allows you to apply as many parameters in one go as you want. With data emerging if all of the parameters are filled in.

Returning to my earlier example of the parse function, these are the formats we’re working with:

- book1 - windows line endings, header, commas for fields
- book2 - Windows line endings, header, pipe for fields

- book3 - Linux ling endings, no header, commas for fields

With the currying approach, we’re creating intermediate steps for setting each parameter of Book3, even though it’s ultimately the only use of each of those parameters. We’re also doing the same for the SkipHeader and Line endings parameters for book1 and book2, even thought they’re the same.

It could be done like this to save space:

```
var curriedParseBooks = parseBooks.Curry();

var parseNoHeaderLinuxCommaDel = curriedParseBoo

var parseWindowsHeader = curriedParseBooks(true)(
var parseWindowsHeaderComma = parseWindowsHeader(
var parseWindowsHeaderPipe = parseWindowsHeader('

// Actual data, Enumerables of Book data
var bookData1 = parseWindowsHeaderComma("books.cs
var bookData2 = parseWindowsHeaderPipe("books2.cs
var bookData3 = parseNoHeaderLinuxCommaDel("books
```

But it’s much cleaner if we can just use partial application to apply the 2 parameters neatly.

```
// I'm using an extension method called Partial t
// parameters.  Check out the next section for in

var parseNoHeaderLinuxCommarDel = ParseBooks.Part

var parseWindowsHeader =
 curriedParseBooks.Partial(true,Environment.NewLi
var parseWindowsHeaderComma = parseWindowsHeader
var parseWindowsHeaderPipe = parseWindowsHeader.P

// Actual data, Enumerables of Book data
var bookData1 = parseWindowsHeaderComma("books.cs
var bookData2 = parseWindowsHeaderPipe("books2.cs
var bookData3 = parseNoHeaderLinuxCommarDel("book
```

I think that's pretty elegant as a solution, and it still allows us to have re-usable intermediate functions where we need them, but still only a single base function.

In the next section, I'll show you how to actually implement this.

Partial Application in .NET

This is the bad news. There's absolutely no way whatsoever to elegantly implement Partial Application in C#. What you're going to have to do is create an extension method for each and every combination of the number of parameters going in to the number of parameters going out.

In the example I just gave, I’d need:

- 4 parameters to 1 for `parseNoHeaderLinuxCommaDel`
- 4 paramters to 2 for `parseWindowsHeader`
- 2 parameters to 1 for `parseWindowsHeaderComma` and `parseWindowsHeaderPipe`

Here’s what each of those examples would look like:

```
public static class PartialApplicationExtensions
{
// 4 parameters to 1
 public static Func<T4,TOut> Partial<T1,T2,T3,T4,
  this Func<T1,T2,T3,T4,TOut> f,
  T1 one, T2 two, T3 three) => (T4 four) => f(one

// 4 parameters to 2
 public static Func<T3,T4,TOut>Partial<T1,T2,T3,T
  this Func<T1,T2,T3,T4,TOut> f,
  T1 one, T2 two) => (T3 three, T4 four) => f(one

 // 2 parameters to 1
 public static Func<T2, TOut> Partial<T1,T2,TOut>
  this Func<T1,T2,TOut> f, T1 one) =>
```

```
    (T2 two) => f(one, two);

  }
```

If you decide that partial application is a technique you’d like to persue, then you could either add Partial methods to your codebase as you feel they’re needed, or else put aside a block of time to create as many as you think you’re ever likely to need.

Conclusion

Currying and Partial Application are two powerful, related concepts in functional programming. Sadly they’re not available natively in C#, and aren’t ever likely to be.

They can be implemented by the use of static classes or extension methods, which add some boilerplate to the codebase - which is ironic, considering that these techniques are intended in part to reduce boilerplate.

Given that C# doesn’t support higher-order functions to the same level as F# and other functional languages. C# can’t necessarily pass functions around unless they’re converted to `Func` delegates.

Even if functions are converted over to `Func`then the Roslyn compiler can’t always determine parameter types correctly. T hese techniques will never be as useful in the C# world as they are in other languages. Despite that though, they have their uses in reducing boilerplate, and in enabling a greater level of code re-usability than would otherwise be possible.

The decision to use them or not is a matter of personal preference. I wouldn’t regard them as essential for functional C#, but they may be worth exploring nevertheless.

In our next chapter, we’ll be exploring the deeper mysteries of indefinite loops in functional C#, and what on earth Tail Optimised Recursion Calls are.

food tip: If you’re ever in Mumbai, try a Tibb’s Frankie from Shivaji Park, you won’t regret it!

