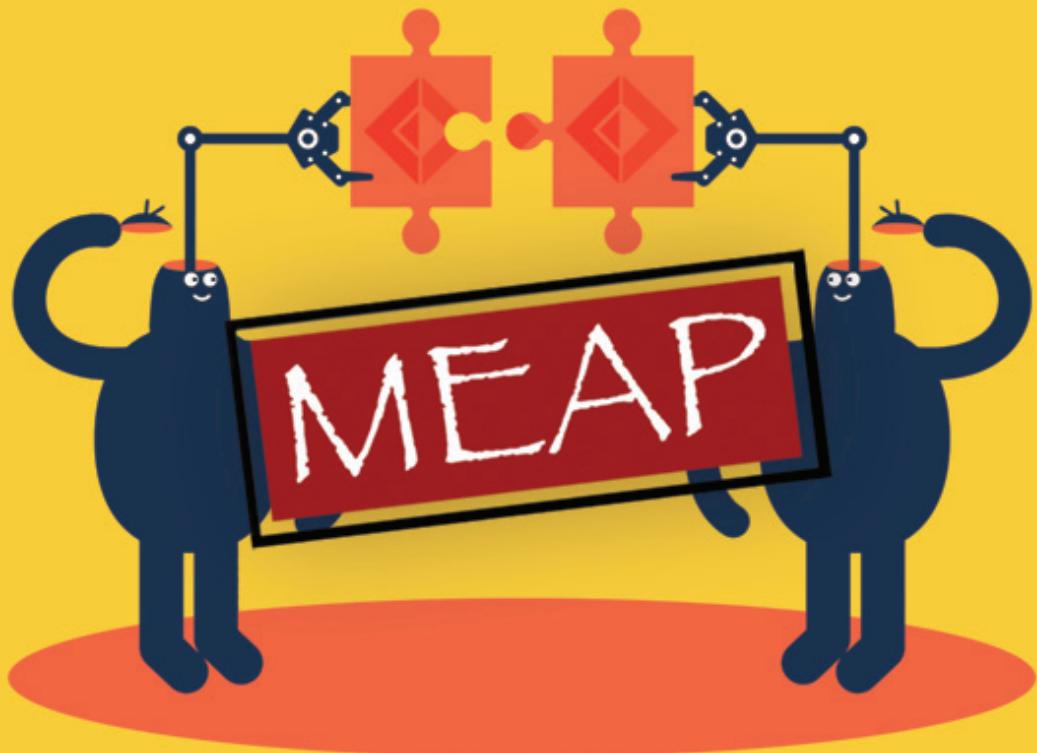


GET PROGRAMMING WITH **F#**

A guide for .NET developers



Isaac Abraham



MEAP Edition
Manning Early Access Program
Get Programming with F#
A guide for .NET developers
Version 8

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

welcome

Hi, and thanks for purchasing the MEAP version of *Get Programming with F#: A guide for .NET developers!* I'm really excited to have reached a stage where we can open the book up to a wider audience – I'm hoping that over the course of the coming weeks as I work to a conclusion on *Get Programming with F#*, you'll be able to provide some honest feedback that will help shape this book to be the best it can be. F# unfortunately has a reputation as a “niche” language designed solely for maths and science, and I'm hoping that you'll see throughout this book that that's not the case – but I'll need your feedback and help to ensure that goal is realised.

The real goal of *Get Programming with F#* isn't to make you an expert in the F# or functional programming. It's an intermediate book aimed squarely at C# and Visual Basic .NET developers – who have heard about F# and functional programming, want to see what the benefits are, and – importantly – how they can use it as a part of their existing toolbox (without needing to throw away all their existing code). Hopefully, by the end of the book, you'll have seen enough to start using it in your day-to-day, as well as understand how and where to further your knowledge of the language. To that end, this book won't waste time teaching you about Visual Studio, or the .NET framework – instead, it'll focus on showing you what I consider to be a core subset of the F# language that can be readily applied in common scenarios with practical examples and tips, as well as popular libraries and frameworks within the F# world.

The book is more or less divided into two distinct sections – the first half (which is almost complete) covers the F# language and development “process” that F# developers tend to follow. We won't cover hard-core functional programming theory (so you won't see any explanation of monads – sorry!), but we'll see what I consider to be the core set of FP techniques in F#, and why you'll want to use them. You'll spend most of your time learning how to write idiomatic F# code on the .NET framework in Visual Studio, and what tools to use to give you the best experience within VS. Then, the second half of the book will show you how to apply F# in larger, real world scenarios – so we'll cover things such as interop to C# / VB projects, SQL database access, web programming and unit testing. Again, many of these lessons will assume knowledge of frameworks and tools such as NuGet and ASP .NET etc., so that we can focus on understanding how F# fits within the context of those tools.

Note that the source code presented in the lessons (without the extra characters used to mark code annotations) is available at <https://github.com/isaacabraham/learnfsharp>.

I'm really hoping that with your help we can make *Get Programming with F#* a great way to introduce the language to C# and VB .NET developers – I look forward to working with you!

Cheers

—Isaac

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/get-programming-with-f-sharp>

Licensed to Gang Li <lg.careercross@gmail.com>

brief contents

Lesson 0 Welcome to Get Programming with F#!

UNIT 1: F# AND VISUAL STUDIO

- Lesson 1 The Visual Studio tooling experience*
- Lesson 2 Creating your first F# program*
- Lesson 3 The REPL - changing the way we develop*

UNIT 2: HELLO F#

- Lesson 4 Saying a little, doing a lot*
- Lesson 5 Trusting the Compiler*
- Lesson 6 Working with immutable data*
- Lesson 7 Expressions and Statements*
- Lesson 8 Capstone 1*

UNIT 3 TYPES AND FUNCTIONS

- Lesson 9 Shaping data with Tuples*
- Lesson 10 Shaping data with Records*
- Lesson 11 Building composable functions*
- Lesson 12 Organising functions without classes*
- Lesson 13 Achieving code reuse in F#*
- Lesson 14 Capstone 2*

UNIT 4: COLLECTIONS IN F#

- Lesson 15 Working with collections in F#*
- Lesson 16 Useful collection functions*
- Lesson 17 Maps, Dictionaries and Sets*
- Lesson 18 Folding our way to success*
- Lesson 19 Capstone 3*

UNIT 5: THE PIT OF SUCCESS WITH THE F# TYPE SYSTEM

- Lesson 20 Program Flow in F#*
- Lesson 21 Modelling relationships in F#*
- Lesson 22 Fixing the billion-dollar mistake*
- Lesson 23 Business rules as code*
- Lesson 24 Capstone 4*

UNIT 6:	LIVING ON THE .NET PLATFORM
<i>Lesson 25</i>	<i>Consuming C# from F#</i>
<i>Lesson 26</i>	<i>Working with Nuget Packages</i>
<i>Lesson 27</i>	<i>Exposing F# Types and Functions to C#</i>
<i>Lesson 28</i>	<i>Architecting hybrid language applications</i>
<hr/>	
<i>Lesson 29</i>	<i>Capstone 5</i>
UNIT 7:	WORKING WITH DATA
<i>Lesson 30</i>	<i>Introducing Type Providers?</i>
<i>Lesson 31</i>	<i>Building Schemas from Live Data</i>
<i>Lesson 32</i>	<i>Working with SQL</i>
<i>Lesson 33</i>	<i>Creating Type Provider-backed APIs</i>
<i>Lesson 34</i>	<i>Using Type Providers in the real world</i>
<hr/>	
<i>Lesson 35</i>	<i>Capstone 6</i>
UNIT 8:	WEB PROGRAMMING
<i>Lesson 36</i>	<i>Asynchronous Workflows</i>
<i>Lesson 37</i>	<i>Exposing data over HTTP with Web API</i>
<i>Lesson 38</i>	<i>Consuming HTTP data</i>
<hr/>	
<i>Lesson 39</i>	<i>Capstone 7</i>
UNIT 9:	UNIT TESTING
<i>Lesson 40</i>	<i>Unit testing in F#</i>
<i>Lesson 41</i>	<i>Property Based Testing in F#</i>
<i>Lesson 42</i>	<i>Web Testing</i>
<hr/>	
<i>Lesson 43</i>	<i>Capstone 8</i>
UNIT 10:	WHERE NEXT?
<i>Appendix A</i>	<i>The F# Community</i>
<i>Appendix B</i>	<i>F# in my organisation</i>
<i>Appendix C</i>	<i>Must-visit F# resources</i>
<i>Appendix D</i>	<i>Must-have F# libraries</i>
<i>Appendix E</i>	<i>Other F# language features</i>

0

Welcome to Get Programming with F#!

Welcome to F#! I'm hoping that you're reading this book because you've heard something interesting about F# and want to learn more about how you can start using it within your daily work cycle. Perhaps you've heard something about its data processing capabilities, or have heard that it can lead to systems that have less bugs, or that it can lead to more rapid development cycles than C# or VB .NET. These are *all* true, but there's a whole lot more to F# than just that. F# opens a whole host of possibilities to you as a .NET developer that will open your eyes to a better way to develop software – one that leads to you enjoying what you do more, whilst making you more productive in your day-to-day job.

In this introduction, we'll summarise at a high level what F# is (and isn't!), and then discuss some of the benefits that you'll receive from using it. We won't spend much time looking at the language in this chapter, so you'll have to take some of what I say at face value, but you will end up with a good idea of where F# can potentially fit in with your day-to-day role. We'll also cover the relationship of F# within the wider context of the .NET ecosystem, as well as taking a look at the F# community and how F# fits into the open source world. At the end of this introductory chapter you'll have a good understanding of the "what" and "why" of F# - the rest of the book will then explain the "how"!

Whilst I do think it's important to read this, if you're really keen to just "dig in", feel free to go straight to Lesson 1 and start bashing out some code!

0.1 What is F# - and why does it matter?

Let's first start at the very beginning and discuss at a relatively high level some of the key points of what F# really is (and isn't!), and then discuss the key question that you probably have on the tip of your tongue – why should you spend time learning a new language with a new way of doing things, when you could be productively bashing out code right now in C# or VB?

0.1.1 What is F#?

So, what's this whole buzz about F#? F# is a language that's rapidly growing in popularity, and attracting developers to be used in a variety of domains, not just from within the .NET ecosystem but also from without.

Let's start by quoting directly from the fsharp.org website:

F# is a mature, open source, cross-platform, functional-first programming language. It empowers users and organizations to tackle complex computing problems with simple, maintainable and robust code.

Source: <http://fsharp.org/>

Let's discuss a few points mentioned in that quote: -

- **Mature** – Firstly, F# is a mature language with an established community. Based on the ML family of programming languages, and now at version 4, it's been a first-class citizen of Visual Studio since Visual Studio 2010 (though you may not even have been aware of it) and runs on the .NET framework, which itself is over 15 years old now. So, in terms of the “risk” of F#, you don't have to be concerned about things such as sourcing package dependencies or that you won't be able to find resources or help online.
- **Open Source** – F# as a project is entirely open source, and has been for several years now (way before Microsoft's recent public shift towards open source). The compiler is open source, anyone can submit changes to the compiler and language (subject to approval of course!), and it runs not only on Windows but also on Mono, with support for Microsoft's new .NET Core well on the way at the time of writing.
- **Functional-first** – F# is prescriptive in the sense that the language encourages us – *but does not force us* – as developers to write code in a functional programming (FP) style. We'll come on to what those features are in more detail shortly, and how they affect how we code, but for now just know that it's an alternative way to modelling and solving problems compared to the object oriented (OO) paradigm that you're used to. However, as F# runs on the .NET framework (an inherently OO framework) F# needs to support OO features. So, you can also consume *and create* classes and interfaces etc. just as you can with C# – it's simply that it's not the “idiomatic” way to write F#. In this book, we'll be effectively ignoring the OO side of F# – which would mostly just be about learning a new syntax – and focusing on the FP side, which is much more interesting.

Another point that the quote above alludes to is that F# helps us focus on delivering business logic and ultimately business value, instead of having to focus on complex design patterns, class hierarchies and so on in order to achieve quality. This is partly due to the fact that F# contains a powerful compiler that does a lot of the heavy lifting for us, which can lead to more succinct solutions than you might normally try in C#. However, in addition, the FP paradigm (in conjunction with F#) emphasises composing small pieces of functionality together to naturally

build more powerful abstractions rather than through designing large class hierarchies up front and building downwards.

WHAT IS FUNCTIONAL PROGRAMMING?

In order to explain this a little more, let's quickly define first what I'm considering FP to be within the context of this book - just like OO design, it can be a subjective matter to define what FP exactly is, and many languages that support FP to some extent will have some features that another one doesn't. At the end of the day, you can think of FP as a "sliding scale" of language features that encourage a particular style of programming; I would suggest that there are a few core fundamentals which any FP language should have good support for:

- **Immutability.** The ability to create values that can never be changed in their lifetime, which leads to a clear separation of *data* and *functionality* (unlike the OO world which merges both state and behavior into classes).
- **Expressions.** The notion that every operation in your program has a tangible output that can be reasoned about.
- **Functions as values.** The ability to easily create, use and share functions as a unit of abstraction and composition within a system.

There's certainly more to FP than just that, and many languages have features such as sum types, pattern matching, type classes and the often-dreaded "monad"; we'll deal with *some* of these features throughout this book, but the above three are the key ones that we'll really emphasise. Just remember that FP is just another way to model solutions to programming problems – there's no real magic involved, and nor do you need to be a mathematical genius.

0.1.2 F# alongside other programming languages

This diagram illustrates where I would consider F# to "fit in" within the spectrum of other mainstream programming languages.

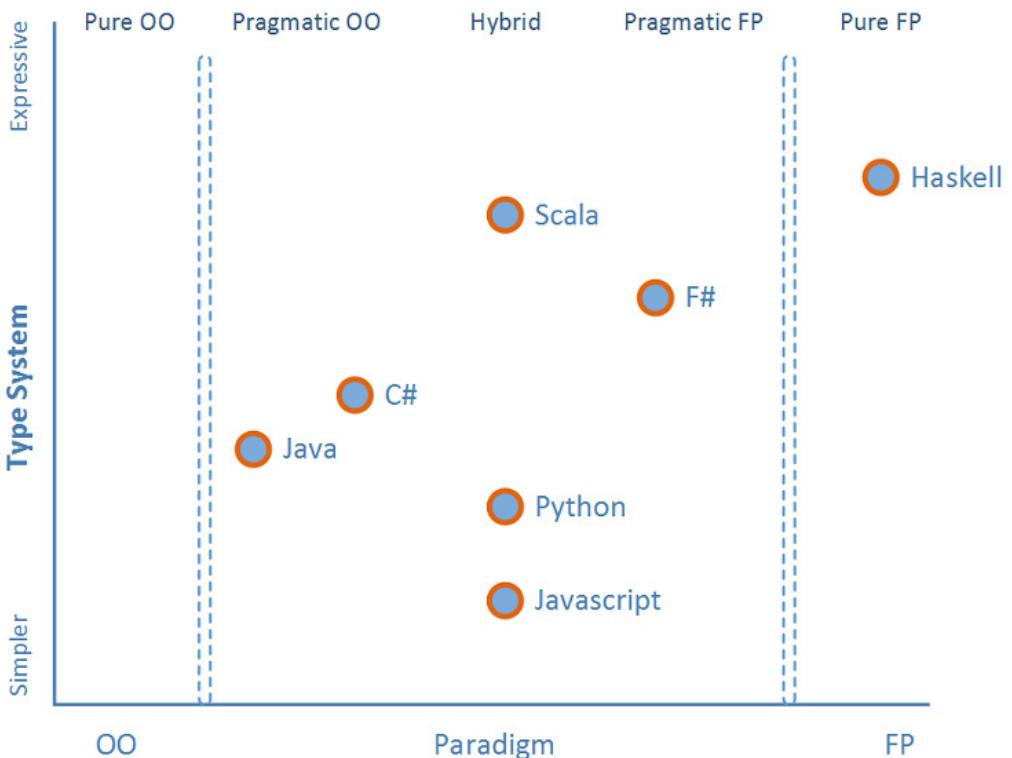


Figure 0.1 F# within the context of other programming languages

F# AND C# - TWO SIDES OF THE SAME COIN

Let's start by looking at F# which, as already stated, is a functional *first* language. It's clearly on the "FP" side of the axis, yet it's not on the far right. Why is this? Well, F# makes it easy to write code in the FP style, but is actually a *hybrid* language, supporting both OO and FP features -it's not a "pure" functional language. So, it allows us to write code that might not be allowed in a "stricter" FP language such as Haskell.

