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

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.

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

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

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

978-1-492-09707-5

Preface

Functional Programming is not only 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 often include Functional content somewhere as a talking point these days.

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

With the growth of concepts like containerization and serverless applications, Functional 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.

The big question though - What is it? And, 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 has been written with a few different catagories of people in mind:

- Those of you 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#. Perhaps you're an F# developer looking for ways to keep using the Functional toys you're used to, or migrating to .NET from another functional, or functional-supporting language.
- Developers that have learned the basics of .NET and Object Orientated development, but want to take it further. Learn more advanced techniques to write better, more robust code.
- 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 8 colours - and one of them was black. I had the more advanced model with 48k of memory, though my Dad had the earlier version which had a single kilobyte 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 you effectively booted to the IDE, 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 add in 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[1], 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.

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 who read the early drafts and provided invaluable feedback. Thanks, guys!

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 *https://github.com/oreillymedia/title_title*.

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

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)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at LINK TO COME.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

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.

---

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

Chapter 1. Introduction

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 first 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.

Before we get cracking with some code, I’d like to talk 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 and 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.

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. Imperative and Object Orientated (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 T-SQL. When writing T-SQL statements you aren't concerned with what the order of operations are (it's not really SELECT then WHEN 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 might find some of the ideas coming up easier to grasp than those that haven't.

There are many, many more paradigms under these categories, but they're well beyond the scope of this book.

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

Variables in Functional code have their value set upon being defined, and after that point they are considered Constants, and the value may never be changed again. If a new value is required, then a new variable should be created, based on the old one. 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.

Higher-order Functions

These are Functions that are passed around as variables. Either as local variables, parameters to a function or return values from a function. The Func<T1,T2> delegate type is a perfect example of this. These functions can then 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. It's not, as the name suggests, that other paradigms don't function at all. 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".

Expressions - Not Statements

Functional is written out as a series of expressions - as mentioned above, more like mathematical working. There is no use of language statements that might alter the flow of execution, so we should ideally avoid the use of If, For, ForEach, While, etc. A ternary If-statement is allowed, but not the more complicated form with curly braces and nested code blocks.

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.

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

Functions that are developed with referential transparency in mind are known as "Pure Functions". These are functions with the following properties:

- They rely entirely on their own parameters and nothing else, I.e. there can be no reference to properties of the object they belong to, or any global or static functions or state.
- 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. No matter what state the system is in.

This is one of the features that massively increases the testability of functional code. It does mean that other methods 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. 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.

Recursion

This should be familiar to anyone that's ever written code to traverse a folder structure, or perhaps written an efficient sorting algorithm. Recursion 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.

Recursion is one of the methods Functional Programming uses as an alternative to statements like While and ForEach. There are some performance issues however, which is why some languages make use of Tail Recursion - more on this later. C# doesn't exactly support Tail Recursion, but there are still things we can do. Stick with me for now, all will become clear…

Pattern Matching

Switch statements with go-faster stripes. 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 have been enhancing it regularly.

It's switching where you can switch on the type of the object under examination, as well as its properties. It can be used to reduce a large, complex set of nested if-statements into a few fairly concise lines. It's an incredible, powerful feature, and one of my favourite things.

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

Seriously. Feels like purest craziness, doesn't it? There **is** strictly a state, but it's more of an emergent property of the system, rather than the OO approach of having a state object that's periodically updated.

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). An immutable object, which isn't updated, instead a function is created that takes the old state, a command, and any required parameters, then returns a new, updated state object based on that.

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[2]:

An Imperative Cake

This isn't real C#, it's just a sort've .NET themed pseudocode to give an impression of the imperitive solution to this imaginary problem.

```
Oven.SetTemperature(180);
for(int i ==0; i < 3; i++)
{
  bool isEggBeaten = false;
  while(!isEggBeaten)
  {
    Bowl.Beat();
    isEggBeaten = Bowl.IsStirred();
  }
}
for(int i == 0; i < 12; i++)
{
  OvenTray.Add(paperCase[i]);
```

```
      OvenTray.AddToCase(bowl.ExtractInG(55));
    }
    Oven.Add(OvenTray);
    Thread.PauseMinutes(25);
    Oven.ExtractAll();
```

For me, this represents typical convoluted imperical 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 Declaritive Cake