Haskell, obviously, but also Brook and Curry, two lesser-known languages

Chapter 9. Indefinite Loops

We've seen in previous chapters how functional programming replaces `For` and `ForEach` loops with LINQ functions like `Select` or `Aggregate`, and that's absolutely terrific - provided you are working with a fixed-length array, or an `Enumerable` that will determine for itself when it's time to finish iterating.

What do you do though, when you aren't at all sure how long you'll want to iterate for? What if you're iterating indefinitely until a condition is met?

Here's an example of some non-functional code that shows the sort of thing I'm talking about. I'm imagining some code here for a game of Monopoly. You're stuck in jail, you naughty individual! The rules for getting out are doing one of the following:

- Pay 50 in whichever currency you play in (Pounds Sterling for me)
- Roll a double
- Use a "Get out of Jail Free" card[1]

In a real game of Monopoly, there are other player's turns to consider, but I'm simplifying it down to looping indefinitely until one of these conditions are met. I'd probably add some validation logic in here too if I were doing this for real, but once again, I'm keeping it simple.

```
var inJail = true;
var inventory = getInventory();
var rnd = getRandomNumberGenerator();

while(inJail)
{
  var playerAction = getAction();
  if(playerAction == Actions.PayFine)
  {
   inventory.Money -= 50;
   inJail = false;
  }
  else if(playerAction == Actions.GetOutOfJailFre
  {
   inventory.GetOutOfJailFree -= 1;
   inJail = false;
  }
  else if(playerAction == Actions.RollDice)
  {
   var dieOne = rnd.Random(1, 6);
   var dieTwo = rnd.Random(1,6);
   inJail = dieOne != dieTwo; // Stay in jail if
  }
}
```

You can't possibly do the above with a `Select` statement, it's simply not possible. We can't say when the criteria will be met, and we'll continue to iterate around the `While` loop until one of them are.

How can we make this functional? A `While` loop is a statement (specifically a control flow statement), and as such not preferred by functional programming languages.

There are a few options, and I'll describe each of them, but this is one of those areas where some sort of trade-off is required. Each of the choices have consequences, and I'll do my best to consider their respective pros and cons.

Buckle up your seatbelts, here we go…

Recursion

The classic functional programming method for handling indefinite loops is to use recursion. In brief, for those of you unfamiliar with it - Recursion is the use of a function that calls itself. There will be a condition of some sort too that determines whether there should be another iteration, or whether to actually return data.

If the decision is made at the end of the recursive function, this is known as *tail recursion*.

A purely recusive solution to the Monopoly problem might look like this:

```
public record Inventory
{
 public int Money { get; set; }
 public int GetOutOfJail { get; set; }
}

// I'm making the Inventory object a Record to ma
// a bit easier to be functional
var inventory = getInventory();
var rnd = getRandomNumberGenerator();

var updatedInventory = GetOutOfJail(inventory);
```

```
private Inventory GetOutOfJail(Inventory oldInv)
{
 var playerAction = getAction();
 return playerAction switch
 {
  Actions.PayFine => oldInv with
  {
   Money = oldInv.Money - 50
  },
  Actions.GetOutOfJailFree => oldInv with
  {
   GetOutOfJail = oldInv.GetOutOfJail - 1
  },
  Actions.RollDice =>
  {
   var dieOne = rnd.Random(1, 6);
   var dieTwo = rnd.Random(1,6);
   // return unmodified state, or else
   // iterate again
   return dieOne == dieTwo
    ? oldInv
    : GetOutOfJail(oldInv);
  }
 };
}
```

Job done, right? Not really, and I would think very carefully before using a function like the one above. The issue is that every nested function call adds a new item onto the Stack in the .NET runtime, and if there are a lot of recursive calls, then that can either negatively effect performance or else kill the application with a Stack Overflow Exception.

If there are guaranteed only to be a handfull of iterations, then there’s nothing fundamentally wrong with the recursive approach. You’d also have to be sure that this is revisited if the code’s usage is ever significantly changed following an enhancement. It could turn out that this rarely used function with a few iterations one day turns into something heavily used with hundreds of iterations. If that ever happens, then the business might wonder why their wonderful application suddenly becomes near unresponsive almost overnight.

So, like I said, think very carefully. This has the advantage of being relatively simple and not requiring you to write any boilerplate to make it happen.

F#, and many other more strongly functional languages, have a feature called *Tail Optimised Recursion Calls*, which means it’s possible to write recursive functions without them exploding the stack. This isn’t available in C# however, and there are no plans to make it available in the future, either. Depending on the situation, the F# optimization will either create Intermediate Language (IL) code with a `while(true)` loop, or else make use of an IL command called `goto` to physically move the execution environment's pointer back to the beginning of the loop.