Conversely, if we relate this diagram with two languages that we know well – C# and VB.NET - I would suggest that both are currently going through something of a change of identity. What should we consider "idiomatic" C# code to be? On the one hand, it started life as an OO language with extremely limited support for FP. More recent iterations of the language have introduced *some*(limited) FP features - and the next version of C# yet more FP features - so perhaps it'll move further across this axis. Yet it has its roots in the OO paradigm, and the design of the language makes typical OO features e.g. mutability, statements and classes deliberately easy to utilise.

C# and VB .NET

I'll take this opportunity to acknowledge that the diagram mentions C# only; as far as I'm concerned, aside from a few corner cases and syntactical features, C# and VB .NET are the same language both in terms of features and their ideology towards problem solving and modelling (although interestingly, Microsoft in early 2017 stated that in the future VB won't follow C# just to ensure parity). So, if later in this book, I only mention C# or VB .NET for the sake of brevity, just do a find-and-replace operation in your head to show both languages.

In some ways, we can think about F# as being the alternative side of the same coin from a C# perspective – often, things that are difficult to achieve in C# are easy to implement in F# - and vice versa. Let's take a quick look at one example of that – creating mutable variables and immutable values in C# and F#.

Listing 0.1 Declaring values and variables in C# and F#

```
var name = "Isaac";    ①
const string name = "Isaac"; ②
let mutable name = "Isaac"   ③
let name = "Isaac"        ④
```

- ① Declaring a mutable variable in C#
- ② Declaring an immutable value in C#
- ③ Declaring a mutable variable in F#
- ④ Declaring an immutable value in F#

In C#, we normally use mutable variables by default to store data; to make them *immutable*, we need to use extra keywords (`readonly`, `const` etc.). Conversely in F#, to make *mutable* variables, we need to specify an extra `mutable` keyword, otherwise the language assumes you're working with immutable data. This mentality applies across all of F# - it's prescriptive about making it easy to work with certain language features that fit in with the functional programming style, whilst allowing us to revert to modelling problems with objects and imperative code if we're willing to "pay the cost" of a few extra keywords. Here are some distinctions between the "default" ways of working across the three .NET languages; all of the features on the right-hand side can be considered conducive to writing code in the FP style.

Table 0.1 Comparing and contrasting OO and FP based languages on .NET

Feature	C# / VB .NET	F#
Execution Model	Statements	Expressions
Data Structures	Mutable data	Immutable data
Program Flow	Imperative	Declarative
Modelling Behaviour	Stateful Classes	Functions with separate immutable state
Reuse	Inheritance	Composition

You might now say "You *can* implement e.g. stateless functions or expressions in C#", and that's true. However, this isn't the *default* way of working in C# - it requires extra keywords, or extra hoops that need jumping through in order to achieve the required behaviour. In F#, the opposite is true for all of these features. Expressions are easy to achieve; statements are not. Composition is easy to achieve; inheritance feels somewhat unnatural etc.

Is FP "better" than OO?

Or is F# better than C# for that matter? I deliberately avoid saying either of these things when I give talks on F#. Not only can it be perceived as a controversial statement, but it also depends on your *perspective*. If you prefer solving problems using statements, using mutable data and classes with inheritance, then you'll probably find C# a more elegant language, because that's how it was designed to be used! But if you're trying to use features from the right-hand side of the table in your day-to-day C# or VB .NET, F# will quickly feel like a natural fit for you. I find that I can solve problems much more easily using FP features, and as such, F# is a much better fit for me.

EXPRESSIVENESS

In the previous figure, we can also see a second dimension - that of the "expressiveness" of a language. F# allows you to define your intent extremely succinctly, without a large amount of verbosity in terms of syntax. Instead, you concentrate on encoding your business rules in F# within the confines of a few simple rules – the compiler will get on with doing the heavy lifting for you. Note that this does *not* mean that writing F# leads to unreadable code! Coming from a C# or VB .NET background, where we're used to features such as curly braces, explicit typing and statements everywhere, F# *can* initially appear unusual - even a little daunting. However, it's more accurate to think of it as succinct and expressive with a powerful "vocabulary" that requires a little bit of effort to learn. Once you're familiar with the syntax, the F# code is actually extremely easy to understand, and you can rapidly express complex business logic in a succinct manner.

0.1.3 Why F#?

So we've described at a fairly high level what F# is – a functional-first general purpose programming language that runs on the .NET platform. The question now remains, why should you, as an experienced .NET developer who is productive in C# or VB .NET and confident at modelling problems using OO design paradigms, need to look at an alternative way of building solutions?

NEW POSSIBILITIES

You might be looking at F# because you're interested in using it as a way to start exploring new concepts and domains such as type providers and functional programming. This is true. You'll find that F# opens the door to solving entirely new classes of problems such as data analysis, machine learning and DSLs as varied as web testing and build management – some

of which you might never have thought achievable within .NET. Whilst this book won't deal in depth with all of these areas, we'll definitely cover a few of them, and point you in the direction of some resources that you can further explore in your own time.

HIGH QUALITY SOLUTIONS

The F# language syntax and type system is designed for writing software that exhibits some of the key attributes of high quality software. You can think of some of these attributes as: -

- **Readability** – how easy is it to read and *reason about* some code
- **Maintainability** – how easy it is to modify an existing piece of code
- **Correctness** – how simple it is to write code that works as *intended*

In all of these areas, the design of F# leads you down a path which will naturally guide you towards code that exhibits these characteristics. Firstly, it has a syntax which, whilst initially appearing a little unusual when coming from C-style languages, is much easier to read and understand the intent of without performing a "compilation" in your head as you read the code. Let's discuss one of the points that I made earlier – that of imperative vs declarative code. Imperative code can be thought of as the low-level "how" you want to implement something. Conversely, declarative code concentrates more on expressing the "what" you want to achieve and leaving the low-level details to another party. Here's an example of an imperative way of filtering out odd numbers from a list using both imperative C# and declarative F#: -

Listing 0.2 Imperative and Declarative code samples in C# and F#

```
IEnumerable<int> GetEvenNumbers(IEnumerable<int> numbers) {
    var output = new List<int>(); 1
    foreach (var number in numbers) { 2
        if (number % 2 == 0) 3
            output.Add(number); 4
    }
    return output; }

let getEvenNumbers = Seq.filter(fun number -> number % 2 = 0) 5
```

- 1 : Temporary collection to store output
- 2 : Manual iteration through collection
- 3 : Actual filter logic
- 4 : Manual addition to output collection
- 5 Focus on business logic.

In the declarative version, we're more interested in our *intent* than the implementation details. We don't need to know the details of how the `Seq.filter` function is actually implemented because we understand the general logic of how it *behaves*. Armed with this, we can simply focus on our core goal, which is to identify numbers that are even. The code is much smaller – there's less to read, less to think about, and less that can go wrong. Of course, you can achieve similar code in C# via LINQ; in fact, if you typically use LINQ for these sorts of

operations today, that's great! In F#, this is the *default* way of working - so you'll find that this style of code will be much more common throughout your applications.

Again, I don't want you to worry too much at this stage above the nitty gritty of the F# syntax (or to wonder why there are apparently no arguments in the function!) – I'll explain all of that in more detail later. However, I *do* want you to think about the lack of type annotations and the lightweight syntax with the minimum of boilerplate – this sort of minimalistic approach is a common theme when working with F#.

Where are the types?

Just like C# and VB .NET, F# is a statically typed language. However, it has an extremely powerful type inference engine which means that it has the succinctness close to dynamic languages such as Python, but with the backing of a strong type system and compiler – so you get the best of both worlds. Again, more on this later.

F# also emphasises *composition* rather than *inheritance* to achieve reuse, with support for this baked into the language. Functions are the main component of reuse, rather than classes – so expect to write small functions that "plug together" to create powerful abstractions. Individual functions are nearly always easier to understand and reason about than entire sets of methods coupled together through state in a class, just like writing stateless functions are much easier to reason about than those that are stateful. As such, making changes to existing code is much easier to do with a greater degree of confidence. Here's a simple example of how we might build behaviours into more powerful ones in C# and F#. Again, in C# I've deliberately kept things at the method level, but also think about the effort required when trying to compose more complex sorts of behaviours across classes and objects.

Listing 0.3 Composing behaviours in C# and F#

```
IEnumerable<int> SquareNumbers(IEnumerable<int> numbers) {
    // implementation of square elided...
}

IEnumerable<int> GetEvenNumbersThenSquare(IEnumerable<int> numbers) {
    return SquareNumbers(GetEvenNumbers(numbers))           ①
}
let squareNumbers = Seq.map(fun x -> x * x)
let getEvenNumbersThenSquare = getEvenNumbers >> squareNumbers ②
```

- ① Manually composing the logic of two functions together
- ② Composition as a first-class language feature

Without any understanding of how the `>>` symbol works, it should be clear to you that this operator "fuses" two functions together into one.

Lastly, the F# type system is extremely powerful compared to C# and VB .NET. It allows us to write code in such a way that we can encode many more rules directly into our application so that the compiler verifies our code is valid without the need to resort to e.g. unit

tests. But F# does this in an extremely succinct way, so that we aren't discouraged from being explicit about these rules. To be honest, the majority of the features in F# *can* be achieved in C# and VB .NET, but the cost of doing them would be so high in terms of the amount of code you would need to write, we simply don't do it. In F#, the lightweight syntax is a game changer because it means we can encode more rules into our code without a massive cost increase in terms of code.

PRODUCTIVITY

The difference with F# and other .NET languages is that the sorts of benefits I've mentioned so far are apparent from the syntax of the language right through to the F# core libraries and packages that you'll use. In effect, you have to write a whole lot *less* code in order to design a solution that is *quicker* to write, *easier* to understand and *cheaper* to change than in C# or VB .NET. It's not uncommon to hear F# developers joke that if their code *compiles*, then there's a good chance it *works!* Whilst this isn't *always* strictly true, and unit tests still have a place in F#, there's also definitely an ability in F# to encode more business rules directly into your program, so that you don't have to e.g. spend time writing and maintaining unit tests. You'll find you spend a bit more time in F# itself defining your application logic than in C#, but a *lot* less time in the debugger.

IMPROVING YOUR C# AND VB .NET

Not only will you start to learn how to write applications that are more maintainable, easier to understand and reason about, but – particularly if the *primary* language in your software career has been C# or VB .NET on Visual Studio – F# will open your eyes and make you a better all-round software developer. You'll gain a better appreciation in C# for features such as lambda expressions, expression-bodied members (a recent addition in C#6), and LINQ, and will realise that the application of these sorts of features can be used for more than just data access layers. You'll also probably start to approach modelling problems differently in C#, with less reliance on inheritance and other OO constructs as the "only" way to solve problems.

0.1.4 Working with a smarter compiler

Imagine you have a system with a simple domain model. You decide to add a new field onto one of your core domain classes, and hit "build" in Visual Studio. In the C# and VB .NET world, this sort of change will compile without a problem, despite the fact that you've not actually used or initialised the new field yet. You could even start to reference the field elsewhere in your application, without the compiler preventing you from accessing it without having ever initialised it. Later on, perhaps you'll run the application and receive a null reference exception somewhere further down the chain. Perhaps you'll be unlucky and not actually test it out because it's a particularly obscure branch of code. Instead, a few months later, your users will eventually hit that part of code – and crash the application.

You might say to me that you could write unit tests to force this issue to the surface sooner – that's true. But another truth is that people often don't write unit tests, particularly for seemingly small changes. It would be nice if the *language* could actually support us here. In the F# world, simply adding a new field will instantly break your code. Wherever you create instances of that type, you'll need to *explicitly* set the value of the new field at initialisation time. Only once you'd fixed all the assignments and usages would you be in a position to deal with *how* that new field is being used. Nulls aren't allowed for F# types either – so you wouldn't be able to set it as such – instead, F# has the notion of "optional" types to cater for both possibilities.

The net result? You spent some more time fixing compiler issues and evaluating possible branches of code, but the benefits are that you don't need to write any unit tests to guarantee consistency of the type, nor is there a risk of getting a null reference exception at runtime – and you won't need to debug it to prove that you won't get one.

You may feel an instinctive negative reaction to what I've just told you –after all, why would you want a compiler to "get in the way" of you writing code and slow you down from running your application? Don't worry - this is a normal reaction, and is part of the learning curve in trusting the compiler to help us write correct code in a much richer way than we're used to, which saves us a *lot* more time further down the road.

0.1.5 What's the catch?

So, whilst I've been talking about the benefits of F# and said that it makes your life easier – we all know that there's no such thing as a free lunch. Surely there's a catch somewhere? The truth is somewhere in between.

On the one hand F# *isn't* as hard as is made out; you might have heard that F# should only be used for finance applications or mathematical modelling. **Don't believe this.** Whilst it's certainly true that F# *is* a good fit for those use cases because of the nature of the language and its feature set, the language is *also* just as suitable for writing line of business applications, domain modelling, web development or back end services that fetch and shape data from a data store such as SQL Server or Mongo DB. It's almost as though OO languages with curly braces are apparently the only form of general purpose programming languages in existence! And since F# runs on .NET, you won't have to waste time learning lots of new libraries to be productive (although F#-specific libraries *do* exist), and as we'll be using Visual Studio 2015, you'll be instantly familiar with the development environment (although, again, there are some differences which we'll cover later).

The biggest challenges you'll face are first learning the syntax of F#, and then, more importantly, *unlearning* the dogma of OO methodology. Things such as for loops, classes and mutable state are so deeply ingrained into our mental model of domain modelling and problem solving that it can be difficult to forget them. F# *does* allow you to work with mutable data, imperative styles and classes – but doing these sorts of things in F# certainly isn't idiomatic and the syntax will feel unnatural compared to C# or VB .NET – in essence, you'll get the worst of both worlds. Instead, you'll be better off starting from a clean slate when you

approach solving problems, and you should rely on following a few simple rules and “behaviours” that we’ll cover throughout the book – if you can do that, your solutions will *naturally* end up in a functional style.

When shouldn't I use F#?

To cut a long story short: For the majority of use cases for .NET development today, in my experience F# will let you get things done quicker than in C# or VB .NET with at least the same level of quality. In fact, I'll right now say to you that I would recommend F# over C# or VB .NET as a general purpose programming language for nearly every use case today, be it business logic, data access, rules engines etc. That's a pretty bold statement to make, but having used C# since it first came out, F# for several years now, and run a company that makes F# one of its main selling points, I think I'm allowed to do so!

The only situations that I don't recommend F# for are ones where custom tooling / support is required that is only available in C# or VB .NET e.g. Razor views in ASP .NET, or where the problem domain requires features that are *inherently* OO-based, or requires imperative and / or mutable code. As you'll see in this book, these cases are actually few and far between.

0.2 F# and .NET

We use .NET every day and, by and large, we all like it. It has a wide set of ready-made classes in the BCL across all sorts of areas such as collections, data access classes, UI frameworks and web access etc., and the CLR has many great features such as a smart garbage collector. We also now have a rich ecosystem of rapidly evolving libraries through NuGet that can be released outside of full .NET framework releases. Why should you have to give all this up to use F#? The answer is that you don't - F# runs on .NET.