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

```
    Oven.SetTemperatureInC(180);
    var cakeBatter = EggBox.Take(3)
      .Each(e => Bowl.Add(e)
                    .Then(b =>
                        b.While(x => !x.IsStirred, x.Beat())
                      )
                    )
            .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 it was Functional Progamming 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.

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 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 2 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-orientated. 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[3]. 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.

Erlang

Not a pure Functional language, but there have been a few high-profile projects developed using it - RabbitMQ amongst them.

Is It Worth Learning a Pure Functional Language First?

I have heard people argue this point. Wouldn't it be best to learn Functional Programming in its pure form *first* then come to apply that learning to 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 entirely [4]. Latin is interesting, but *hard* and 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 exactly the same way, I feel that pure functional languages can be harder to learn than .NET, and they’re less likely to be something you can actually use immediately in your day job. 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#.

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 all sorts of Functional tricks and toys available straight out-of-the-box, 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. But…

…no, I don’t think it’s worth learning F# before giving Functional C#. It’s not because F# isn’t awesome. I don’t know F#, but everything I’ve seen of it looks great, and something I’d love to play with.

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 descision. 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 atendee of a big developer’s 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. Adding F# into your codebase is adding in a technology you may put 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.

It's my firm wish that this change 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 on the cards for you some time soon.

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

Offers the same level of support for Functional features as .NET. Further to that, there's a spin-off project called Scala that offers far more Functional features than Java itself does.

F#

There is also a .NET Functional language as well, of course. That's F#. I'm not planning to talk in any depth about F# in this book, for now just be aware that it exists. There are a few Functional features that can't be easily accomplished in C#, and F# exists if you absolutely can't live without them. We will be able to do nearly everything, though. F# is worth looking into if you want to take your Functional journey further than it's possible to in C#, but still stay in the .NET realm. It's also possible to have interopability between C# and F# libraries, so you can even 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 Delegates 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-orientated 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 hard one[5], and we often inherit complex codebases that we need to get to grips with quickly. The longer and harder it is to work out what a function is actually doing, the more money the business is losing. 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 favourite 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 Devleopment (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, or else if they do, then preventing any unexpected behaviour further on, and make it easier to report the issue accurately.

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.

Enables 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 (virtual or otherwise) the application runs instead on a sort've mini-Virtual Machine (VM) which is generated by a script at deploy time. It solves the "it worked on my machine" problem that is sadly all-to familiar to many developers. Many companies have software infrastructure that involves stacking up many instances of the same appliation in a arrays 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 devleopers as Azure Functions or AWS Lambdas. This is code that isn't deployed to a traditional 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 spend 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.

Simple, Easy Code

Fun! The final thing I'd like to mention is that Functional Programming is fun. Freed of the tedium of writing the amount of boilerplate required by Imperative code, as well as all the additional steps required to prevent unhandled exceptions, developers are freer to just get on with composing interesting code that's not going to cause anywhere near as many headaches once it passes into long-term maintenance.

That's not 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

Functional 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. 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 async 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 investigating 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 it's more stable and robust than applications coded with the Object-Orientated paradigm. If it's incredibly important that the system should stay up, and not have a crash to desktop 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 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, so compromise is necessary. The same goes for 3rd party modules imported from Nuget. 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 SMTP features of C# and the MainMessage class 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 virtual hardware at the app - and this is likely to be an order of magnitude cheaper than the additional developer time that would otherwise be required to develop, test, debug and maintain the codebase that is likely to take longer with procedural 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.

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 worth 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 Recognition
- 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 inginuity.

Come back just over the page to hear all about it. Same .NET time. Same .NET channel[6].

---

1 including Vanilla, and my personal favorite - Banana

2 with a little creative liberty taken

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

4 Although I **am** learning Latin. I'm a nerd. It's the sort've thing I do for fun.

5 at least that's what we tell our managers

6 or book, if we're being picky

Chapter 2. Getting Started - What can we do already?

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 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.

If you ask most people why they don't fancy getting into Functional Programming, they usually say one or more of the following:

- It looks funny
- It's hard
- I don't want to learn another language

The reality is that there is actually *less* to learn with Functional Programming than there is with Object Orientation. 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.

This is an absolutely trivial example to begin with, 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.

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]