I did investigate the possibility of referencing a generic Tail Optimised Recursion Call from F# and exposing it via a compiled DLL to C#, but that has its own performance issues that make it a waste of effort.

There's another possibility I've seen discussed online, and that's to add a post-build event that directly manipulates the .NET Intermediate Language that C# compiles to so that it makes retrospective use of the F# Tail Optimization feature. That's very clever, but sounds too much like hard work to me. It would be a potential extra maintainance task too.

In the next section, I'll look into a technique to simulate Tail Optimised Recursion Calls in C#.

Trampolining

I'm not entirely sure where the term *Trampolining* came from, but it pre-dates .NET. The earliest references I could find were academic papers from the 90s, looking at implementing some of the feaures of LiSP in C. I'd guess it's a little older even than that, though.

The basic idea is that you have a function that takes a *thunk* as a parameter - a thunk being a block of code stored in a variable. In C# these are implemented as `Func` or `Action`.

Having got the thunk, you create an indefinite loop with `while(true)` and some way of assessing a condition that determines whether the loop should terminate. This might be done with an additional `Func` that returns a `bool` or else some sort of wrapper object that needs to be updated with each iteration by the thunk.

But at the end of the day, what we're looking at is basically hiding a `while` loop at the back of our codebase. It's true that `while` isn't purely functional, but this is one of those places where we might need to compromise. Fundamentally, C# is a hybrid language, supporting both OO and FP paradigms. There are always going to be places where it's not going to be possible to have it behave in exactly the way F# does. This is one of them.

There are a number of ways that you could implement trampolining, but this is the one I'd tend to go for:

```
public static class FunctionalExtensions
{
        public static T IterateUntil<T>(
                this T @this,
                Func<T, T> updateFunction,
                Func<T, bool> endCondition)
        {
```

```
                        var currentThis = @this;

                        while (!endCondition(currentThis
                        {
                                currentThis = updateFunct
                        }

                        return currentThis;
                }
        }
```

By attaching to type `T` - a generic - this attaches to everything. The first parameter is a `Func` delegate that updates the type that `T` represents to a new form based on whatever rules the outside world defines. The second is another `Func` which returns the condition that will cause the loop to terminate.

Since this is a simple `While` loop, there aren’t any issues with the size of the stack. It’s not pure functional programming, though. It’s a compromise. At the very least though, it’s at least a single instance of a `while` loop that’s hidden somewhere, deep in the codebase. It may also be that one day, Microsoft will release a new feature that enables proper tail optimized recursion calls to be implemented somehow, in which case this function can be re-implemented and the code should continue to work as it did, but with one instance fewer of imperative code features.

using this version of indefinite iteration, the Monopoly code would now look like this:

```
// we need everything required to both update anc
// assess whether we should continue or not in a
// single object, so I'm considering it "state" r
// simply inventory
var playerState = geState();
var rnd = getRandomNumberGenerator();

var playerState.IterateUntil(x => {
 var action = GetAction();
  return action switch
  {
   Actions.PayFine => x with
   {
    Money = x.Money - 50,
    LastAction = action
   },
   Actions.GetOutOfJailFree => x with
   {
    GetOutOfJail = x.GetOutOfJail - 1,
    LastAction = action
   },
   _ => x with
   {
    DieOne = rnd.Random(1, 6),
    DieTwo = rnd.Random(1, 6)
   }
  }
},

x => x.LastAction == Actions.PayFine ||
 x.LastAction == Actions.GetOutOfJailFree ||
 x.DieOne == x.DieTwo

);
```

There is a third option, which you could consider which requires quite a bit more boilerplate, but which ultimately looks a little friendlier than the previous two versions.

Have a look, and see what you think.

Custom Iterator

The third option is to hack around with `IEnumerables` and `IEnumerators`. The thing about `IEnumerables` is they aren't actually arrays, they're just pointers to an item of data, and an instruction on how to get the next item. That being the case, we can create our own implementation of the IEnumerable interface, but with our own behavior.

In the case of our Monopoly example, we want an `IEnumerable` that's going to iterate until the user has selected a method to get out of jail, or else rolled a double.

We actually start in the outside world with an `IEnumerable`, but the `IEnumerable` only has a single function that has to be impemented: `GetEnumerator()`. The `IEnumerator` is the class that sits behind the scenes and actually does the work of enumerating. That's what we need to start with.