This is something that may seem obvious but is often overlooked or misunderstood, so it's worth stressing. *F# runs on .NET*. Virtually all of the types that you use in the BCL with C# are also accessible directly from within the F# language. You can use the NuGet packages and reference any .NET DLL, just like you would from C# or VB .NET. At runtime, F# is compiled into Intermediary Language (IL) and is hosted on the CLR so you get all the normal features of the CLR such as garbage collection. You can reference C# or VB .NET assemblies from F# and vice versa, and F# has interop features to allow seamless interaction across languages, whilst also taking advantage of some F#-specific features to make dealing with .NET libraries nicer. You can consume (and create) classes with inheritance, interfaces, properties, methods etc. etc. Indeed, one of the strengths of F# is that it permits the developer to mix both FP and OO styles where appropriate. In other words, you're not going to have to give up the libraries that you already know - or the knowledge you've learned over the past years regarding the CLR, garbage collection, reflection etc.

Whilst C# will, I suspect, always remain the focal point of Microsoft's investment in programming languages on .NET, this has always been the case – VB .NET, for example, has never been marketed by Microsoft as the “main go-to” language of .NET. Similarly, whilst F#

does not receive quite the same level of investment as C#, it's still an important part of the .NET story and will continue to be in the .NET Core world.

In terms of VS integration, since Visual Studio 2010, F# has been supported out-of-the-box as a first-class citizen of Visual Studio. This has continued up until the latest version at the time of writing, VS 2015 (and will continue with VS2017). You can create all the things you would expect to do with VS and F#, including projects in solutions that might also contain C# or VB .NET projects. You can build assemblies in VS using the same process that you're used to. You can create console applications, class libraries, Windows applications, Web applications etc. – all within Visual Studio. What you *won't* find in F# in Visual Studio 2015 out of the box are things like support for code generation, refactorings, or the “smarter” features in VS that are designed specifically for C# or VB .NET such as code analysis or code metrics. However, as you learn more about F#, you'll see that there are ways both through third-party tooling and the *language itself* that negate the need for the tooling that we have come to rely upon in C# or VB .NET.

Visual F#? F#? What's the difference?

It's worth taking a moment to explain the differences between F# and Visual F#. Visual F# is the *Microsoft-managed version* of F# which is included with Visual Studio. It's been open sourced on GitHub, accepting contributions from the public – but is maintained by Microsoft, who ultimately have the final say on what features and changes are accepted. You can consider this repository to currently be the “core” repository for the F# compiler and language. There's a second repository (also on GitHub) which is based off of Visual F# and generally has a virtually identical codebase known as F# Open Edition. This itself feeds into the cross-platform tooling support for Mono and Xamarin as well as other open source tools and code editors. You can find out more about this relationship on <http://fsharp.github.io/>.

For the purposes of this book, we're dealing exclusively with Visual F#, so whenever you see the word F#, you can consider it to mean Visual F#.

0.2.1 F#'s place within .NET

Let's now visually take a look at where F# fits in within the .NET framework and CLR runtime.

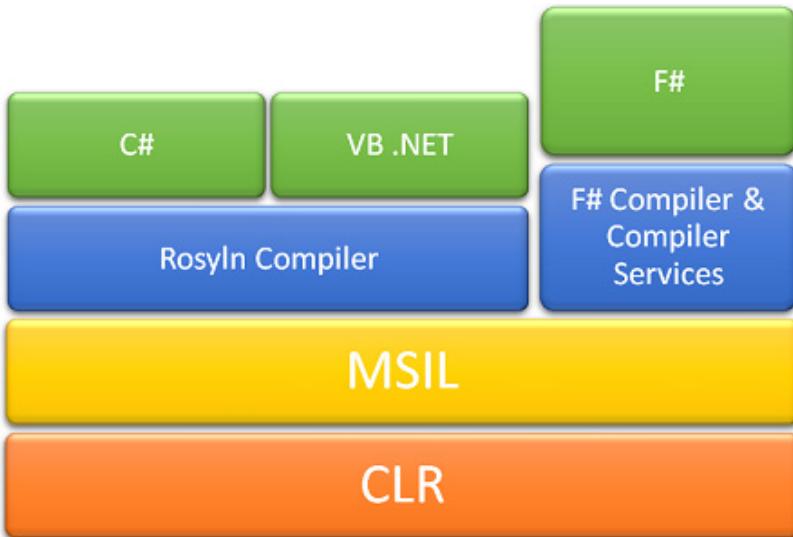


Figure 0.2 F# alongside other popular .NET languages

The F# language sits on top of the Common Language Runtime (CLR), just like C# and VB .NET do. The F# compiler, `fsc.exe`, emits .NET assemblies, just like the C# compiler `csc.exe`(and its replacement Roslyn) does. In addition, the F# Compiler Service (FCS) acts in a similar manner to Roslyn, allowing us to reason about F# code for the purposes of e.g. refactorings etc.

You'll notice that the F# compiler is drawn as a "larger" box than the Roslyn box. That's not to suggest that Roslyn / C# / VB .NET are smaller projects (in fact, quite the contrary)! It's more to do with the fact that if you compare like-for-like idiomatic C# and F#, you'll see that the F# compiler does a *lot* more work for us. It's not unheard of for a single line of F# to emit the same IL as maybe a dozen lines or more of C#. Whilst this might sound a bit of a gimmick, it's actually very important. Once you've been using F#, you'll start to realise how having a smarter compiler allows us to truly focus on the essentials of our problem domain, and to trust the compiler to help us write systems that have a greater "pit of success" – a larger chance of writing the correct code first time.

0.3 Summary

That's the end of the introduction! You should now have an understanding of the "what" F# is and what its benefits are. You learned that F# sits on top of .NET and is included with Visual Studio, so it's going to be easy for us to get up and running. Up until now, you've just been taking my word for all of this – so the remainder of this book will prove it to you, through practical examples.

There's one last piece of advice I'd like to give you before going any further. In order to fully benefit from learning F# in the most effective manner, to quote a wise man: you must *unlearn* what you have *learned*.

Unit 1

F# and Visual Studio

In the first unit of Get Programming with F#, we're going to gently ease in to getting up and running with F#. We'll mostly look at how F# works within the development environment that you're already most familiar with, Visual Studio. In this book, we'll be targeting VS2015.

We'll also get up and running with writing a simple application in F# as quickly as possible, so that you understand which existing skills and knowledge from your C# / VB .NET expertise can be re-applied within the world of F#, and which won't be.

Lastly, we'll explore a different way of developing software in Visual Studio called the REPL.

By the end of this unit, you'll have a good understanding of how F# fits into Visual Studio and what you should (and shouldn't!) expect from the typical F# developer workflow.

Here we go!

1

The Visual Studio Experience

I'm assuming that as a C# or VB .NET developer, you're already familiar with Visual Studio (VS). In this book, we'll be using VS2015 (and F# 4, which it comes with). In this lesson, we'll cover how to ensure that your installation of VS contains everything needed for the remainder of the book so that you're on a level playing field with me, and we can then dive straight into F#! We'll look at:-

- Installing Visual Studio with F#
- Downloading F# extensions for Visual Studio 2015
- Configuring Visual Studio 2015 for use with F#

If you've already used F# before, you might feel the need to skip this lesson – I'd advise you to at least quickly skim through it to ensure that you're working from the same baseline as I am to avoid any confusion later on.

1.1.1 *Installing VS2015 with F#*

Firstly, if you don't have VS2015, don't worry - since VS2015, Microsoft have released a free-to-use version, known as VS2015 Community Edition. This version will be completely usable for the purposes of this book, and if you don't have it, I'd encourage you at this point to download it from <https://www.visualstudio.com>.

When you install VS2015, it's important to not simply select the default options during the installation process. That's because, as of VS2015, Visual Studio has become a much more componentised system, such that from now on, the default options will install a much more "bare bones" version. This means that by default F# is not installed out of the box with VS2015; instead, select Custom and then pick Visual F# from the options.

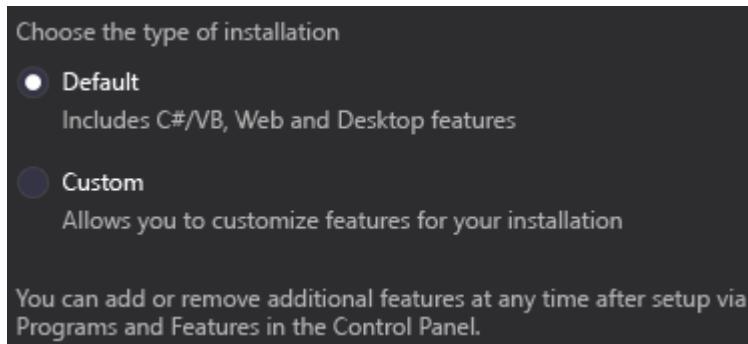


Figure 1.1 Custom options during the Visual Studio 2015 installation process

I've already installed VS2015 without F#

What happens if you've already installed VS2015 and didn't install F# at the time? Don't worry! The first time you use VS2015 to create a new or open an existing F# project e.g. Console Application or Class Library, the installer process will kick in and download the F# features for you automatically. Careful though – the same doesn't apply when creating or opening a *standalone F# file*.

1.2 Configuring Visual Studio for F#

1.2.1 Visual F# Tools configuration

The next thing you should do is to configure the core Visual F# tools as follows: -

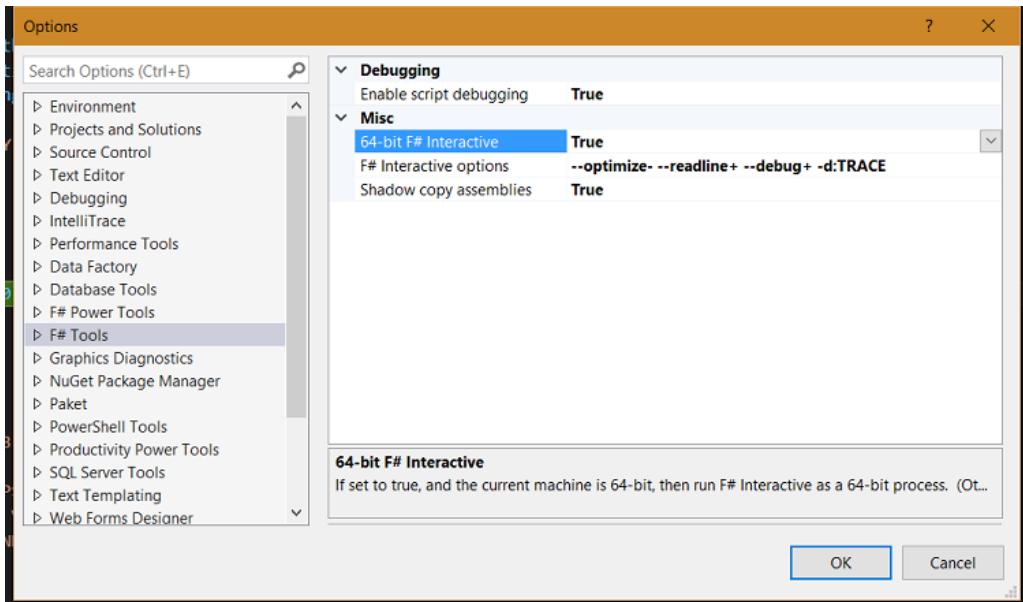


Figure 1.2 Configuring Visual F# Tools

It's not too important at this point to know what these do – and indeed you probably won't ever need to touch these settings again. They'll simply make life easier for you as an F# developer by: -

- Allowing you to reference assemblies within scripts without holding locks on the file, or allowing you to debug F# scripts
- Allowing you to debug F# scripts in Visual Studio
- Allowing you to output trace messages within F# scripts

Other IDEs for .NET

It's worth pointing out that in the last couple of years, a number of other IDEs have cropped up that allow .NET development on Windows without Visual Studio. This include Atom and Microsoft's very own lightweight editor, Code, and both allow you to work with F# through a fantastic external plugin called Ionide. There are many times that I use Code rather than VS as it's extremely lightweight and has great integration with many tools. Nonetheless, I still recommend Visual Studio as the best .NET IDE out there on Windows.

1.2.2 Configuring the F# Editor

Lastly, ensure that you specify in **Text Editor -> F# -> Tabs to Insert spaces**, and not **Keep tabs**. This is very important, as F# uses spaces to denote scope.

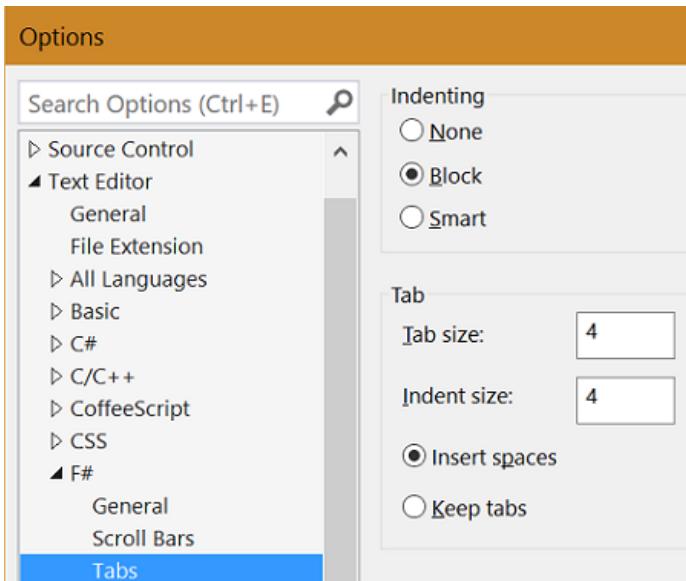


Figure 1.3 Configuring Text Editor options for F#

1.3 Getting the best out of VS 2015 and F#

Almost done! I'd like to finally show you a number of extras that are worth configuring now before we progress further through an extension.

1.3.1 Installing Visual F# Power Tools

Without a doubt, the Visual F# Power Tools (VFPT) is the most important extension to add to VS2015 for F# development. It adds a number of extra features, including: -

- Refactorings
- Code collapsing
- Code generation
- Code formatting
- Syntax colouring
- Code rules

The Power Tools project is completely open source, so you can visit the repository and make changes to the tool if you have new features you'd like to see added. It also actually uses the code from a number of other projects which are available as standalone tools, such as Fantomas (code formatting) and F# Lint (rules). However, just installing the VFPT will bring all of these and more in as one extension, so it's the quickest way to get up and running.

To install, simply go to **Tools -> Extensions and Updates** and search for *fsharp*. Alternatively, you can download the extension from the Visual Studio extensions website (<https://visualstudiogallery.msdn.microsoft.com>).

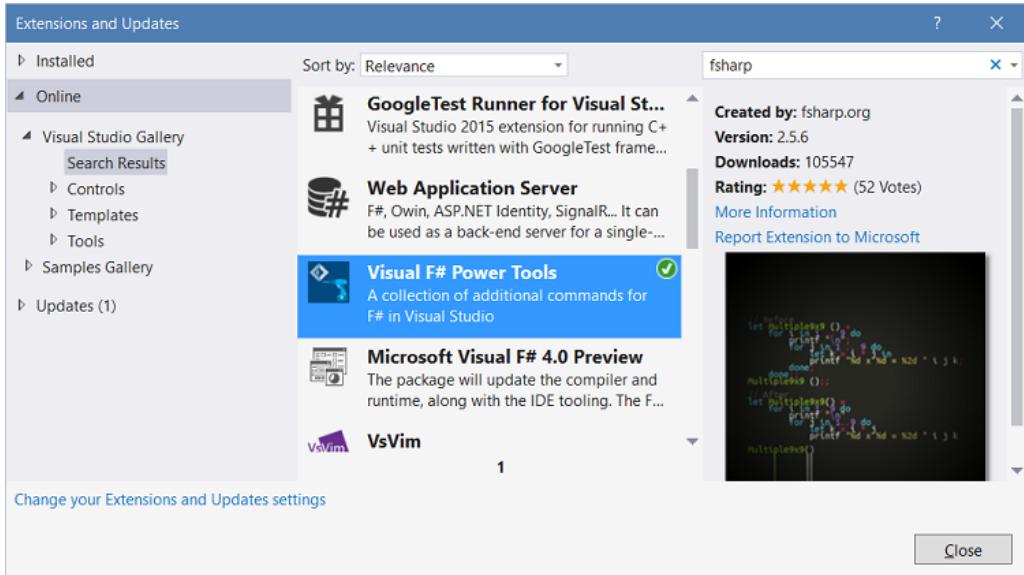


Figure 1.4 Installing F# Power Tools

What about Visual Studio 2017?