```
var allFilms = GetAllFilms();
var actionFilms = new List<Film>();

foreach (var f in allFilms)
{
    if (f.Genre == "Action")
    {
        actionFilms.Add((f));
    }
}
```

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 indeed. I don’t know about you, but I’ve seen legacy code with functions over a thousand lines long. I won’t disagree that it should’t have been allowed to get into that state, but nevertheless, that was how it was. If this code sample were to appear at the top of a long function, the allFilms and actionFilms variables would both remain in scope, and thus in memory all that time, even if they aren’t in use.

Not only are they both in scope, but we’re holding two copies of all action films, one in the original allFilms variable, the other in this new actionFilms variable. That’s more memory than we strictly need to hold.

More importantly, 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.

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, and all optimisations are the coder's responsibility.

One of out 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 Functionally? I'd hope many of you might already guess at how you would re-write it.

```
var actionFilms = GetAllFilms()
                 .Where(x => x.Genre == "Action");
```

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.

Think about it, you're passing in functions as parameters - that's Higher Order Functions. In other words, Functions passed around like variables.

Linq doesn't change the source array, it returns a new array based on the old one - Immutability.

We've eliminated the use of a ForEach and an If - this satisfies the requirement for expressions instead of statements.

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.

The iteration could well be done by recursion to 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 list of data, creating a new array 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-Orientated Programming

A common feature of Functional code is that is 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.SomethingElse;
obj.PropertyB = sourceData.Ping * sourceData.Pong;

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.

In our example 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. It's 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(SourceData source) =>
    new ComplexCustomObject
    {
       PropertyA = source.Something + source.SomethingElse,
       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 list of data - so abstract that it doesn't contain any, 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 Loading*. Lazy is a good thing in development. Don't let anyone tell you otherwise.

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:

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

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

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. It isn't necessarily, though.

If I were to add some basic logging into each of those functions, you'd actually get back 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 Declaritive 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 => DoSomethingThree(x));
```

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)).ToArray();
var temp2 = input.Select(x => DoSomethingTwo(x)).ToArray();
 var finalAnswer = input.Select(x => DoSomethingThree(x)).ToArray();
```

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()[3], 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. Mostly for performance reason. 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.

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're reading this book, you're most likely aware of Select statements, and how to use them. There are a few tricks 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 has a series of select statements literally one after the other 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 only thing to get away from is the Object-Orientated 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

There's one final tricky case I'd like to cover here. 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 Jeunet")
                                .OrderByDescending(x =>
x.BoxOfficeRevenue);

 var i = 1;

 Console.WriteLine("The films of visionary French director");
 Console.WriteLine("Jean-Pierre Jeunet in descending order"
 Console.WriteLine(" of financial success are as follows:");

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

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

We're iterating through a list of complex objects that's already been ordered by Linq, so we can't use the Enumerable.Range trick here. Even a Tuple won't do, because we'd need a way to join the two arrays together. To make it worse, you don't know how long an Enumerable is until it's enumerated, so we wouldn't even know how long to make our Range list.

Instead we could 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, and will work as the equivalent of the "i" iterator in the previous code samples. This is how our functional version of the code looks:

```
 var films = GetAllFilmsForDirector("Jean-Pierre Jeunet")
                           .OrderByDescending(x =>
x.BoxOfficeRevenue);

 Console.WriteLine("The films of visionary French director");
 Console.WriteLine("Jean-Pierre Jeunet in descending order"
 Console.WriteLine(" of financial success are as follows:");

 var formattedFilms = films.Select((x, i) => $"{i} - {x.Title}");
 Console.WriteLine(string.Join(Environment.NewLine,
formattedFilms));

 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 a For or Foreach loop with a List. There are nearly always smarter, Declaritive ways to solve the problem.

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.

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.

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 Hitchcock");
var totalRevenue = films.Sum(x => x.BoxOfficeRevenue);
```

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.

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 Nolan");

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 aggregatingd down, and the current running total.

The seed doesn't have to be a primitive, like an integer or whatever, it can just as easily be a complex object. In order to remake the code sample, above, functionally, however, we just need a simple Tuple.