Anatomy of an Enumerator

This is what the `IEnumerator` Interface effectively looks like (there's some inheritance involved, so this isn't all actually contained in a single Interface):

```
public interface IEnumerator<T>
{
 object Current { get; }
 object IEnumerator.Current { get; }

 void Dispose();
 bool MoveNext();
 void Reset();
}
```

Each of these functions has a very specific job to do:

- Current / IEnumerator.Current - Get the current item. If iteration hasn’t begun, this typically returns NULL.
- Dispose - IEnumerator implements IDisposable
- MoveNext - Move from the current item to the next. The `bool` return value indicates whether another item **was** found. If there are no more items to iterate over, false is returned.
- Reset - move back to the beginning of the set of iteratable items.

Most of the time, an `Enumerator` is simply enumerating over an array, in which case I’d imagine the implementation *probably* works something like this:

```
public class ArrayEnumerable<T> : IEnumerator<T>
{
        public readonly T[] _data;
        public int pos = -1;
        public ArrayEnumerable(T[] data)
        {
                this._data = data;
        }

        private T GetCurrent() => this.pos > -1

        T IEnumerator<T>.Current => GetCurrent()

        object IEnumerator.Current => GetCurrent
```

```
            public void Dispose()
            {
                    //  Run!  Run for your life!
                    // Run before the GC gets us all
            }
            public bool MoveNext()
            {
                    this.pos++;
                    return this.pos < this._data.Leng
            }
            public void Reset()
            {
                    this.pos = -1;
            }
    }
```

I expect the real code is likely far more complicated than that, but this simple implementation gives you an idea of what sort of a job it is that the `Enumerator` does.

Implementing Custom Enumerators

Knowing how it works under the surface, you can see how it's possible to implement any behavior whatsoever that you'd like in an `Enumerable`. If you wanted to, you could do madness like an `Enumerable` that only iterates through every *other* item in an array by providing an alternative implementation of `MoveNext` like this:

```
public bool MoveNext()
{
        pos += 2;
        return this.pos < this._data.Length;
}

// This turns { 1, 2, 3, 4 }
// into { 2, 4 }
```

How about an `Enumerator` that loops over ever item twice, effectively creating a duplicate of each item when enumerating:

```
public bool IsCopy = false;
public bool MoveNext()
{
 if(this.IsCopy)
 {
  this.pos = this.pos + 1;
 }
 this.IsCopy = !this.IsCopy;

 return this.pos < this._data.Length
}
```

```
// This turns { 1, 2, 3 }
// into { 1, 1, 2, 2, 3, 3 }
```

Or an entire implementation that goes backwards, starting with an Enumerator outer wrapper:

```
public class BackwardsEnumerator<T> : IEnumerable
{
 private readonly T[] data;

 public BackwardsEnumerator(IEnumerable<T> data)
 {
  this.data = data.ToArray();
 }

 public IEnumerator<T> GetEnumerator()
 {
  return new BackwardsArrayEnumerable<T>(this.dat
 }

 IEnumerator IEnumerable.GetEnumerator() => GetEn
}
```

And aftwards the actual `Enumerator` that drives the backwards motion:

```
public class BackwardsArrayEnumerable<T> : IEnume
{
  public readonly T[] _data;

 public int pos;

 public BackwardsArrayEnumerable(T[] data)
 {
  this._data = data ?? new T[0];
  this.pos = this._data.Length;
 }

 T Current => (this._data != null && this._data.L
  this.pos >= 0 && this.pos < this._data.Length)
   ? _data[pos] : default;

 object IEnumerator.Current => this.Current;

 T IEnumerator<T>.Current => this.Current;

 public void Dispose()
 {
  // Nothing to dispose
 }

 public bool MoveNext()
 {
  this.pos = this.pos - 1;
  return this.pos >= 0;
 }

 public void Reset()
 {
  this.pos = this._data.Length;
 }
}
```

The usage of this backwards enumerable is pretty much exactly the same as a normal enumerable:

```
  var data = new[] { 1, 2, 3, 4, 5, 6, 7, 8 };
  var backwardsEnumerator = new BackwardsEnumerato
  var list = new List<int>();
  foreach(var d in backwardsEnumerator)
  {
   list.Add(d);
  }

  // list = { 8, 7, 6, 5, 4, 3, 2, 1 }
```

So, now that you've seen how easy it is to create your own `Enumerable` with whatever custom behavior you want, it should be easy enough to conjure up an `Enumerable` that iterates indefinitely.

Indefinitely Looping Enumerables

Try saying this section title ten times fast!