VS2017 has now been released! If you're using VS2017, there are some important things to note. Firstly, the F# Power Tools will no longer be available – VFPT is not compatible with VS2017. This is because as part of VS2017, the entire F#/VS integration was rewritten from the ground up to use Visual Studio's Roslyn IDE system (just like Typescript does). There are several benefits of this – adding new refactorings will be much easier, whilst the entire IDE experience will align much more closely with C# and VB .NET.

There's a flip-side to this though: The rewritten integration with VS2017 was somewhat late to the party and suffered from delays, as well as a number of reliability issues – not to mention that (again, at the time of writing) it does not offer feature-parity with VS2015 + VFPT (although I should point out that there's been a huge effort from both the Visual F# team and, even more importantly, several people in the F# community, to improve this in a short space of time).

I'd strongly recommend waiting until the experience is equivalent to VS2015, and the reliability issues have been fixed, before moving across; this might even be an update that is released shortly after RTM. If you are using VS2017, most, if not all of the VFPT features that are mentioned in this book will eventually be ported, and I predict that within 12 months' time, F# will have as strong editor support in VS as C# or VB does today.

Once you've installed and restarted VS2015, go to **Tools > Options > F# Power Tools > General**. You'll see a dialog similar to that below:

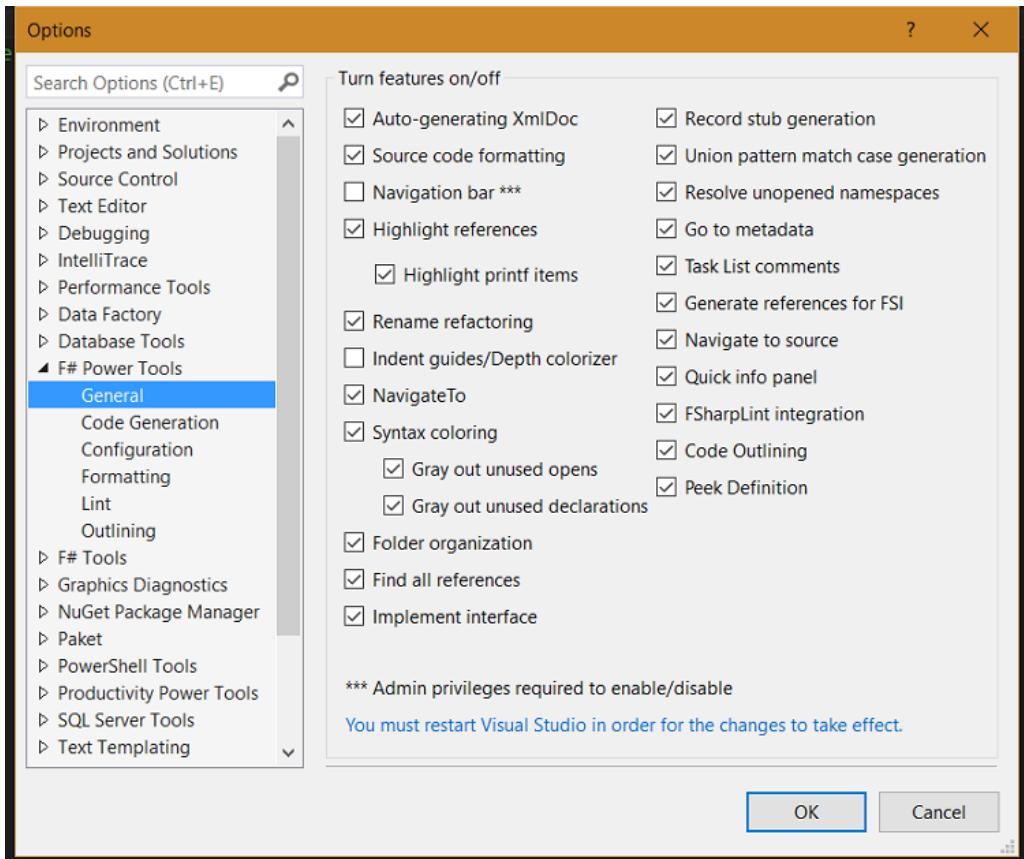


Figure 1.5 The Visual F# Power Tools options page

You should probably activate all of these, as they're all generally useful, with the possible exception of graying out unused opens & declarations as they can be quite CPU intensive. I'd recommend you leave it turned on to start with and see how your machine copes.

You'll also notice the Lint tab, which contains a whole host of "rules" that analyse your code in the background and provide helpful suggestions regarding writing more effective F#. These will be highlighted with orange "squiggly" underlines in code – they are never full compiler errors, but there are "best practice" tips (although like all code analysers, it'll occasionally recommend something that isn't feasible).

1.3.2 Configuring F# Syntax Highlighting

As you'll see later, because of the nature of the F# syntax and its use of type inference, it's important to us to have an understanding of how the compiler "sees" our code to aid us when fixing compile-time errors. Now that we have VFPT installed, just spend a couple of minutes configuring syntax highlighting. To do that, go to **Tools > Options > Environment > Fonts and Colors** and scroll down until you see the F# options.

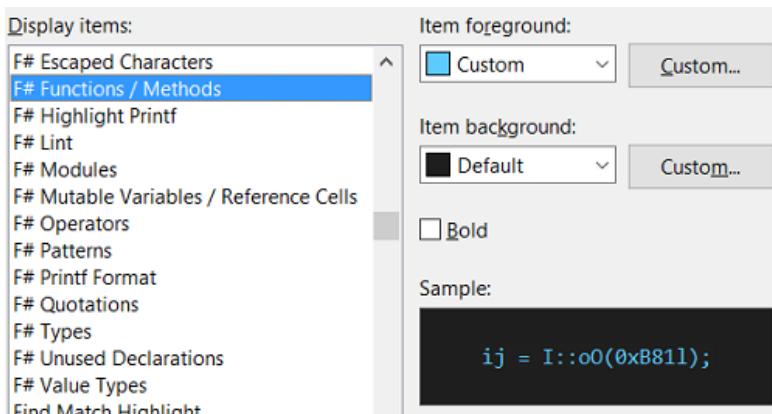


Figure 1.6 F# Syntax Highlighting options

You don't have to configure all of these, but the following ones you will want to change ahead of the defaults. You'll find the following values will set your environment up to distinguish between many of the types of symbols easily: -

Table 1.1 - Custom F# Syntax Highlighting

Dark Theme			Light Theme			
	R	G	B	R	G	B
F# Functions / Methods	094	203	255	086	156	214
F# Modules	255	128	000	000	128	128
F# Mutable Variables	255	128	128	204	000	000
F# Operators	255	128	255	185	000	092
F# Patterns	255	255	128	200	100	000

What about .NET Core and non-Windows?

This book focusses on Windows, Visual Studio and the full .NET on Windows. Nonetheless, F# the language works just the same on Mono as well as .NET Core – so whilst the tool-chain is slightly different (and I won't cover that in this book), the actual language features are the same. When it comes to .NET framework features, things are a little different though, as some technologies such as WPF are Windows-specific and don't exist on .NET Core.

1.4 Summary

In this lesson:

- You installed Visual Studio 2015 and ensured that it was properly configured not only with the in-built tools but also with an extension, Visual F# Power Tools.
- You've also learned that F# is a first-class citizen of Visual Studio, as it is included "in the box" of the installer, or can optionally be applied at a later date if you choose.
- Lastly, you saw that there's very good integration for F# in the sense that the standard VS tooling and editor options are already F# aware. You're now ready to start writing F# applications!

2

Creating your first F# program

Now that you've installed Visual Studio and have the F# tools installed in it, what can we do with it? By the end of this lesson, you'll see how and where F# integrates with Visual Studio. In this lesson: -

- We'll look at creating an F# console application in Visual Studio
- You'll get a brief look at some F# syntax
- You'll also get a feeling for F#'s "less is more" approach.

2.1 F# Project Types

Firstly, we can create projects in a solution, just like we would do in C#, in exactly the same way. F# has support out of the box for a number of project templates, the most important of which are: -

- **Library:** A Visual Studio project that compiles into a .NET assembly ending with ".dll" and can be referenced by other .NET projects and assemblies. You can think of this as equivalent to the **Class Library** project that you'll be familiar with from C#.
- **Console Application:** A Visual Studio project that compiles into a .NET assembly ending with ".exe" capable of being run. This can also be referenced by other .NET projects and assemblies.

Sounds familiar, right?

Now You Try

Let's try to create our first F# project. You'll already know these project types from C# or VB .NET, and they work in essentially the exact same way.

1. Creating one is as simple as hitting **File > New Project** and picking the appropriate project type.

2. If this is the first time you've used F#, you'll have to navigate down to the **Other Languages** node to locate the Visual F# templates.
3. Select **Console Application** and set the name to **MyFirstFSharpApp** before hitting OK.

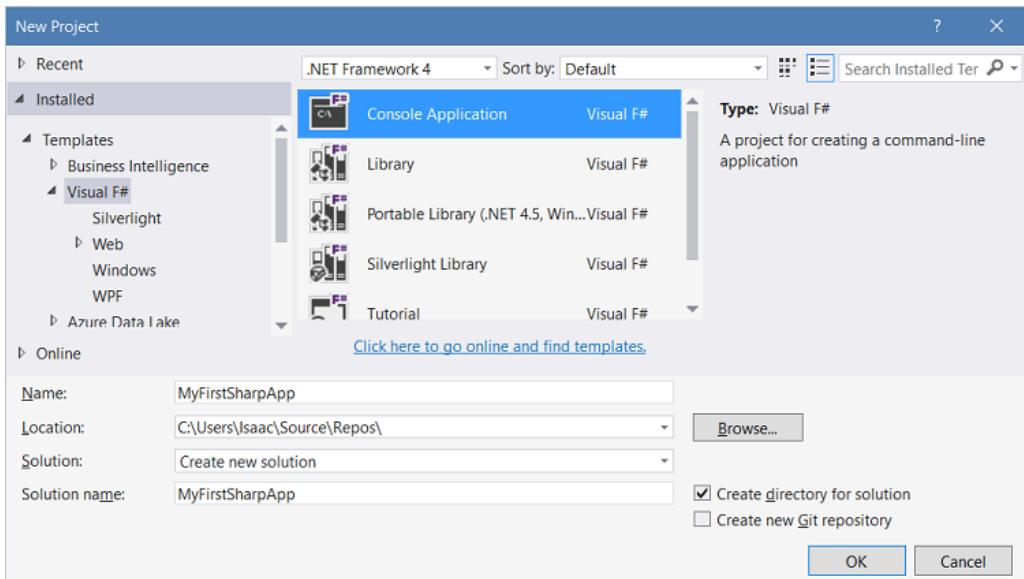


Figure 2.1 - Creating an F# Console Application in VS2015

Where are the projects?

You might notice that in the screenshot above, there's a Web node that you won't have. You might also be wondering where the Windows Forms or WPF project templates are. The answer is that out of the box, VS2015 does not come with any of them. However, there are a number of ways (as we'll see in more detail later in this book) that we can overcome this, such as third party templates or packages.

That's actually one of the reasons why F# works very, very well on other IDEs as well – since it's never been reliant on Visual Studio to provide much at all, its ecosystem has built up in such a way that it can work well across both multiple IDEs and multiple OSes.

The result will be a solution with a single project with a number of files in it.

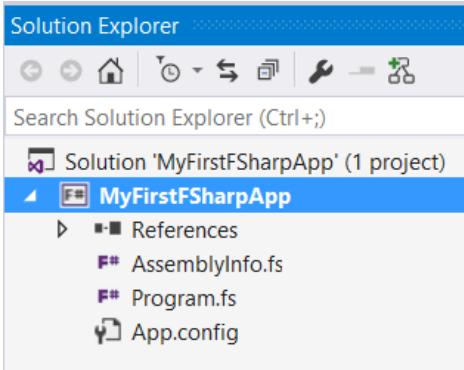


Figure 2.2 - A stock F# Console Application

Let's go through them one by one: -

- `AssemblyInfo.fs` – This file sets properties of the assembly and fulfils essentially the same function as `AssemblyInfo` files in C# or VB projects. It's not particularly interesting, so let's leave it for now.
- `Program.fs` – This is the launch point of your application, and is worth looking into in a little bit more depth.

Listing 2.1 Console Application entry point

```
[<EntryPoint>]
let main argv =
    printfn "%A" argv
    0 // return an integer exit code
```

- ➊ : An attribute which tells F# that this is the function to call when starting the application.
- ➋ : The declaration of the function.
- ➌ : Body of the function - prints out the arguments that were passed in.
- ➍ : Body of the function - returns 0 as the result of the application.

You've probably got a number of questions about this code snippet, and I'll answer the most likely ones by thinking about a comparison with a C# Console Application.

- **No class declaration?** That's right. F# actually does normally require you to create a *module* as a container for functions inside of `.fs` files, but for Console applications you can skip this out entirely - instead, the compiler uses the name of the file as the module implicitly. You can see this by hovering over the main function name, where you'll see the fully qualified name as `Program.main`.
- **Why do I need to use EntryPoint?** In C#, you don't need the entry point because instead you must specify the entry point through the project properties – in F# this is achieved simply by marking the function with an attribute (note that in F#, attributes

are specified as [`<Attribute>`] rather than [Attribute] – the [] syntax is used elsewhere).

- **Where's the return keyword / curly braces / semi colons / type declarations etc. etc.?** Not normally required or valid in F#! You'll find out more in the coming lessons. F# instead is *whitespace significant*, which means that *indentation* of code is used to represent blocks.

F# File Types

Unlike C# and VB .NET, which only have a single file type, F# has two files types:-

`.fs` – equivalent to `.cs` or `.vb`, these are compiled as part of a project by MSBuild and end up in a `.dll` or `.exe`.

`.fsx` – an F# *script* file that is a standalone piece of code that can be executed without first needing to be compiled into a DLL. You'll learn more about script files in the coming lessons, but for now, just remember that these are a lightweight and easy way to explore code without the need for a full-blown console application. In the next version of Visual Studio, C# will have a similar file type to this known as `csx`.

Now You Try

Running an application in F# is the same as normal. Hitting F5 runs with debugger attached, and CTRL + F5 will run without the debugging attached. Running the application without the debugging attached will show you something like the following:

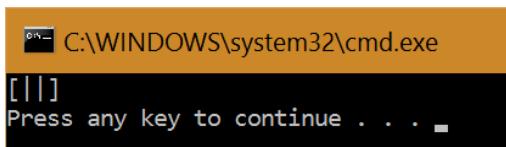


Figure 2.3 - Running the stock F# Console Application.

That's pretty, but what is it? Well, looking back in the application, we can see that we print out the arguments supplied into the app with the call to `printfn`. But we're not supplying any yet, so what is `[]]`? The answer is that F# has *native language* support for some data structures and collections, including standard .NET arrays (we'll learn more about arrays in the Collections unit of the book). The arguments passed into `main (argv)` is an array of strings (or `string []`), just like in C# / VB. In F#, the syntax for arrays is essentially: -

```
let items = [| "item"; "item"; "item"; "item" |]
```

The closest equivalent in C# would probably be: -

```
var items = new [] { "item", "item", "item", "item" };
```

As we're passing in an empty array to our application, that's all that's printed out i.e. [| |]. Let's change that by supplying some arguments to the application.

1. Go to the properties pane of the project by selecting the project node in Solution Explorer and hitting ALT + ENTER.
2. Navigate to the **Debug** tab.
3. In the **Command Line** arguments box, enter the text **HELLO WORLD**.
4. Rerun the application.
5. You'll now see the following: -

```
C:\WINDOWS\system32\cmd.exe
[ "HELLO"; "WORLD" ]
Press any key to continue . . .
```

Figure 2.4 - A command line F# application printing out input arguments.

So, the arguments supplied are automatically converted into the array. That's nice, but let's now actually look at the F# syntax a little more, before writing some F# ourselves and incorporating it into our existing application.