```
 var films = GetAllFilmsForDirector("Christopher Nolan");

 var (totalBudget, totalRevenue) = films.Aggregate(
                        (0.0M, 0.0M),
                        (runningTotals, x) => (
                               runningTotals.Item1 + x.Budget,
                               runningTotals.Item2 +
x.BoxOfficeRevenue
                           )
                    );
```

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

It's also an example of another concept important to Functional Programming - Recusion.

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 exits the application, or all available memory on the computer is consumed. As Treguard, the legendary Dungeon Master of the popular British Fantasy RPG gameshow *Knightmare* would put it: “Oooh, Nasty”.
- They tend to be fairly intense on the 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)
   {
     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 currentValue, int i = -1)
    =>
         currentValue == 0
             ? i
             : GetFirstPositionWithValueZero(currentValue +
    deltas[i], i + 1);

     return GetFirstPositionWithValueZero(10);
```

This is Functional now, but it's not really ideal. For a start, I've nested a function in a function. Delightfully recursive, but it's not very elegant.

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 Recursion* which allows the use of recursion without this memory usage problem.

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

    public ClassA(string propA, int propB, DateTime propC,
  IEnumerable<double> propD, IList<string> propE)
    {
      this.PropA = propA;
      this.PropB = propB;
      this.PropC = propC;
      this.PropD = propD;
      this.PropE = propE;
    }

  }
```

Is it immutable now? No, honestly it’s not. The integer property PropB and the IEnumerable PropD are fine, but everything else is still mutable. It’s true that we can’t actually outright replace any of them, but the List can still have its elements alterered, the string is effectively a char[], so anything can be done to it, and any date element of PropC can be changed too.

If we didn’t actually need to hold a mutable copy of PropE, we could easily replace it with an IEnumerable or IReadOnlyList, but that still leaves the issue of the string and DateTime fields.

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 set; }
    public IList<string> PropE { get; private set; }
    public SubClassB PropF { get; private set; }

    public ClassA(string propA, int propB, DateTime propC,
  IEnumerable<double> propD, IList<string> propE, SubClassB propF)
    {
      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.

My solution is far from perfect. I would simply *pretend* that Immutability exists in the project, and never change any object. There's nothing in older versions of 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 a lot of the 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[4]. 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 missing for each season.[5]

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.

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 of the
// file into an array where each line of the original file
// (i.e. each record) is an array element
var splitLines = text.Split(Environment.NewLine);

// Split each line into an array of fields, splitting the
// source array by the ',' character.  Convert to Array
// for each access.
var splitLinesAndFields = splitLines.Selct(x =>
x.Split(",").ToArray());

// Convert each string array of fields into a data class.
// parse any non-string fields into the correct type.
// Not strictly necessary, based on the final aggregation
// that follows, but I believe in leaving behind easily
// extendible code
var parsedObject = splitLinesAndFields.Select(x => new Story
{
   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 of Story
// objects for each season of the TV series
var groupedBySeason = parsedObject.GroupBy(x => SeasonNumber);

// Use a 3 field Tuple as the aggregate state:
// S (int) = the season number.  Not required for
//                the aggregation, but we need a way
//                to pin each set of aggregated totals
//                to a season
// NoEps (int) = the total number of episodes in all
//                serials in the season
// NoMisEps (int) = The total number of missing episodes
//                from the season
var aggregatedReportLines = groupedBySeason.Select(x =>
    x.Aggregate((S: x.Key, NoEps: 0, NoMisEps: 0),
      (acc, val) => (acc.S,
                        acc.NoEps + val.NumberOfEpisodes,
                         acc.NoMisEps +
val.NumberOfMissingEpisodes)
    )
);

// convert the Tuple-based results set to a proper
// object and add in the calculated field PercentageMissing
// not strictly necessary, but makes for more readable
// and extendible code
var report = aggregatedReportLines.Select(x => new ReportLine
{
   SeasonNumber = x.S,
    NumberOfEpisodes = x.NoEps,
    NumberOfMIssingEpisodes = x.NoMisEps,
    PercentageMissing = (x.NoMisEps/x.NoEps)*100
});

// format the report lines to a list of strings
var reportTextLines = report.Select(x => $"{x.SeasonNumber},
{x.NumberOfEpisodes}," +
"{x.NumberofMissingEpisodes},{x.PercentageMissing}");

// join the lines into a large single string with New Line
// characters between each line
var reportBody = string.Join(Environment.NewLine,
reportTextLines);
var reportHeader = "Season,No Episodes,No MissingEps,Percentage
Missing";

// the final report consists of the header, a new line, then the
reportbody
var finalReport = $"{reportHeader}{Environment.NewLine}
{reportTextLines}";
```