As you saw in the previous section though, there's no special reason that an `Enumerable` has to start at the beginning and loop to the end. We can make it behave in absolutely any way we care to.

What I want to do in this case, is - instead of an array - I want to pass in a single state object of some kind, along with a bundle of code (i.e. a Thunk, or `Func` delegate) for determining whether the loop should continue or not.

Working backwards, the first thing I'll make is the `Enumerator`. This is an entirely bespoke enumeration process, so I'm not going to make any effort to make it generic in any way. The logic I'm writing wouldn't make sense outside of a game state object.

I might want to do several different iterations in my hypothetical Monopoly implementation though, so I'll make the operation and loop termination logic somewhat generic.

```
public class GameEnumerator : IEnumerator<Game>
{
// I need this in case of a restart
 private Game StartState;
```

```
    private Game CurrentState;
    // old game state -> new game state
    private readonly Func<Game, Game> iterator;
    // Should the iteration stop?
    private Func<Game, bool> endCondition;
    // some tricky logic required to ensure the fina
    // game state is iterated.  Normal logic is that
    // the MoveNext function returns false, then the
    // anything pulled from Current, the loop simply
    private bool stopIterating = false;

    public GameEnumerator(Func<Game, Game> iterator,
     Func<Game, bool> endCondition, Game state)
    {
     this.StartState = state;
     this.CurrentState = state;
     this.iterator = iterator;
     this.endCondition = endCondition;
    }

    public Game Current => this.CurrentState;

    object IEnumerator.Current => Current;

    public void Dispose()
    {
     // Nothing to dispose
    }
```

```
  public bool MoveNext()
  {
   var newState = this.iterator(this.CurrentState)
   // Not strictly functional here, but as always
   // this topic, a compromise is needed
   this.CurrentState = newState;

  // Have we completed the final iteration?  That
  // reaching the end condition
   if (stopIterating)
       return false;

   var endConditionMet = this.endCondition(this.Cu
   var lastIteration = !this.stopIterating && endC
   this.stopIterating = endConditionMet;
   return !this.stopIterating || lastIteration;
  }

  public void Reset()
  {
  // restore the initial state
   this.CurrentState = this.StartState;
  }
 }
```

That's the hard bit done!! We have an engine under the surface that'll allow us to iterate through successive states until we're finished - whatever we decide "finished" means.

Next item required is the `IEnumerable` to run the `Enumerator`. That's pretty straightforward:

```
public class GameIterator : IEnumerable<Game>
{
 private readonly Game _startState;
 private readonly Func<Game,Game> _iterator;
 private readonly Func<Game,bool> _endCondition;

 public GameIterator(Game startState, Func<Game,
  Func<Game, bool> endCondition)
 {
  this._startState = startState;
  this._iterator = iterator;
  this._endCondition = endCondition;
 }

 public IEnumerator<Game> GetEnumerator() =>
  new GameEnumerator(this._startState, this._ite
 }

 IEnumerator IEnumerable.GetEnumerator() => GetE
}
```

Everything is now in place to carry out a custom iteration. I just need to define my custom logic, set up the iterator.

```
var playerState = getState();
var rnd = getRandomNumberGenerator();
var endCondition = (Game x) => x.x.LastAction ==
  x.LastAction == Actions.GetOutOfJailFree ||
  x.DieOne == x.DieTwo);

var update = (Game x) => {
  var action = GetAction();
    return action switch
  {
   Actions.PayFine => x with
   {
    Money = x.Money - 50,
    LastAction = action
   },
   Actions.GetOutOfJailFree => x with
   {
    GetOutOfJail = x.GetOutOfJail - 1,
    LastAction = action
   },
   _ => x with
   {
    DieOne = rnd.Random(1, 6),
    DieTwo = rnd.Random(1, 6)
    }
  }
}

var gameIterator = new GameIterator(playerState,
```

There are a couple of options for how to handle the iteration itself, and I'd like to take a little time out to discuss each of those options in a little more detail.

Using Indefinite Iterators

Strictly speaking, as a fully-fledged `Iterator` any LINQ operation can be applied, as well as a standard `ForEach` iteration.

`ForEach` would probably be the simplest way to handle this iteration, but it wouldn't be strictly functional. It's up to you, if you want to compromise. It might look like this:

```
foreach(var g in gameIterator)
{
// store the updated state outside of the loop.
 playerState = g;

 // Here you can do whatever logic you'd like to
 // to message back to the player.  Write a messa
```

```
   // or whatever is useful for them to be promptec
  }

  // At the end of the loop here, the player is nov
  // the game can continue with the updatd version
```

That wouldn’t give me too many causes for concern in production code, honestly. But, what we’ve done is negated all of the work we’ve put into attempting to get rid of non-functional code from our codebase.