Now You Try

We'll deal with more of the F# syntax in the next couple of lessons, but even exploring this small code snippet raises a number of interesting questions.

1. Mouse over each of the values defined in `Program.fs` to get intellisense over them. You'll see that the values all have explicit types according to the tooltips – yet none are specified in the code. Where do they come from? This is F#'s type inference engine at work – we'll find out more about this in Lesson 5.
2. If you installed the Power Tools extension as per Lesson 1, you'll see that `main` and `printfn` are highlighted in a different colour to `argv`. This is because both `main` and `printfn` are functions. You'll find the colouring of different symbols extremely useful in understanding the context of your code and learning how the F# compiler parses your code.

Quick Check

1. What are the two basic project types for F# shipped with Visual Studio?
2. What is the `[<EntryPoint>]` attribute for?
3. What are the two types of F# files in a project?

2.2 Debugging applications in F#

Debugging applications is less important in F# than in other languages – because of the language design and features like the REPL, you'll usually have tested out most outcomes before you ever run your application. However, there's still a very good debugging experience in Visual Studio for F#, with the usual features such as breakpoints and watches etc. However, there are times when the debugging experience falls a *little* short of what you're probably used to in C#, simply because some of the more advanced features of F# that don't exist in C# aren't supported by the debugger.

Now You Try

Let's explore with the Visual Studio debugger a little.

1. Go to line 6 in `Program.fs` and press F9 (or if you have the breakpoints column turned on in the editor, click there).
2. Debug the application with F5.
3. Mouse over the `argv` value when the breakpoint is hit. Observe that you can drill into the array to see the values passed in.
4. The normal commands such as F10 for step over and F11 for step into all work as normal.

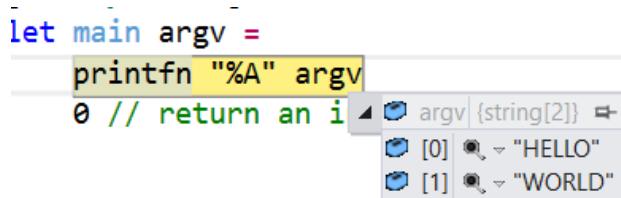


Figure 2.5 - The VS debugger within an F# console application.

Quick Check

4. What keyboard shortcut is used for adding debug breakpoints?

2.3 Writing your first F# program

Let's end this lesson with you writing a little bit of F#. We'll change the output of the `printfn` to show us the length of the array passed in as well as the items themselves. Change the code as follows: -

Listing 2.2 An enhanced Console Application

```
[<EntryPoint>]
let main argv =
    let items = argv.Length
```

```
printfn "Passed in %d items: %A" items argv
0 // return an integer exit code
```

You'll see that when you "dot into" `argv`, you'll get intellisense for the array. Make sure you supply the arguments to `printfn` in the correct order!

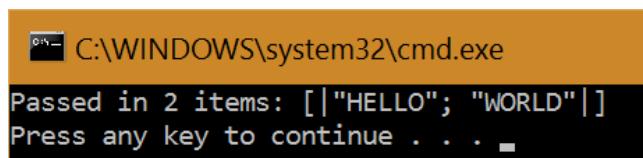
printfn in F#

`printfn` is a useful function (along with its sibling `sprintf`) that allow us to inject values into strings using placeholders. These placeholders are also used to indicate the type of data being supplied: -

- `%d`: int
- `%f`: float
- `%b`: Boolean
- `%s`: string
- `%O`: The `.ToString()` representation of the argument
- `%A`: F# Pretty Print representation of the argument; falls back to `%O` if none exists.

Simply supply the args, space separated, after the raw string. Don't use brackets or commas to separate the arguments to `printfn` – just spaces (the reason for this will be explained!)

Once you've amended the application, running it should show you the following: -



The screenshot shows a Windows command prompt window titled 'cmd.exe' with the path 'C:\WINDOWS\system32'. The window displays the output of a F# program using `printfn`. The output reads: 'Passed in 2 items: [| "HELLO"; "WORLD" |]'. Below the output, there is a message 'Press any key to continue . . .' followed by a cursor icon.

Figure 2.6 - An enhanced Hello World console application in F#

Quick Check

5. What placeholder is used for printing strings in `printf`?

2.4 Summary

In this lesson: -

- You created a simple console application in F#, and learned how to define entry points to them.
- You then explored how to run console applications, and saw that running and debugging F# applications isn't very much different from the C# or VB .NET experience.

- Lastly, you had a first look at the F# language in a standalone application.

Try This

Enhance the application to print out the length of the array as well as the items that were supplied using a combination of `printfn` and the `Length` property on the array (use dot notation, just like you're used to).

Quick Check Answers

1. Console and Class Library.
2. Marking the entry function to a console application.
3. `fs` (compiled file) and `fsx` (script file).
4. `F9`
5. `%s`

3

The REPL – Changing how we develop

In this lesson, we'll look at an alternative way to developing applications than what you're probably currently used to in terms of both tools and process; we'll first discuss the pros and cons of techniques such as debugging and unit tests, and then see how we can shorten our development cycle through some alternatives that F# provides us with.

Think for a moment about your typical development process in terms of how you write and then validate your code. Once you've written some C# or VB .NET, how do you confirm that it works as you intended? What tools or process do you follow? I'm willing to bet it follows one of the following three patterns: -

3.1.1 Application-based Development

You develop code for a while, thinking about the problem at hand. When you think it's "ready", you'll run the application and run the section of the application to stress the code you just wrote. If it behaves as expected, you'll move on to solving the next problem.

The problem with this approach is that it's not particularly efficient – the section of code you've just written may take time to navigate through the application – or simply not be feasible e.g. some code that only gets called under circumstances out of your control e.g. only when the network connection drops.

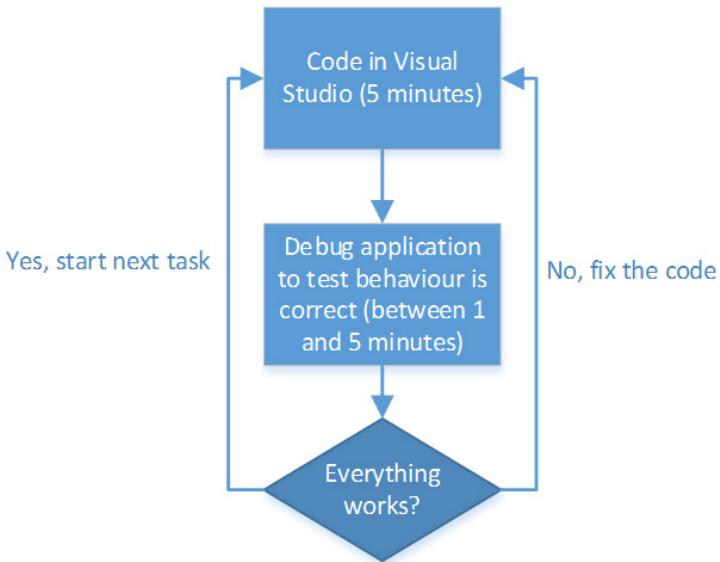


Figure 3.1 Application-Based Development.

3.1.2 Console Test Rigs

You develop code for a while. When you’re ready to test it, you write a console application that calls into the application code that you’ve written directly – perhaps a number of times with specific arguments.

It’s hard to maintain these “console test harnesses” – in fact, the cost of keeping these up to date often grows very quickly, particularly as you want your test rig to grow to cater for more scenarios and areas of your code.

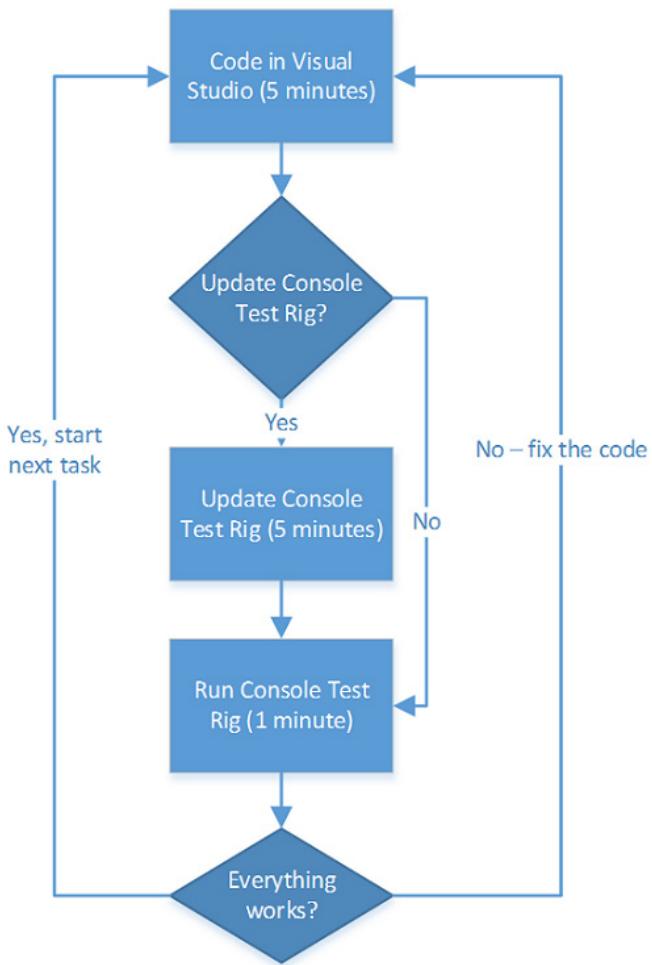


Figure 3.2 Console Test Rigs.

3.1.3 Automated unit tests

Unit tests are a great way to test out parts of code without running a full application. They also give us regression tests so that as we make changes, we know if we've broken existing behaviour. As you can see from the flow chart, the process is more complex than with test rigs. Unfortunately, the truth is that test-driven development (TDD) is a difficult skill to master, and if done "wrongly" can be extremely costly - with fragile unit tests that are difficult to reason about and hard to change. It's also expensive as a means to "explore" a domain - you wouldn't use unit tests as a means to test out a new nuget package, for example.

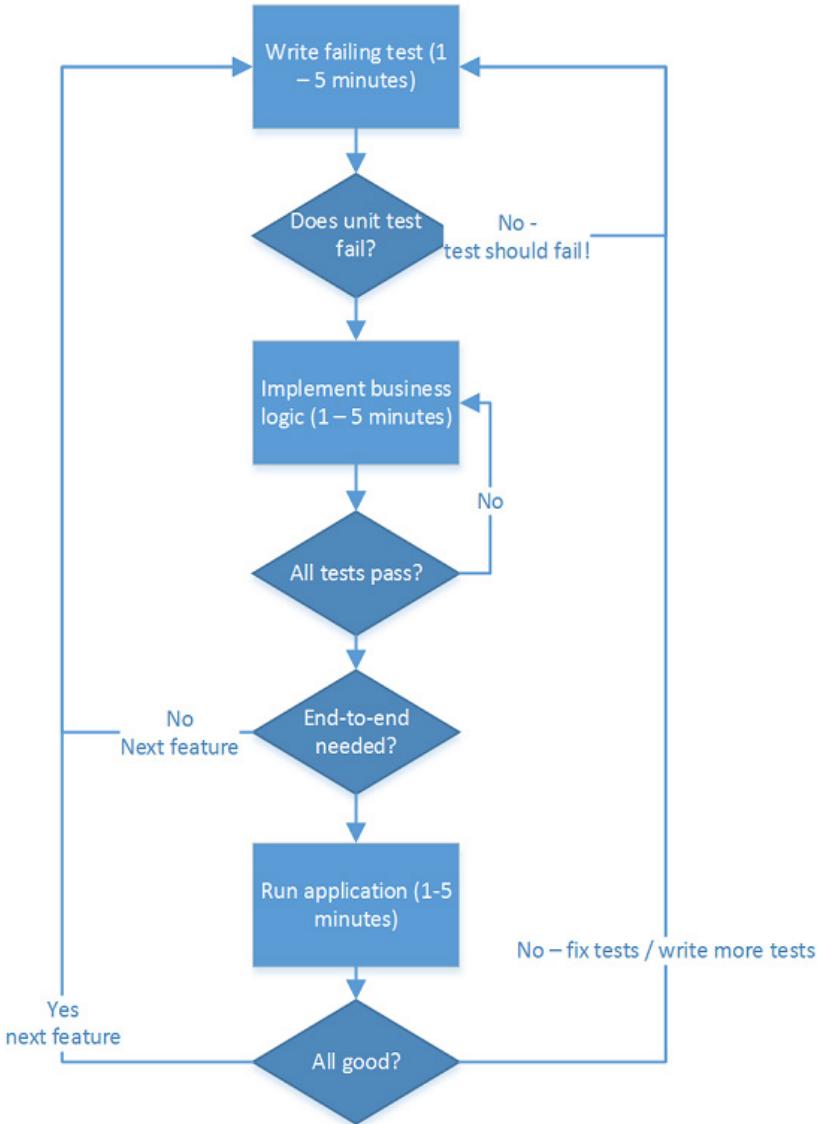


Figure 3.3 Test Driven development process.

I should point out here that I'm actually a big fan of TDD in C# - I used it religiously for many years – and found that it can indeed raise the quality of software. But it's *hard* to get right! And, it doesn't lend itself well to trying out new things quickly. What we need is a lightweight way to "experiment" with some code, test out some ideas, and when we're happy with the

results, push them into a real code base – perhaps along with some unit tests to prove the behaviour.

3.2 Enter the REPL

3.2.1 What is the REPL?

REPL stands for **Read Evaluate Print Loop** – it's a mechanism for you to enter arbitrary, ad-hoc code into a standalone environment and get immediate feedback. The easiest way to think of a REPL is to think of the Immediate Window in Visual Studio, but rather than being used when *running* an application, a REPL is utilised when *developing* an application. You send some code to the REPL, it evaluates it, and then prints the result. This "feedback loop" is a tremendously productive way to develop.

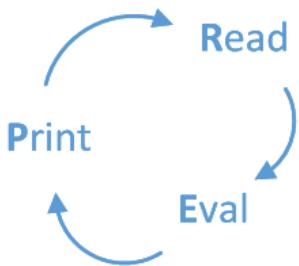


Figure 3.4 - The Read Evaluation Print loop is an effective way to rapidly explore and develop solutions

A REPL can be an effective replacement for all three of the use cases we outlined earlier. Here are some example uses of why and how you would use a REPL: -

- Writing some new business logic that you want to add to your application
- Testing out some existing production code with a predefined set of data
- Exploring a new NuGet package that you've heard about in a lightweight exploratory mode
- Rapid data analysis and exploration - think of tools such as SQL Server Management Studio, LINQPad or similar.

As you can see from the next figure, the emphasis is very much on quick exploration cycles – trying out ideas in a low-cost manner, getting rapid feedback, before pushing that into the application code base. What's nice about F# is that because the language allows us to encode more business rules into the code than in C#, you can often have a great deal of confidence that your code will work, meaning that you will not need to run end-to-ends particularly often. Instead, you'll find yourself working in Visual Studio and the REPL more and more, focusing on writing code that delivers business value.