In case you're curious, the results would look something like this (I've added a few Tabs in to make it readable):

```
Season    No Episodes    No Missing Eps    Percentage Missing,
1          42              9                     21.4,
2          39              2                     5.1,
3          45              28                    62.2,
4          43              33                    76.7,
5          40              18                    45,
6          44              8                     18.2,
7          25              0                     0,
8          25              0                     0,
9          26              0                     0,

...
```

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

```
var reportTextLines = File.ReadAllText(filePath)
                      .Split(Environment.NewLine)
```

```
                              .Selct(x => x.Split(",").ToArray())
                              .GroupBy(x => x[0])
                              .Select(x =>
        x.Aggregate((S: x.Key, NoEps: 0, NoMisEps: 0),
          (acc, val) => (acc.S,
                              acc.NoEps + int.Parse(va[4]),
                               acc.NoMisEps + int.Parse(val[5]))
        )
    )
    .Select(x => $"{x.S}, {x.NoEps},{x.NoMisEps},
    {(x.NoMisEps/x.NoEps)*100}");


    var reportBody = string.Join(Environment.NewLine,
    reportTextLines);
    var reportHeader = "Season,No Episodes,No MissingEps,Percentage
    Missing";

    var finalReport = $"{reportHeader}{Environment.NewLine}
    {reportHeader}";
```

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.

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.

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 everythign as a Const.
- None of the following statements are permitted - If, For, ForEach, While. If is acceptable only in a Ternary expression.
- Where possible write as many functions as small, concise arrow functions.

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 or Project Euler. Something you can get your teeth into.

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 4.0.

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.

---

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

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

3 As a Functional programmer, and a believer in exposing the most abstract interface possible, I literally **never** use ToList(). Only ever ToArray().

4 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

5 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

I'm not sure when the decision was made to make C# a hybrid Object-Orientated/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 uses 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.

```
 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.

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, 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 show you now 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-Orientated worked examples - Bank Accounts. I'm going to create a set of bank account object 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, so they must be very wealthy indeed). They have an overflow

decimal to add their extra money. They need the interest to be calculated based on both balances.

- A Monopoly player's bank account[1]. They get an extra 200 for passing go.

These are my classes:

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

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

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

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

The Object Orientated approach to these rules - or as I think of it the "long-hand" approach, would look like this:

```
public decimal CalculateInterest(StandardBankAccount sba)
{
  // If real type of object is PremiumBankAccount
  if (sba.GetType() == typeof(PremiumBankAccount))
  {
    // cast to correct type so we can access the Bonus interest
    var pba = (PremiumBankAccount)sba;
    if (pba.Balance > 10000)
    {
        return pba.Balance * (pba.InterestRate +
pba.BonusInterestRate);
    }
```

```
     }

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

      // if real type of object is a Monopoly Player's bank account
    if(sba.GetType() == typeof(MonopolyPlayersBankAccount))
    {
      // cast to the correct type so we can get access to the bonus
      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.

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 CalculateInterest(StandardBankAccount sba)
 {
     // If real type of object is PremiumBankAccount
     if (sba is PremiumBankAccount pba)
     {
         if (pba.Balance > 10000)
         {
             return pba.Balance * (pba.InterestRate +
pba.BonusInterestRate);
         }
     }

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

     // if real type of object is a Monopoly Player's bank
account
     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 CalculateInterest(StandardBankAccount sba)
 {
   switch (sba)
   {
      case PremiumBankAccount pba when pba.Balance > 10000:
        return pba.Balance * (pba.InterestRate +
pba.BonusInterestRate);
```

```
        case MillionairesBankAccount mba:
          return (mba.Balance * mba.InterestRate) +
                   (mba.OverflowBalance & mba.InterestRate);
        case MonopolyPlayersBankAccount mba:
          return (mba.Balance * mba.InterestRate) + PassingGoBonus;
        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 syntax that more closely matched JSON. 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.