The other options involve the use of LINQ. As a fully-fledged `Enumerable`, our GameIterator can have any LINQ operations applied to it. Which ones would be the best, though?

`Select` would be an obvious starting place, but it might not entirely behave as you’d expect. Usage is pretty much the same as any normal `Select` list operation you’ve ever done before:

```
  var gameStates = gameIterator.Select(x => x);
```

The trick here is that we’re treating gameIterator as an array, so `Select` -ing from it will result in an array of game states. What you’ll basically have is an array of every intermedate step the user has gone through, finishing with the final state in the last element.

The easy way to reduce this down to simply the final state is to substitute `Select` for `Last`:

```
var endState = var gameStates = gameIterator.Las
```

This assumes, of course that you aren't interested in the intermediate steps. It might be that you want to compose a message to the user for each state update, in which case you might want to select, and provide a transformation.

Something like this, perhaps:

```
var messages = gameIterator.Select(x =>
 "You chose to do " + x.LastAction + " and are "
  (x.InJail ? "In Jail" : "Free to go!");
);
```

That eradicates the actual game state, though, so possibly `Aggregate` might be a better option:

```
var stateAndMessages = (
 Messages: Enumerable.Empty<string>(),
 State: playerState
);
```

```
var updatedStateAndMessages =
 stateAndMessages.Aggregate(stateAndMessages, (ac
  acc.Messages.Append("You chose to do " + x.Last
   (x.InJail ? "In Jail" : "Free to go!")),
   x
 ));
```

The *x* in each iteration of the `Aggregate` process is an updated version of the Game State, and it'll carry on aggregating until the declared end condition is met. Each pass appends a message to the list, so what you finally get at the end is a `Tuple` containing an array of strings, which are messages to pass to the player, and the final version of the game state.

Bear in mind that any use of LINQ statements that will terminate the iteration early in some manner - `First`, `Take`, etc. will also premeturely end this iteration proces, possibly in our instance with the player still in jail.

Of course, this might be a behavior you actually want! Maybe you're restricting the player to just a couple of actions before moving onto another part of the game, or another player's turn. Something like that.

There are all sorts of possibilities for the logic you could come up with, playing with this technique.

Conclusion

We’ve looked into how we can implement indefinite iteration in C# without making use of the `ForEach` statement, which results in cleaner code with fewer possible side effects of execution.

It’s not strictly possible to do this purely functionally, and several options are available - all of which carry some level of compromise to the functional paradigm, but this is the nature of working in C#.

Which option - if any - you wish to make use of is entirely a matter of personal choice and whatever constraints apply to your project.

Please be very cautious with the use of recursion, though. It’s a fast method of iteration that’s purely functional, but if you aren’t careful, it can lead to significant performance issues when it comes to memory usage.

In the next chapter, I’ll look at a nice way to take advantage of pure functions to improve performance in your algorithms.

What are these supposed to be, anyway?

Chapter 10. Memoization

There are more advantages to pure functions than just producing predictable results. Granted that's a good thing to have, but there's another way to use that behavior to our advantage.

Memoization is somewhat like caching, specifically the `GetOrAdd` function from `MemoryCache`. What this does is takes a key value of some kind, and if that key is already present in the cache, it returns the object out. If it isn't, then you need to pass in a function that will generate the required value.

Memoization works to the exact same principle, except its scope might not extend beyond a single calculation.

Where this is useful is in a multi-step calculation of some kind that might be recursive, or involve the same calculations being performed multiple times for some reason.

Maybe the best way to explain this is with an example…

Bacon Numbers

Ever wanted an entertaining way to waste an afternoon or two? Have a look into Bacon Numbers. It's based on the idea that Kevin Bacon is the center of the acting universe, connecting all actors together. Like all roads lead to Rome, all actors somehow connect at some level to Kevin Bacon[1]. An actor's Bacon number is the number of film connections you have to work through in order to reach Kevin Bacon. Let's work through a few examples:

Kevin Bacon: easy. Bacon number of 0, because he **is** the big Bacon himself.

Tom Hanks: Bacon number of 1. He was with KB in one of my personal favorites, Apollo 13. Frequent Tom Hanks collaborator **Meg Ryan** is also 1, because she appeared with KB in In The Cut.

David Tennant: Bacon number of 2. He appeared with Colin Firth in St. Trinian’s 2. Colin Firth appeared in Where the Truth Lies with Kevin Bacon. That’s 2 films before we find a connection, so a Bacon number of 2. Believe it or not **Marilyn Monroe** also has a score of 2 due to KB appearing in JFK with Jack Lemon, who was also in Some Like it Hot.