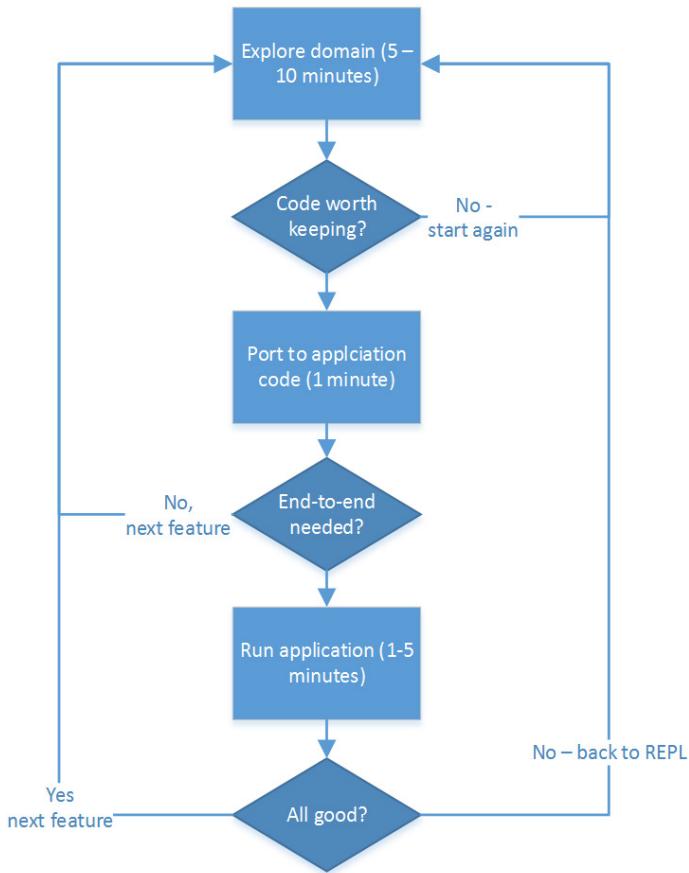


Figure 3.5 – Example REPL driven development

REPLs, REPLs everywhere!

There are many languages and development environments that already have a REPL – in fact, there are full blown standalone applications that are extremely powerful REPLs with built in-charting and reporting facilities, such as Python's IPython Notebook (and more recently, Jupyter). Since newer versions of VS2015, C# now finally does have a basic REPL called C# Interactive.

3.2.2 F# Interactive

Let's try out the REPL that comes with VS2015 for F#, called F# Interactive, or FSI for short.

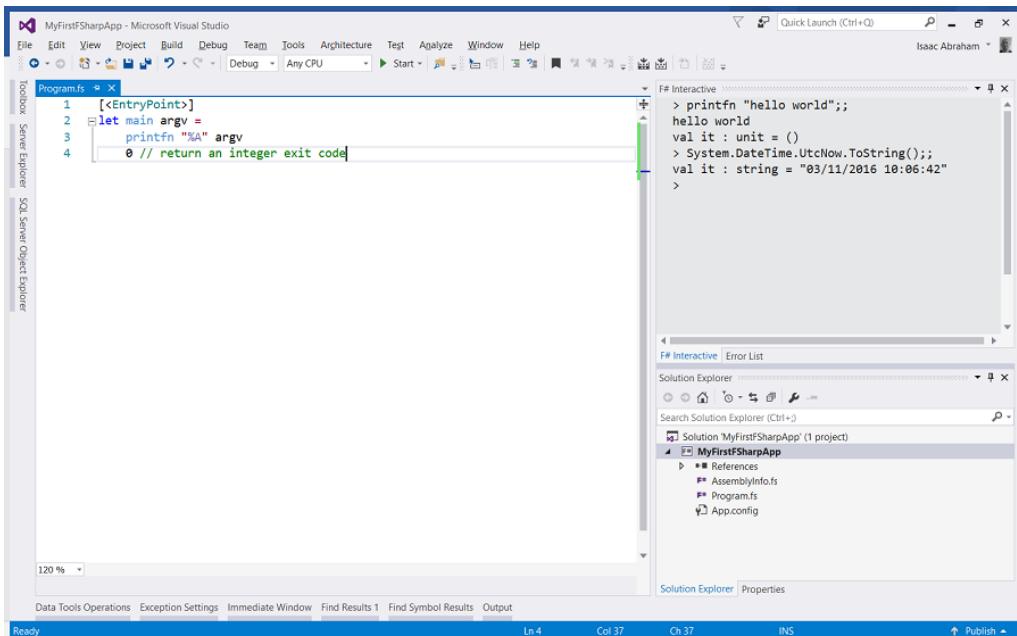


Figure 3.6 The F# development REPL experience in VS2015. The code window is shown on main left, with the top right pane representing F# Interactive (FSI).

Now you try

1. Open up F# Interactive from the **View -> F# Interactive** menu option.
2. You'll be presented with a new tool window. This will most likely be docked to the bottom of the window – my preference is to dock it to right hand side of the window, occupying about 25%-30% of the width of the IDE (although I use a widescreen monitor).
3. Clicking on the FSI pane will allow you to execute any valid F# code.
4. Enter the command `printfn "Hello World";;`
5. You'll see the following output:

```
Hello world!
val it : unit = ()
```

6. Enter the command `System.DateTime.UtcNow.ToString();;`
- ```
val it : string = "06/04/2016 10:30:42"
```

In other words, we're able to execute arbitrary F# code without needing to run an application – Visual Studio itself is a host to our code. FSI outputs anything that we print out (such as the

text “Hello World” as well as information itself, such as the content of values when they are evaluated and so on – you’ll see more in the coming lessons.

### 3.2.3 State in FSI

FSI maintains state across each command, and you can bind the value of a particular expression to a value using the `let` keyword that you can then access in subsequent calls. For example: -

#### **Listing 3.1 A simple let binding**

```
let currentTime = System.DateTime.UtcNow;;
currentTime.TimeOfDay.ToString();;
```

Note the `; ;` at the end of each command. This tells FSI to execute the text currently in the buffer. Without this, it will simply add that command to the buffer until it encounters a `; ;` and will execute all the commands that are in the buffer.

If you want to reset all state in FSI, you can either right-click and choose **Reset Interactive Session** or hit **CTRL + ALT + R**. Similarly, you can clear the output of FSI (but retain its state) using **Clear All** or **CTRL + ALT + C**.

#### **What's “it”?**

You probably noticed the `val it =` text in FSI for commands that you executed. What is this? `it` is simply the default value that any expressions are bound to if you don't explicitly supply one using the `let` keyword. So executing the command `System.DateTime.UtcNow;;` is the same as executing `let it = System.DateTime.UtcNow;;`

#### **Quick Check**

1. What does REPL stand for?
2. Name at least two conventional “processes” used for developing applications.
3. What is the F# REPL called?

## 3.3 F# Scripts in Visual Studio

If you've tried out the exercises above, your first thought is probably something like “Well, this is quite useful, but the IDE support is awful!”. And you're not far wrong. You get no syntax highlighting, no intellisense support - no nothing, really. There are a couple of experimental extensions to improve this (plus some extremely promising work being done in MonDevelop, Xamarin Studio and with Roslyn), but at the moment working directly in FSI isn't a great experience.

### 3.3.1 Creating Scripts in F#

Luckily there's a much better way to work with FSI that *does* give you intellisense and syntax highlighting etc., called F# scripts, or `fsx` files – so let's now create our first script and experiment with it a little.

#### **Now you try**

1. Right-click on your F# project in Visual Studio and select Add -> New Item....
2. From the dialog that pops up, select Installed -> Code -> Script File and set the filename to `scratchpad.fsx`.
3. The file will automatically open in VS.

You'll now be looking at a blank file in the main panel. Entering any text in here will give you full intellisense and code completion etc. just like inside a full `fs` file. Start by entering something simple into the script: -

```
let text = "Hello, world"
```

Hovering over `text` will show you that F# has identified the value as a string, and if you move to the next line and type `text.` you'll see immediately that you get full intellisense for it.

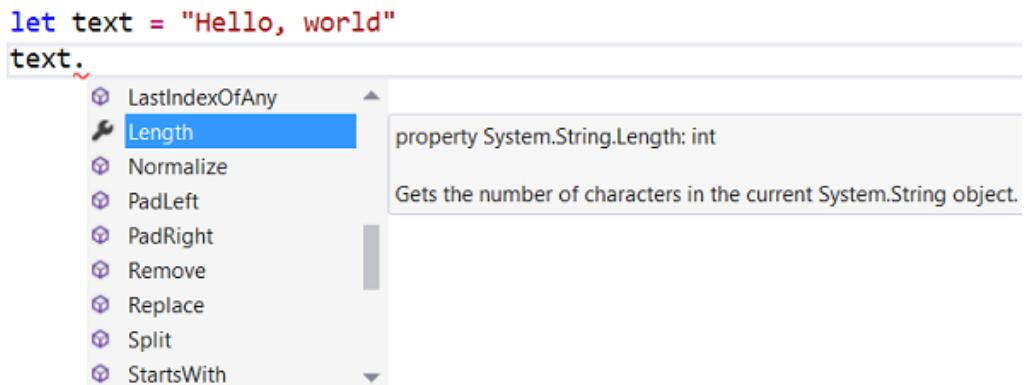


Figure 3.7 Working with F# in an `fsx` script file

In this way, you can work with all .NET types, and values, within a script file for experimentation. Even better, scripts can work together with FSI to give you a fantastic experience, as we'll see next. And the best bit is that a script doesn't need a solution or *project* in which to function. You can simply open up Visual Studio, hit **File -> New**, pick an F# script and start working!

### 3.3.2 The relationship between scripts and FSI

When working with scripts, you'll obviously want some way to "evaluate" the results of code you've entered. This is extremely easy within an F# script – simply hitting **ALT + ENTER** will send the current line to FSI for evaluation. In this way, you can think of the rapid feedback cycle here as somewhat similar to writing unit tests (particularly in a TDD style), albeit in a much more lightweight form. You can also highlight multiple lines and use the same keypress to send all of the highlighted code to FSI.

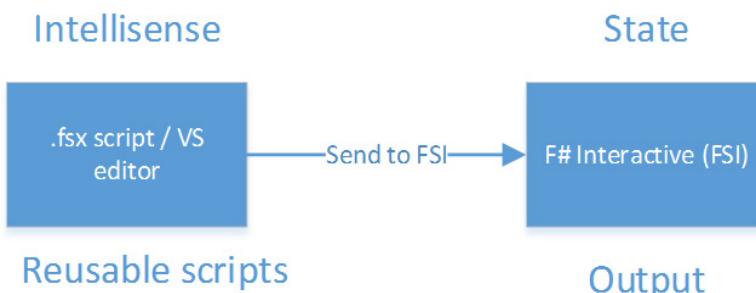


Figure 3.8 – The relationship between scripts and FSI.

#### Resharper and F#

You might find that **ALT + ENTER** doesn't work for you. This might be because you have Resharper installed, which "steals" this shortcut for its own use. You'll have to re-configure Resharper to not do this, or choose a different keyboard shortcut for FSI Send to Interactive from within the keyboard shortcut options of VS.

#### Now you try

We'll now explore the relationship between fsx scripts and FSI.

1. Move to the first line in your script and hit **ALT + ENTER**.
2. Observe that FSI outputs result `val text : string = "Hello, world".`
3. On the next line in the script, send the command `text.Length` to FSI and observe the result in FSI: `val it : int = 12.`
4. Reset FSI.
5. Highlight both lines and send them both to FSI simultaneously.
6. Reset FSI again.
7. Highlight just the second line and send it to FSI. Notice that FSI returns an error that `text` is not defined – you've not sent the first line that defines it to FSI.

So, instead of you having to manually enter code in FSI, you can first enter it into a *repeatable* script and send entire sections of the script to FSI as needed. You can use scripts

as a way to completely remove the need for Console test rigs – instead, you'll just have a set of scripts to test out your code in a pre-defined way.

### 3.3.3 Working with functions in scripts

Of course, you can also define functions in scripts, send them to FSI and then call them from the script on demand. Let's create a function that takes in someone's name and returns a string that is the greeting of the person.

#### **Listing 3.2 A simple function definition**

```
let greetPerson name age =
 sprintf "Hello, %s. You are %d years old" name age
```

#### **Functions and type inference**

Notice that mousing over `greetPerson` will show you the signature of the function – `name` is inferred to be a string and `age` an integer, and the function returns a string. We'll explore this more in lesson 5 – for now it's enough to know that the `sprintf` function tells us that `name` is a string and `age` an int from the `%s` and `%d` specifiers.

Highlighting the entire function and sending it to FSI will compile the function.

On the next line, enter the code

```
let greeting = greetPerson "Fred" 25
```

before sending it to FSI to execute the function call; you can repeatedly highlight this line and send it to FSI. We'll learn more about `let` in lesson 4, but in case you're wondering – repeatedly executing this line does not mutate the value of `greeting` multiple times. Instead, it's creating a new value every time and discarding the previous one.

#### **Quick Check**

4. Do scripts need a project in order to run?
5. Give two reasons why you might use a script rather than coding directly into FSI.

## 3.4 Summary

In the lesson: -

- You explored the REPL, a powerful tool in a developers' arsenal.
- You learned about the relationship between F# Interactive (FSI) and F# Scripts (.fsx files), and how and when to use both of them.
- Finally, you explored writing some more F# code, defining your first function.

### Try This

Spend a little more time in the F# script; write a function that can return the number of words in the string using standard .NET string Split and array functionality. Then, save the string and number of words to disk as a plain text file.

### Quick Check Answers

1. REPL stands for “Read Evaluate Print Loop”.
2. Application-based testing, Console Test Rigs and Unit Testing.
3. F# Interactive, or FSI for short.
4. No – you can run scripts as standalone files.
5. F# scripts have an improved development experience, and are repeatable.

# Unit 2

## Hello F#

In the previous unit, you gained a feeling for the development experience in VS. It's basically not much different from C# and VB .NET insofar as we have the same basic project types, as well as access to the BCL that you already know. However, we only covered the bare essentials of the F# language.

This unit is going to focus on the core foundational elements of F# as a language – how it differs from C# and VB .NET in its aims, how it changes your approach to problem solving in the “small”, and how we learn to work with a compiler and language that make us think in a slightly different way. As you'll see, some of the concepts you'll already be familiar with in C# and VB .NET – it's simply that F# takes those features, and turns the dial up to 11 on them – putting them front and center of the language.

By the end of this unit, you'll have a good understanding of F#'s basic syntax structure and philosophy, as well as experience of working with F#'s rich type inference, working with expressions and statements, as well as understanding the benefits behind immutable data.

# 4

## *Saying a little, doing a lot*

**In this lesson, we'll gain an overview of the basics of the F# language: -**

- We'll take a closer look at the F# language syntax, including the `let` keyword
- You'll learn how to write some more complex functions and values
- You'll learn what scoping is, why it's important for creating readable code, and how it works in F#

Think about the programming languages that you use today. They come in all sorts of different flavours, and are known for being used in different fields or situations. For example, Java is well known as an “enterprise” programming language – which has somewhat negative connotations of a slow moving and verbose language. Others might have a reputation for being used at startups, academia, or in data science. Why are languages “shoehorned” into specific fields, when many of them say that they are “general purpose” programming languages? The answer is that a combination of many factors sometimes help push a language to a specific community or use case.

One of these factors might be the type system. For example, some languages are generally considered to be statically typed (Java, C#) whilst others are dynamic (Python, Javascript, Ruby) etc. The latter have gained a reputation for being used in startups due to the alleged speed at which development can occur for relatively simple systems (although there are several notable examples of organisations having to rewrite entire systems in a static language once the application grew too large, Twitter being one).

Another important feature is the *syntax* of the language. This, just as much as the type system, can turn entire sets of potential developers away from a language. Too verbose and the developer may lose patience with it and move on to another one. Too lightweight and terse, and it might be too difficult for the developer to pick up, particularly if they are used to a language which has a verbose syntax.

These two traits often seem to be naturally grouped: -

**Table 4.1 – Language traits compared**

| Language   | Type     | System   | Syntax                        |
|------------|----------|----------|-------------------------------|
| C#         | Static,  | Simple   | Curly Brace, Verbose          |
| Java       | Static,  | Simple   | Curly Brace, ExtremelyVerbose |
| Scala      | Static,  | Powerful | Curly Brace, Verbose          |
| Python     | Dynamic, | Simple   | Whitespace, Very Lightweight  |
| Ruby       | Dynamic, | Simple   | Whitespace, Very Lightweight  |
| Javascript | Dynamic, | Simple   | Curly Brace, Lightweight      |

The take away is often: Static programming languages are “slower” to develop than dynamic languages, because the syntax is too heavyweight and verbose, and the benefits of a static type system don’t outweigh those costs. However, there are some static languages that exist, such as F# and Haskell, that aim to give the developer the best of both worlds – a powerful, static type system that has an extremely lightweight syntax designed to allow the developer to express their intent without having to worry about lots of different keywords and symbols for everyday use. In this way, and in conjunction with the REPL, we can rapidly develop applications that are underpinned by a powerful compiler and type system.