Also, switch is now a Function itself, so you can create a small, single-purpose function with surprisingly rich functionality, which is simply an arrow function. This means it can also be stored in a Func delegate for potential passing around as a Higher-Order function too.

```
   public decimal CalculateInterest(StandardBankAccount sba) =>
     sba switch
     {
       PremiumBankAccount { Balance: > 10000 } pba => pba.Balance *
                       (pba.InterestRate + pba.BonusInterestRate),
       MillionairesBankAccount mba => (mba.Balance *
  mba.InterestRate) +
                   (mba.OverflowBalance & mba.InterestRate);
       MonopolyPlayersBankAccount mba =>
                  (mba.Balance * mba.InterestRate) +
  PassingGoBonus;
       _ => sba.Balance * sba.InterestRate
     };
   }
```

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.

```
 public decimal CalculateInterest(StandardBankAccount sba) =>
   sba switch
   {
     PremiumBankAccount { Balance: > 10000 and <= 20000 } pba =>
pba.Balance *
                            (pba.InterestRate +
pba.BonusInterestRate),
     PremiumBankAccount { Balance: > 20000 } pba => pba.Balance *
                   (pba.InterestRate + pba.BonusInterestRate *
1.25M),
     MillionairesBankAccount mba => (mba.Balance *
mba.InterestRate) +
                  (mba.OverflowBalance & mba.InterestRate);
     MonopolyPlayersBankAccount {CurrSquare: not "InJail" } mba
=>
                (mba.Balance * mba.InterestRate) +
PassingGoBonus;
     ClosedBankAccount => 0,
     _ => sba.Balance * sba.InterestRate
   };
 }
```

I've added a few more rules into my example with these features. Now there are two categories of PremiumBankAccounts with different levels of special interest rates[2] and another bank account type for a Closed account, which shouldn't generate any interest.

Pattern Matching in C# 10

As of the time of writing, C# 10 is the very latest version available to developers, at least as an actual production-ready version. Some C# 11 features are available in preview.

Like C# 9, there is 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 CalculateInterest(StandardBankAccount sba) =>
   sba switch
   {
     PremiumBankAccount { Balance: > 10000 and <= 20000 } pba =>
pba.Balance *
                              (pba.InterestRate +
pba.BonusInterestRate),
     MillionairesBankAccount mba => (mba.Balance *
mba.InterestRate) +
                   (mba.OverflowBalance & mba.InterestRate);
     MonopolyPlayersBankAccount {CurrSquare: not "InJail" } mba
=>
                  (mba.Balance * mba.InterestRate) +
PassingGoBonus;
     MonopolyPlayersBankAccount {Player.FirstName: "Simon" } mba
=>
                  (mba.Balance * mba.InterestRate) +
(PassingGoBonus / 2);
     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

Microsoft has released previews of C# 11, which is likely to be released towards the end of 2022. As appears to be traditional, we're gaining yet another new piece of Pattern Matching syntax. This might be one of the most interesting additions we've had in a few versions. Microsoft is adding the ability to match on array contents!

This is matching on the actual values contained at the beginning, middle and/or end of the array, then on match returning a value.

The use cases for this are somewhat limited, but it might be nice for parsing data sets from files our device outputs with standard headers or footers, which determines how the data should be processed.

What else could be ahead?

In many ways, when it comes to Pattern Matching, C# is playing catch-up with F#, which has a far richer set of Pattern Matching features, which have been present in the language since the beginning - rather than being late additions, as with C#.

List matching was a major feature we lacked, but with the upcoming release of C#11, that will be available to us at last. So, what else could there be on the horizon for future C# releases?

Discriminated Unions

I'm not sure whether this is something we'll ever get in C# or not. I've been sent at least one attempt to implement this in Nuget[3] but I've not heard of Microsoft planning to include it somewhere in C#12+.