Bollywood superstar **Aamir Khan** has a bacon number of 3. He was with living legend Amitabh Bacchhan in Bollywood Talkies. Amitabh was in The Great Gatsby with Toby McGuire. Toby McGuire was in Beyond all Boundaries with Kevin Bacon.

My Bacon number is Infinity! This is because I’ve never appeared in a film as an actor. Also, the holder of the highest Bacon number I’m aware of is William Rufus Shafter, an American Civil War general, who also appeared in a non-fiction film made in 1989, which still secures him a Bacon Number. It’s a whopping great 10!!

Right, hopefully you understand the rules.

Let’s imagine you wanted to work out which out of these actors had the lowest Bacon Number programatically. Something like this:

```
var actors = new []
{
 "Tom Hanks",
 "Meg Ryan",
 "David Tennant",
 "Marilyn Monroe",
 "Aamir Khan"
};

var actorsWithBaconNumber = actors.Select(x => (a

var report = string.Join("\r\n", actorsWithBaconN
 x.a+ ": " + x.b);
```

There are any numbers of ways that GetBaconNumber could be calculated. Most likely using a web API of flim data of some kind. There are more advanced "shortest path" algorithms out there, but for the sake of simplicity I'll say it's something like this:

- Get all of Kevin Bacon's films. Assign all actors in these films a bacon number of 1. If the target actor (e.g. Tom Hanks) is among them, return an answer of 1. Otherwise continue
- Take each of the actors from the previous step (excluding Kevin Bacon himself), get a list of all of their flims not already checked. Assign all actors from these films not already assigned a number a value of 2.
- And so on in iterations, with each set of actors being assigned progressively higher values until we finally reach the target actor and return their number.

Since there’s an API at work to calculate these numbers, ever actor whose filmography we download, or every film whose cast list we download, all have a significant cost in processing time.

Further to that, there is an awful lot of overlap between these actors and their films, so unless we step in and do something, we’re going to be checking the same film multiple times.

One option is to create a state object to pass into an Aggregate function of some kind. It’s an indefinite loop, so we’d also need to select one of the options for compromising on the functional principles and allowing a loop of this kind.

It might look something like this (N.B. I’ve made up the web API, so don’t expect this to work in a real application):

```
public int CalculateBaconNumber(string actor)
{

        var initialState = (
                checkedActors: new Dictionary<str
                actorsToCheck: new[] { "Kevin Bac
                baconNumber: 0
        );

        var answer = initialState.IterateUntil(
                x => x.checkedActors.ContainsKey(
                acc => {
                        var filmsToCheck = acc.ac
                        var newActorsFound = film
                                .Distinct()
                                .ToArray();

                        return (
                                acc.checkedActors
                                        .Where(x
                                        .Select(x
                                        .ToArray(
                                        .ToDictio

                                newActorsFound.Se
                                        .SelectMa

                                acc.baconNumber +
                        );
                });
        return answer.checkedActors[actor];
}
```

This is fine, but could be better. There’s a lot of boilerplate code that’s concerned with tracking whether or not an actor has already been checked or not. Lots of uses of Distinct

With Memoization we get a generic version of the check, like a cache, but it exists within the scope of the calculation we're performing, and doesn't persist beyond it. If you do want the saved, calculated value to persist between calls to this function, then `MemoryCache` maybe a better choice.

I could create a Memoized function to get a list of films they've been in like this:

```
var getAllActorsFilms = (String a) => this._filmG
var getAllFilmsMemoized = getAllActorsFilms(getAl

var kb1 = getAllFilmsMemoized("Kevin Bacon");
var kb2 = getAllFilmsMemoized("Kevin Bacon");
var kb3 = getAllFilmsMemoized("Kevin Bacon");
var kb4 = getAllFilmsMemoized("Kevin Bacon");
```

I called the same function there 4 times, and by rights it should have gone away to the film data respository and fetched a fresh copy of the data 4 times. In fact it did it only a single line when `kb1` was populated. Every time since then a copy of the same data was returned.

Note also, by the way, that the Memoized version and original version are on separate lines. That's a limitation of C#. You can't call an extension method on a function, only on a `Func` delegate, and the arrow function isn’t a `Func` until it has been stored in a variable.

Some functional languages have out-of-the-box support for Memoization, but F# doesn’t, oddly enough.