Before we dive in, let me just reiterate one point: In F#, the overall emphasis is to allow us to solve *complex* problems with *simple* code. We want to focus on solving the problem at hand without necessarily having to *first* think about design patterns within which we can put our code, or complex syntax etc. All the features you’re going to see now are geared towards helping to achieve that. F# does have some common “design pattern”-ish features, but in my experience, they’re fewer and farther between, with less emphasis on them.

## 4.1 Binding values in F#

The `let` keyword is the single most important keyword in the F# language. We use it to bind *values* to *symbols*. In the context of F#, a value can range from a simple value type such as an integer, a POCO, a complex value such as an object with fields and methods or even a function. In C#, we’re not generally used to treating functions as values, but in F#, they are the same - so any value can be bound to a symbol with `let`: -

### Listing 4.1 Sample let bindings

```
let age = 35 1
let website = System.Uri "http://fsharp.org" 2
let add (first, second) = first + second 3
```

- 1 Binding 35 to the symbol age
- 2 Binding a Uri to the symbol website
- 3 Binding a function that adds two numbers together to the symbol add.

Here are some takeaways from that small sample: -

- **No types.** You'll notice that we haven't bothered with specifying any types – the F# compiler will figure these out for us, so if you mouse over age or website, you'll see `int` and `System.Uri` (although you *can* – and occasionally *must* – specify them). This type inference is scattered throughout the language, and is so fundamental to how we work in F# that lesson 5 is dedicated to it entirely (and will explain how the compiler understands that the add function takes in two numbers – it's not magic!).
- **No “new” keyword.** In F#, the new keyword is optional, and generally not used except when constructing objects that implement `IDisposable`. Instead, F# views a constructor as a function, just like any other “normal” function that you might define.
- **No semi colons.** In F#, they are optional – the newline is enough for the compiler to figure out you've finished an expression. You *can* use them, but they're completely unnecessary (unless you want to include multiple expressions on a single line). Generally, you can forget they ever existed.
- **No brackets for function arguments?** You might have already seen this and asked why this is. F# actually has two ways to define function arguments, known as “tupled form” and “curried form”. We'll deal with this distinction in a later lesson, but for now it's fine to say that both when calling and defining them, functions that take a *single* argument don't *need* round brackets (aka parentheses), although you can put them in if you like; functions that take in zero or multiple arguments (as per the add function) need them, as well as commas to separate the arguments, just like C#.

### **Now you try**

Let's experiment with binding values to symbols.

1. Create a new F# script file.
2. Bind some values to symbols yourself: -
  - A simple type such as a `string` or `int`.
  - An object from within the BCL e.g. `System.Random`.
  - Create a simple one-line function that takes in no arguments and calls a function on the object that you created earlier e.g. `random.Next()`
3. Remember to execute each line in the REPL using `ALT + ENTER`.

#### **4.1.1 Let isn't var!**

Don't confuse `let` with `var`. Unlike `var`, which declares *variables* that can be modified later on, `let` binds an *immutable value* to a symbol. The closest thing in C# would be to declare every variable with the `readonly` keyword (although this isn't *entirely* equivalent). It's better to think of `let` bindings as “copy and paste” directives – wherever you see the symbol, simply replace it with the value that was originally assigned during the declaration.

You may have noticed that you can execute the same `let` binding multiple times in FSI. This is because F# allows you to re-purpose a symbol multiple times within the same scope – this is known as *shadowing*.

#### **Listing 4.2 Reusing let bindings**

```
let foo() =
 let x = 10 1
 printfn "%d" (x + 20) 2
 let x = "test" 3
 let x = 50.0 4
 x + 200.0 5
```

- 1 Binding 10 to the symbol `x`.
- 2 Prints out 30 to the Console.
- 3 Binding “test” to the symbol `x`. The original `x` is now out of scope.
- 4 Binding 50.0 to the symbol `x`. The previous `x` is now out of scope.
- 5 Returning 250.0.

Shadowing is a more advanced (and somewhat controversial) feature, so don’t worry too much about it – but this is why you can declare the same symbol multiple times within FSI.

#### **Quick Check**

1. Give at least two examples of values that can be bound to symbols with `let`.
2. What is the difference between `let` and `var`?
3. Is F# a static or dynamic language?

## **4.2 Scoping values**

I’m sure you’ve heard that global variables are a bad thing! *Scoping* of values is important in any language – they allow us not only to show intent by explaining where and when a value is of use within a program, but also protect us from bugs by reducing the possibilities for a value to be used within an application. In C#, we use `{ }` to explicitly mark scope: -

#### **Listing 4.3 Scoping in C#**

```
using System
public static int DoStuffWithTwoNumbers(int first, int second)
{
 var added = first + second;
 Console.WriteLine("{0} + {1} = {2}", first, second, added);
 var doubled = added * 2;
 return doubled;
}
```

In this context, the variable `added` is only *in scope* within the context of the curly braces. Outside of that, it’s *out of scope* and not accessible by the rest of the program. On the other hand, F# is a *whitespace significant* language. This means that rather than using curly braces, we simply have to *indent code* to tell the compiler that we’re in a nested scope.

#### Listing 4.4 Scoping in F#

```
open System
let doStuffWithTwoNumbers(first, second) =
 let added = first + second ①
 Console.WriteLine("{0} + {1} = {2}", first, second, added)
 let doubled = added * 2
 doubled ②
```

- ① Creation of scope for the `doStuffWithTwoNumbers` function  
 ② Return value of the function

#### Functions Arguments in F#

You might have read that F# normally uses spaces to separate function arguments rather than commas. Don't worry about this – for now, we're going to use C#-style syntax for arguments, with commas, but later on we'll learn about this alternative (and more common) way of defining function arguments with spaces.

There's no specific restriction on the number of spaces you should indent by – it can be 1 space or 10 spaces – as long as you're consistent within the scope! Most people tend to use 4 spaces – it's not worth wasting time on picking the indent size, so I'd advise you to just go with that to start with.

You'll see above that we've opened up the `System` namespace so that we can call `Console.WriteLine` directly. You can also get VFPT to open the namespaces for you, just like standard Visual Studio, through the lightbulb tip: -

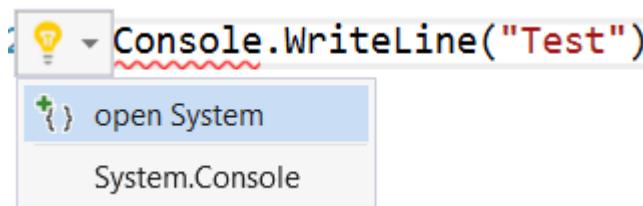


Figure 4.1. Visual F# Power Tools integration in Visual Studio 2015 offering to open up a namespace

You'll also notice a few more things from this multiline function: -

- **No return keyword.** The `return` keyword is unnecessary and not valid F# syntax (although there's one case where they are valid that we'll see in the second half of this book). Instead, F# assumes that the *final expression* of a scope is the result of that scope. So, in this case it's the value of `doubled`.
- **No accessibility modifier.** In F#, `public` is the default for top-level values. There are a number of reasons why this is – but it makes perfect sense in F#, because with nested scopes (see below), you'll find that you can hide values effectively without resorting to accessibility modifiers.

- **No static modifier.** Again, static is the default way of working in F#. This will be different from what you're used to, but it fits with how you'll design most solutions in F#.

### Accessibility Modifiers in F#

It's worth pointing out that in terms of modifiers, although F# supports most of them, there is no `protected` access modifier. This isn't a problem normally – I've certainly never needed it since I started using F#. This is probably because `protected` is a modifier used when working with object oriented hierarchies – something you very rarely use in F#.

## 4.2.1 Nested scopes

We're used to using classes and methods as means of scoping and data hiding. You might have a class that contains private fields and methods, as well as one or many public methods. We can also use methods for data hiding – again, the data is only visible within the context of that function call. In F#, we can define arbitrary scopes at any point we want. Let's assume we wanted to estimate someone's age using the current year: -

### Listing 4.5 Unmanaged scope

```
let year = DateTime.Now.Year
let age = year - 1979
let estimatedAge = sprintf "You are about %d years old!" age
// rest of application...
```

Looking at this code, the only thing we're interested in is the string value `estimatedAge`. The other lines are simply used *as part of the calculation* of that; they're not used anywhere else in our application. But currently, they're at the top level of the code, so anything afterwards that uses `estimatedAge` can also see those two values.

Why is this a problem? Firstly, because it's something more for you as a developer to reason about – where is the `year` value being used? Is any other code somehow depending on it? Secondly (and again, this is slightly less of an issue in F# where values are immutable by default), values that have large scopes tend to negatively impact on a codebase in terms of bugs and / or code smells. In F# we can eliminate this by nesting those values *inside* the scope of `estimatedAge` as far as possible: -

### Listing 4.6 Tightly bound scope

```
let estimatedAge = ①
 let age = ②
 let year = DateTime.Now.Year ③
 year - 1979
 sprintf "You are about %d years old!" age ④
```

① Top-level scope

② Nested scope

③ Value of `year` only visible within scope of "age" value

- ④ Cannot access “year” value.

Now it's clear that `age` is only used by the `estimatedAge` value. Similarly, `DateTime.Now.Year` is only used when calculating `age`. Of course, we can't access any value outside of the scope that they were defined in – so you can think of each of these nested scopes as being “mini classes” if you like – scopes that are used to store data used to generate a value.

## 4.2.2 Nested functions

If you've been paying attention, you'll remember that F# treats *functions* as *values*. This means that we can also create functions *within other functions!* Here's an example of how we can do this in F#: -

### Listing 4.7 Nested (inner) functions

```
let estimateAges(familyName, year1, year2, year3) = ①
 let calculateAge yearOfBirth = ②
 let year = System.DateTime.Now.Year
 year - yearOfBirth

 let estimatedAge1 = calculateAge year1 ③
 let estimatedAge2 = calculateAge year2
 let estimatedAge3 = calculateAge year3

 let averageAge = (estimatedAge1 + estimatedAge2 + estimatedAge3) / 3
 sprintf "Average age for family %s is %d" familyName averageAge
```

- ① Top-level function
- ② Nested function
- ③ Calling the nested function

We declare a function called `estimateAges`, which itself defines a nested helper function called `calculateAge` inside it. It then calls this function three times in order to generate an average age estimate for the three ages that were supplied. The ability to create nested functions means that you can start to think of functions and classes that have a single public method as *interchangeable*.

| Class                              | Function                         |
|------------------------------------|----------------------------------|
| Constructor / single public method | Arguments passed to the function |
| Private Fields                     | Local Values                     |
| Private Methods                    | Local Functions                  |

### Capturing values in F#

Within the body of a nested function (or indeed any nested value), code can access any values defined in its containing (parent) scope without you having to explicitly supply them as arguments to the nested function. You can think of this as similar to a lambda function in C# “capturing” a value declared in its parents’ scope. When we return such a code block, this is known as a *closure*; it’s very common to do this in F# - without even realising it.

### Cyclical Dependencies in F#

This is one of the best “prescriptive” features of F# that many developers coming from C# and VB are shocked by: F# does not (easily) permit cyclical dependencies. What this means in effect is that in F# the *order in which types are defined matters*. Type A cannot reference Type B if it is declared before it, and the same applies to values.

Even more surprising is that this applies to all the files in a project – so file order in a project actually matters! Files at the *bottom* of the project can access types and values defined *above* them, but not the other way around. You can manually move files up and down in VS by selecting the file and pressing ALT + UP or ALT + DOWN (or right clicking on a file and choosing the appropriate option).

As it turns out though, this “restriction” actually turns into a “feature”. By forcing you to avoid cyclic dependencies, the design of your solutions will naturally become easier to reason about, since all dependencies will only ever face “upwards”.

### Now you try

Within a script file, we’ll create a Windows Forms form that will contain a `WebBrowser` control which will host the content of a web resource that you’ll download. Here’s a snippet to get you started: -

#### **Listing 4.8 Creating a Form to display a web page**

```
open System
open System.Net
open System.Windows.Forms ①

let webClient = new WebClient()
let fsharpOrg = webClient.DownloadString(Uri "http://fsharp.org")
let browser = new WebBrowser(ErrorsSuppressed = true, Dock = DockStyle.Fill,
 DocumentText = fsharpOrg) ②
let form = new Form(Text = "Hello from F#!")
form.Controls.Add browser
form.Show()
```

① Opening up namespaces in F#

② Object initializer-style syntax in F#

Rewrite the code above so that the scopes are more tightly defined. For example, `webClient` is only used during creation of the `fsharpOrg` value, so it can live within the definition of that value.

Next, try to make the code a function so that it can be supplied with a url instead of being hard-coded to `fsharp.org`. Creating functions from simple sets of values is incredibly easy in F#: -

1. Indent all the code you wish to make a function.
2. On the line above the code block, define the function along with the argument(s) you wish to take in.
3. Replace the "hard coded" values in the code block with the arguments you defined.
4. Ensure that the last line in the function block is not a `let` statement, but an expression.

Here's an example of refactoring a set of arbitrary assignments and expressions into a reusable function to get you started.

#### **Listing 4.9 Refactoring to functions - before**

```
let r = System.Random()
let nextValue = r.Next(1, 6)
let answer = nextValue + 10
```

#### **Listing 4.10 Refactoring to functions - after**

```
let generateRandomNumber max =
 let r = System.Random() 1
 let nextValue = r.Next(1, max) 2
 nextValue + 10 3
```

- 1 Function declaration added
- 2 Code block indented
- 3 Hard coded value replaced with argument
- 4 Return value as an expression

#### **Quick Check**

4. How do you indicate a new scope in F#?
5. Can you declare functions within a nested scope?
6. Do we normally need to use the return keyword to return from a scope?

### **4.3 Summary**

In this lesson: -

- You learned a great deal about the most fundamental parts of the syntax of F#
- You learned about the `let` keyword
- You saw how scoping works in F#

Hopefully you've seen that it contains a relatively simple and minimalistic syntax, with a minimum of extra symbols and keywords. A large majority of what you've seen so far is simply "stripping away" what turned out to be "unnecessary" syntax features of C# - but some elements of the language will have been new to you, such as nested scopes and the lack of support for cyclic dependencies.

### ***Try This***

Explore scoping in more depth - try creating a set of functions that are deeply nested within one another. What happens if you call a function – e.g. Random.Next() – within another function as opposed to simply using the result of it? What implication does this have for e.g. caching?

### ***Quick Check Answers***

1. Primitive values, values of custom types, functions.
2. Let is an immutable binding of a symbol. Var represents a pointer to a specific mutable object.
3. F# is a statically typed language.
4. Indent code to declare a new scope.
5. Yes, functions can be declared within a nested scope.
6. No – the return keyword is not used in F# to specify the result of an expression.

# 5

## *Trusting the Compiler*

The compiler is one of the most important features in any language, but particularly in a language like F# where the compiler does a lot of heavy lifting for you, it's important that you understand the role it plays in your day-to-day development cycle. In this lesson: -

- We're going to a look at the F# compiler from a developer's point of view (don't get scared!)
- We'll focus specifically on one area of it – type inference.
- You'll recap what type inference is from a C# / VB .NET perspective
- We'll look at how F# takes type inference to the next level

### 5.1 Type inference as we know it

Unless you've only used earlier versions of C#, you'll almost certainly be familiar with the `var` keyword. Let's re-familiarise ourselves with the `var` keyword based on the official MSDN documentation.

Variables that are declared at method scope can have an implicit type `var`. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type.

#### 5.1.1 Type Inference in C# in detail

Here's a simple example of that in action: -

##### **Listing 5.1 Using var in C#**

```
var i = 10; ①
int i = 10; ②
```

- ① Implicitly typed
- ② Explicitly typed

Of course, the right-hand-side of the = can be just about any expression. So, we can write more complicated code that may defer to another method, perhaps in another class. Naturally, you need to give a little bit of thought regarding the naming of variables when using var!