What are Discriminated Unions? Here are a few examples from F#[4]

Descriminated unions are a way of quickly and easily throwing together a class (or Type) which can actually be one of any number of child types. This is an example:

```
type Customer =
    | Registered of Name:string * Email:string * IsEligible:bool
```

```
      | Guest of Email:string
```

In this example the Customer type is something like an Abstract class with two implementations: Registered and Guest. This represents a Customer to your e-commerce website, who may be registered with their own account, or may not be using an account & buying something unregistered. In C#, I've often seen code along these lines:

```
public class Customer
{
   public string Name { get; set; }
   public string Email { get; set; }
   public bool IsEligible { get; set; }
   public bool IsRegistered { get; set; }
}
```

This single class can contain the details of both kinds of customer: Registered and Guest, with a boolean used to give the recipient of the class a clue as to which it is. Any consuming function would need to know to check the IsEligible property to know whether the Name and IsEligible properties are likely to have a usable value.

This is probably breaking something like the Interface Segregation Principle - forcing the class to include properties that might not be something it makes sense to exist for non-registered customers.

A more reasonable implementation, following good Object-orientated design might look something like this:

```
public class Customer
{
   public string Email { get; set; }
}

public class RegisteredCustmer : Customer
{
   public string Name { get; set; }
   public bool IsEligible { get; set; }
   public bool IsRegistered { get; set; }
}
```

This is very similar to what the Discriminated Union (DU) in F# is doing, except there is no actual inheritance. The only thing connecting the two classes is simply that they are in the DU. A more accurate depiction of what the F# class is doing is like this:

```
public abstract class Customer {  }

public class Registered : Customer
{
   public string Email { get; set; }
   public string Name { get; set; }
   public bool IsEligible { get; set; }
}

public class Guest : Customer
{
   public string Email { get; set; }
}
```

From an OO perspective, this is slightly wasteful. The base class doesn't have much reason to exist. From a Functional Programming perspective however, we have something with descriptive names that we can feed into a Pattern Matching function, and handle all possible scenarios appropriately.

We can implement DUs in principle, following the pattern above with an abstract base class, but that's a lot of boilerplate code we'd be adding in. Strictly speaking, the F# version uses something in principle more like a Tuple. DUs in F# can be something thrown together quickly and disposed of just as quickly again. That can't be done in a strongly typed language with a strong OO background like C#.

Only the future will tell whether this feature is ever introduced or not.

Active Patterns

Active Patterns is an F# feature I can see being added to C# sooner or later. It's an enhancement to a 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 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#.

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.

Immutability

As with Pattern Matching, Microsoft has spent a lot of time over many versions of C# to add in more features to help enforce Immutability. For reasons of backwards compatibility - an important design choice for the C# team - it's highly unlikely that C# will ever have true immutability-by-default, as F# does, but more tools are being made available to developers to move things in that direction.

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# 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 Funcional Programmer. We create an object, 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 and 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 readonly string Title;
    public readonly string Directory;
    public readonly IEnumerable<string> Cast;


    public Movie(string title, string directory,
IEnumerable<string> cast)
    {
        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, IList<string>
cast)
    {
        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**. An init property is one that can have its value set when the object it's party 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, along with the properties, 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 was passed by reference - being based on a class, but as of C# 10 and onwards, were more their own thing and could be passed by value. Unlike a struct however, there is no readonly modifier, so immutability has to be enforced using init-only properties 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[5]. 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's Cut",
      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's Cut"
};
```

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.

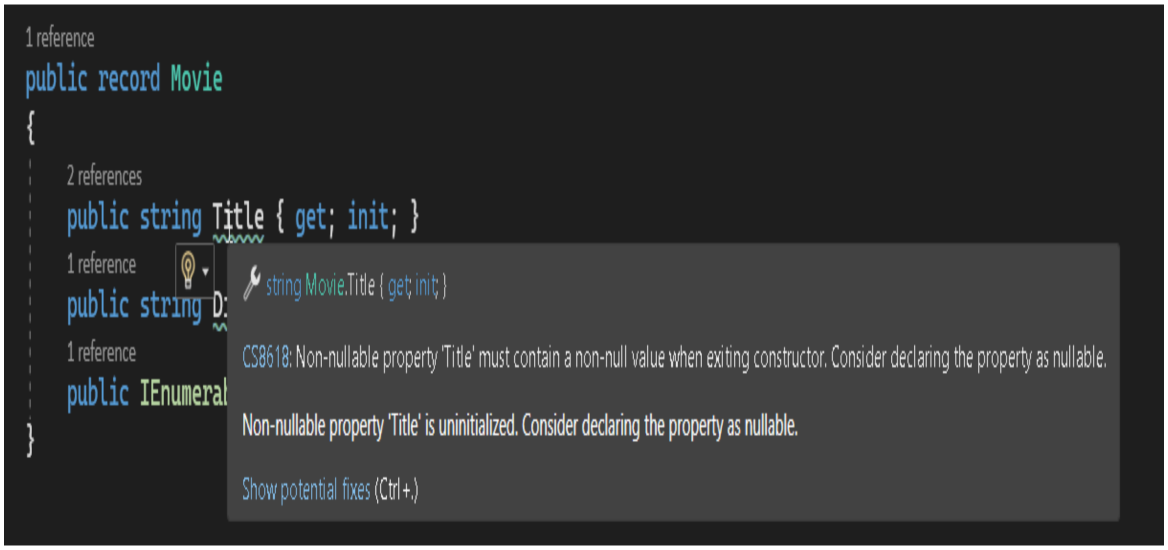
Nullable Reference Types

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 compiler, but it does add an extra set of warnings to the IDE 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 IEnumeral
}
```