This is an updated version of the Bacon Number calculation, this time taking advantage of the Memoization feature:

```
public int CalculateBaconNumber2(string actor)
{

        var initialState = (
                checkedActors: new Dictionary<str
                actorsToCheck: new[] { "Kevin Bac
                baconNumber: 0
        );

        var getActorsFilms = GetAllActorsFilms;
        var getActorsFilmsMem = getActorsFilms.Me

        var answer = initialState.IterateUntil(
                x => x.checkedActors.ContainsKey(
                acc => {
                        var filmsToCheck = acc.ac
                        var newActorsFound = film
                                .Distinct()
```

```
                                        .ToArray();

                        return (
                                acc.checkedActors


                                        .ToArray
                                        .ToDicti

                                newActorsFound.Se
                                        .SelectM

                                acc.baconNumber
                        );
                });
        return answer.checkedActors[actor];
}
```

Literally the only difference here is that we’ve created a local version of the call to get film data from a remote resource, then memoized it, and thereafter referenced only the memoized version. This means that there are guaranteed to be no wastefull repeat requests for data.

Implmementing Memoization in C#

This is how you'd make a memoization function for a simple, single parameter function:

```
public static Func<T1, TOut> Memoize<T1, TOut>(th
{
        var dict = new Dictionary<T1, TOut>();
        return x =>
        {
                if (!dict.ContainsKey(x))
                        dict.Add(x, @this(x));
                return dict[x];
        };
}
```

This version of Memoize expects the "live data" function has only a single parameter of any type. To expect more parameters, further Memoize extension methods would be necessary, like this:

```
public static Func<T1, T2, TOut> Memoize<T1, T2,
{
        var dict = new Dictionary<string, TOut>(
        return (x, y) =>
        {
                var key = $"{x},{y}";
                if (!dict.ContainsKey(key))
                        dict.Add(key, @this(x, y
                return dict[key];
        };
}
```

Now, to make this work, I've assumed that `ToString` returns something meaningful, meaning that most likely it'll have to be a primitive type in order to work. The `ToString` method on a class tends to simply return a description of the *Type* of the class, not its properties.

If you absolutely have to memoize classes as parameters, then some creativity is needed. The easiest way to keep it generic is probably to add parameters to the `Memoize` function which require the developer to provide a custom `ToString` function. Like this:

```
public static Func<T1, TOut> Memoize<T1, TOut>(th
{
        var dict = new Dictionary<string, TOut>(
        return x =>
        {
                var key = keyGenerator(x);
                if (!dict.ContainsKey(key))
                        dict.Add(key, @this(x));
                return dict[key];
        };
}
```

```
public static Func<T1, T2, TOut> Memoize<T1, T2,
{
        var dict = new Dictionary<string, TOut>(
        return (x, y) =>
        {
                var key = keyGenerator(x, y);
                if (!dict.ContainsKey(key))
                        dict.Add(key, @this(x, y
                return dict[key];
        };
}
```

you might call it like this, in that case:

```
var getCastForFilm((Film x) => this.castRepo.Get
var getCastForFilmM = getCastForFilm.Memoize(x =
```

This is only possible though, if you keep your functions *Pure*. If there are side effects of any kind in your “live” func, then you might not necessariy get the results you expect. Depending on what those side effects are.

From a practical perspective, I wouldn’t be worried about adding logging into the “live” function, but I might be concerned if there were properties that were expected to be unique in every instance of the generated class.

There may be cases that you *want* the results to persist between calls to the `Memoize` function, in which case you'll also need to add a MemoryCache parameter and pass in an instance from the outside world. I'm not convinced there are many circumstances where that's a good idea, though.

Conclusion

In this chapter we looked at Memoization, what it is and how to implement it. It's a lightweight alternative to caching which can be used to drastically reduce the amount of time taken to complete a complex calculation with a lot of repetitive elements.

That's it for theory now! This isn't just the end of this chapter, but also of this entire part of the book. Turn over, if you dare, to enter…Part Three…

It probbaly isn't true, sorry Mr. Bacon. I still love you, though!

About the Author

Simon J. Painter has been developing professionally for far, far too long now (well, since 2005) and has worked with every version of .NET ever released (including Compact Framework—remember that?) in around a dozen different industries. As well as his day job, he also appears regularly at user groups and conferences to give talks on Functional Programming and general .NET topics. Before becoming a professional, Simon had been a coding enthusiast since he was old enough to read his Dad's copy of the Sinclair ZX Spectrum BASIC handbook. Besides code, he loves Playing Music, Cryptic Crosswords, Fighting Fantasy Gamebooks and far more coffee than is likely to be healthy for him. He lives in a small town in the UK, with his wife and children.