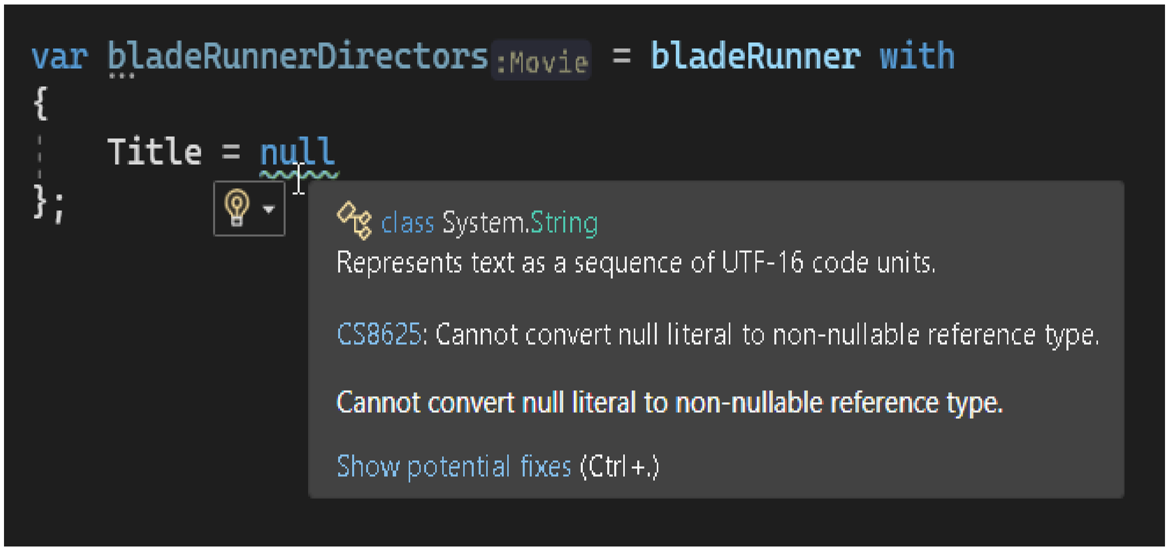
string Movie.Title { get; init; }

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.

If there's a perfectly good reason for a value to be NULL[6] 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; }
}
```

But let me stress again, I don't like having to do this. There are cases where I have no choice, because I'm having to consume a 3rd party data source of some kind that requires nullable properties. That's the only circumstance under which I'd ever consider deliberately adding a nullable property to my codebase. Even then, I wouldn't allow the Nullable to be persisted through the rest of my code - I'd probably tuck it away somewhere 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, most of the talk of C#'s future revolves around the upcoming release of C# 11 towards the end of 2022. There hasn't been much talk yet of C# 12, and what it might add, but 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.

---

1 I don't know about you, but I always preferred to play as the Racing Car, No idea why, I don't even like cars!

2 Which, frankly no bank would ever offer

3 Kim Hugener-Olsen's Sundew.DiscriminatedUnions: *https://github.com/sundews/Sundew.DiscriminatedUnions*

4 Thanks to F# Guru Ian Russell for these examples.

5 Vastly superior to the theatrical cut, in my opinion.

6 I don't personally ever recommend doing this. There are better ways than using NULL in Functional Programming

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.