#### **Listing 5.2 Variable naming with type inference**

```
var i = customer.Withdraw(50); ①
var newBalance = customer.Withdraw(50); ②
```

- ① Implicitly typed. Withdraw() returns an int, so i is inferred to be an int.
- ② Use of intelligent naming to explain intent to the reader.

#### **The multipurpose var**

There is another reason for the use of var in C# - to store references to types that have no formal declaration aka Anonymous Types. Anonymous Types don't exist in F#, although as we'll see later, you very, very rarely miss them as there are some very good alternatives that are in many ways more powerful.

It's important to stress that var mustn't be confused with the dynamic keyword in C#, which is (as the name suggests) all about dynamic typing. var allows us to use static typing without the need to explicitly specify the type by *allowing the compiler to determine the type for us at compile time*.

```
var name = "Isaac Abraham";
name.ToLower();
var number = 123;
number.ToLower();
```

'int' does not contain a definition for 'ToLower'

Figure 5.1 – Simple Type Inference offered by C#

#### **5.1.2 Practical Benefits of Type Inference**

Even in its restricted form in C#, type inference can be a nice feature. The most obvious benefit is that of readability – we can focus on getting results from method calls etc. and use the human-readable name of the value to gain its meaning, rather than the type. This is especially useful with generics – and F# uses generics *a lot*. However, there is another subtler benefit that we gain – the ease of refactoring.

Imagine we have a method Save() that stores data in the database and returns an integer value – let's assume that this is the number of rows saved. We then call it in our main code: -

**Listing 5.3 Depending on method results with explicit typing**

```
int result = Save(); ①
if (result == 0) ②
 Console.WriteLine("Failure!");
else
 Console.WriteLine("Worked!");
```

- ① Explicit binding to int  
 ② Where the value is actually explicitly used as a int

Note that we're explicitly marking `result` as an integer, although the actual declaration of the variable could just have well have been `var`. Then at some point in the future, we decide that we want to return a boolean that represents success / failure instead. We have to change two things: -

1. We need to manually change the method signature to specify that the method returns `bool` rather than `int`. There's no way around this in C#.
2. Now we need to go through every callsite to `Save()` and manually fix it to bind the `result` to `bool` rather than `int`. If we had used `var`, this wouldn't have been a problem at all, because we'd have left the compiler to "figure out" the type of `result`.

Of course, even when using `var`, at some point you would normally have to make *some* kind of change to your code to handle a `bool` instead of an `int` – in this case, it's the conditional expression for the `if` statement. So, although `var` won't fix *everything* for you – it's not magic – the difference is that the compiler would have taken care of fixing the "boilerplate" error for you automatically, leaving you to actually change the "real" logic (i.e. changing the expression from comparing with 0 to comparing with `true`).

**Critics of Type Inference**

There are a few developers that shy away from type inference. The more common complaint I hear is that it's "magic", or alternatively that one cannot see at a glance what type a variable is. The first point can be easily dispelled simply by reading the rules for type inference – the compiler doesn't guess the types; there are a set of precedence rules that guide it. The second point can also be dispelled by the number of excellent IDEs (including VS2015) that give you mouse-over guidance for types, as well as following good practices such as sane variable naming (which is generally a good thing to do). Overall, particularly in F#, the benefits massively outweigh any costs.

**5.1.3 Imagining a more powerful Type Inference system**

Unfortunately, type inference in C# and VB .NET are restricted to the single use case I've illustrated above. Let's look at a slightly larger code snippet: -

**Listing 5.4 Hypothetical type inference in C#**

```
public static var CloseAccount(var balance, var customer)
{
```

```

if (balance < 0) ①
 return false; ②
else
{
 customer.Withdraw(customer.AccountBalance); ④
 customer.Close();
 return true; ③
}

```

- ① Balance compared with 0
- ② Returning a Boolean
- ③ Returning a Boolean
- ④ Calling methods and accessing properties on a type

This is invalid C#, because I've omitted all types. But couldn't the compiler possibly "work out" the return type or input arguments based the following: -

1. We're comparing `balance` with 0. Perhaps this is a good indicator that `balance` is also an integer. However, it could also be a float or other numeric type?
2. We're returning `Boolean` values from all possible branches of the method. Perhaps we want the method to return `Boolean`?
3. We're accessing methods and properties on the `customer` object. How many types in the application are there that have `Withdraw` and `Close` methods and an `AccountBalance` property (which is also compatible with the input argument of `Withdraw`)?

### Quick Check

1. Can you think of any limitations of the C# type inference engine?
2. What is the difference between dynamic typing and type inference?

## 5.2 F# Type Inference Basics

We've discussed some of the benefits of type inference in C#, as well as some of the issues and concerns about it. All of these are magnified with F#, because type inference in F# is *pervasive*. You can literally write entire applications without making a single type annotation (although this isn't always possible, nor always desirable). In F#, the compiler can infer: -

- Local bindings (as per C#)
- Input arguments for both in-built and custom F# types
- Return types

F# uses a sophisticated algorithm which relies on what's known as the Hindley–Milner type system. It's not especially important to know what that really is, although feel free to read up on it in your own time! What *is* important to know is that HM type systems *do* impose some restrictions in order to operate that might surprise you, as we'll see shortly.

So, without further ado let's finally get onto some F#! Thus far, all the examples you've seen in F# haven't used type annotations, but I'll show you a simple example now that we can break down piece-by-piece to understand how it works.

#### **Listing 5.5 Explicit type annotations in F#**

```
let add (a:int, b:int) : int =
 let answer:int = a + b
 answer
```

We'll cover functions in more depth later on, but to get us going here, a type signature in F# has three main parts: -

- The function name
- All input argument type(s)
- The return type

You can see from the code sample that both input arguments `a` and `b` are of type `int`, and the function also returns an `int`.

#### **Type annotations in C# and F#**

Many C# developers recoil when they see types declared *after* the name of the value. In fact, most languages outside of C / C++ / Java / C# use this pattern. It's particularly common in languages like F# where type inference is especially powerful, or optional type systems like Typescript.

Start by removing just *the return type* from the type signature of the function; when you compile this function in FSI, you'll see that the type signature is exactly the same as before: -

#### **Listing 5.6 Omitting the return type from a function in F#**

```
let add (a:int, b:int) =
...
// val add : a:int * b:int -> int
```

F# infers the return type of the function, based off the result of the final expression in the function. In this case, that's `answer`. Now go one step further and remove the type annotation from `b`. Again, when you compile, the type signature will be the same. In this case, it raises an interesting question: How does the compiler know that `b` isn't a float or decimal etc.? The answer is that in F#, *implicit conversions are not allowed*. This is another feature of the type system that helps enforce safety and correctness, although it's not something that you'll necessarily be used to. In my experience, it's not a problem at all. And given this restriction, the compiler can safely make the assumption that `b` is an `int`. Finally, let's remove the remaining two type type annotations: -

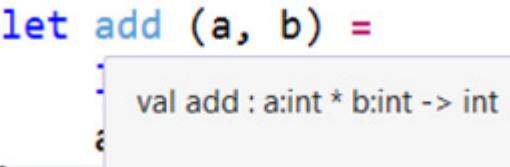


Figure 5.2 - F# Type Inference operating on a function

Amazingly, the compiler still says that the types are all integers! How has it figured this out? In this case, it's because the `+` operator binds *by default* to integers, so all the values are inferred to be ints.

### **Now you try**

Let's now experiment with this code a little more to see the compiler respond to code changes.

1. Mix some type annotations on the function – for example, mark `a` as `int` and `b` as `string`. Does it compile?
2. Remove all the type annotations again, and rewrite the body to add an explicit value e.g.

```
a + b + "hello"
```

3. Does this compile? What are the types? Why?
4. What happens if you call the function with an incompatible value?

Earlier, we demonstrated that type inference can not only improve readability by removing unnecessary keywords that can obscure the meaning of your code, but also speed up refactoring – for example, by allowing you to change the return type of a function without necessarily breaking the caller. This benefit is increased by a significant factor when working with F#, since you can automatically change the return type just by changing the implementation of a function without needing to manually update the function signature. Indeed, when coupled with its lightweight syntax and ability to create scopes simply by indenting code, it means you can create new functions and change type signatures of existing code *incredibly* easily – particularly because type inference in F# can escape local scope, unlike in C#.

#### **5.2.1 Limitations of type inference**

There are a few more restrictions and limitations in F# related to type inference – let's go through them one by one here.

## WORKING WITH THE BCL

Firstly, type inference works best with types *native* to F#. By this I mean basic types such as ints, or F# types that you define yourself. We've not looked at F# types yet, so this won't mean much to you, but if you try to work with a type from a C# library (and this includes the .NET BCL), type inference won't work *quite* as well - although often a single annotation will suffice within a codebase: -

### Listing 5.7 Type inference when working with BCL types in F#

```
let getLength name = sprintf "Name is %d letters." name.Length ①
let getLength (name:string) = sprintf "Name is %d letters." name.Length ②
let foo(name) = "Hello! " + getLength(name) ③
```

- ① Doesn't compile – type annotation is required
- ② Compiles
- ③ Compiles – “name” argument is inferred to be string based on the call to getLength()

The first function won't compile, as the F# compiler doesn't know that name is a `String` (and therefore has a `Length` property). The second version works, because of the annotation. Any code that calls that function *won't* need an annotation – the initial one will “bleed out” naturally.

## CLASSES AND OVERLOADED METHODS

Secondly, in F#, overloaded functions are not allowed. You can create (or reference from C# libraries) *classes* that contain *methods* which are overloaded, but *functions* declared using the `let` syntax cannot be overloaded. For this reason, type inference doesn't completely function on classes and methods.

### 5.2.2 Type Inferred Generics

F# can apply type inference not just on simple values but also for *type arguments* - we can either use the `_` to specify a “placeholder” for the generic type argument, or simply omit the argument completely: -

### Listing 5.8 Inferred type arguments in F#

```
open System.Collections.Generic
let numbers = List<_>() ①
numbers.Add(10)
numbers.Add(20)

let otherNumbers = List() ②
otherNumbers.Add(10)
otherNumbers.Add(20)
```

- ① Creating a generic List, but omitting the type argument
- ② This syntax is also legal

You should understand that F# infers the type based on the first available usage of the type argument. So the call to `numbers.Add(10)` is used to tell the compiler that the List is of type int. If you were to call `numbers.Add` with 10 and then "Hello", you'll get a compiler error on the second call, as by this stage the compiler has selected int as the type argument.

F# will also automatically make functions generic when needed. Let's try out a simple function which adds some items to a list. In this example, there's no type specified for the value `output` anywhere in the code – so the compiler can't infer the type of `List`. In this case, it will simply make the entire `createList()` function generic!

#### **Listing 5.9 Automatic generalization of a function**

```
let createList(first, second) =
 let output = List()
 output.Add(first)
 output.Add(second)
 output
// val createList : first:'a * second:'a -> List<'a>
```

In this case, you can think of `'a` as the same as `T` in C# i.e. a generic type argument. You can specify the generic argument placeholder (and use it as a type annotation) if you really want to e.g. `let createList<'a>(first:'a, second)` but you should generally just let the compiler infer the arguments – it's very powerful, and will save you a lot of time.

#### **Quick Check**

3. How does F# infer the return type of a function?
4. Can F# infer types from the BCL?
5. Does F# allow implicit conversions between numeric types?

### **5.3 Following the breadcrumbs**

Unlike C#, because type inference escapes function scope in F#, the compiler will go through your entire code-base and notify you where the types *eventually* clash. This is normally a good thing, but it does mean that occasionally you'll find that you need to remember how the type inference system works in order to diagnose compiler errors. Let's look at a relatively simple example, and look at how making changes to the types in it can lead to errors occurring in unusual places: –

#### **Listing 5.10 Complex type inference example**

```
let sayHello(someValue) = ①
 let innerFunction(number) = ②
 if number > 10 then "Isaac"
 elif number > 20 then "Fred"
 else "Sara"

 let resultOfInner = ③
 if someValue < 10.0 then innerFunction(5)
 else innerFunction(15)
```

```

 "Hello " + resultOfInner ④
let result = sayHello(10.5) ⑤

① : Function declaration
② : innerFunction - signature is int -> string
③ : String result of calling innerFunction()
④ : String result of overall function
⑤ : Sample callsite

```

If you follow the flow, you'll notice that the current logic suggests that the "Fred" branch will never be called; don't worry about that – we're more interested in the type system and F#'s inference engine here.

### Now you try

Copy the code from listing 5.10 into an F# script in VS; everything will compile by default. Now let's see how we can break this code! We'll start by first changing the **first** if / then case in `innerFunction` to compare against a string ("hello") rather than an int (10):

```

let innerFunction(number) =
 if number > "hello" then "Isaac"
 elif number > ~~~ then "Fred"
 else "Sara"

let resultOfInner =
 if someValue < 10.0 then innerFunction(5)
 else innerFunction(~~~)

```

Figure 5.3 – Following the breadcrumbs with type inference

You'll see that rather than this line showing an error, we'll see errors in three *other* places! Why is this? This is what I refer to as "following the breadcrumbs" – you'll need to track through at least one of the errors and see the inferred types to understand *why* this has happened. Let's look into the first error and see if we can work out why it's occurring: **This expression was expected to have type string but here has type int.** Remember to mouse over the values and functions to get intellisense of the type signatures being inferred!

1. Looking at the compiler error message, we can see that the callsite to `innerFunction` now expects a `string`, although we know that it *should* really be an `int`.

2. Let's then look at the function signature of `innerFunction`. It used to be `int -> string`, but now is `string -> string` i.e. given a `string`, it returns a `string`.
3. Let's look at the function body. We can see that the first branch of the if / then code now returns a `string` rather than an `int`. So, the compiler has used this to infer that the function should take in a `string`.
4. We can prove this by hovering over the value "number", which sure enough is now inferred to be a `string`.
5. To help us, and to guide the compiler, let's temporarily explicitly type annotate the function as per Figure 5.4.
6. You'll see now that the "false" compiler errors disappear, and the compiler now correctly identifies the error as being "hello", which should actually be an `int`.
7. Once you correct the error, you can remove the type annotation again.

```
let innerFunction(number:int) =
 if number > "hello" then "Isaac"
 elif number > 20 then "Fred"
 else "Sara"
```

Figure 5.4 – Explicit type annotations can help to drill down on the source of an error

From this example, you can see that adding in type annotations *can* sometimes be useful, particularly when trying to narrow down an error caused by clashing types. I would recommend, however, that you in general try to avoid working with explicit types everywhere – your code will look much cleaner as a result.

### Now you try

Try out some more examples of changing values to experiment with how F# type inference works: -

1. Replace "Isaac" with 123. Look at the different errors that show up. Why do they appear?
2. Replace "Fred" with 123. Why is the error different to when you changed "Isaac"?
3. Replace 10.0 with 10 - what happened? Why?

### Quick check

6. Why are type annotations sometimes useful when looking at compiler errors?

## 5.4 Summary

That was a pretty intensive lesson! We covered: -

- Basics of type inference
- Simple type inference in C# / VB .NET

- Type inference in F#
- Limitations of F# type inference
- Diagnosing type inference problems in F#

It's well worth us spending the time to understand type inference in F#, because it's a crucial part of the "flavour" of the language – again, fitting with the "more with less" philosophy, as well as another side of F# that we discussed at the start of this lesson, which is trusting the compiler.

It's a very different mindset to simply create functions and arguments without type annotations and let the compiler fill in the gaps, and as you saw, there are times when it's very important that you have an understanding of what the compiler is actually doing under the hood. However, as we'll see over the coming lessons, type inference is incredibly useful in writing succinct, easily refactorable code without needing to resort to a third-party tool to "rewrite" your code for you.

### **Try This**

Try creating other generic objects that you already know within the BCL. How does F# work with them? Then, experiment with the code that you created in the previous lessons. Can you remove any of the type annotations? How does it affect the "look and feel" of the code?

### **Quick Check Answers**

1. Types cannot be inferred from method scope, and are only valid on assignment.
2. The latter is statically typed, but types are resolved by the compiling. Dynamic typing truly does not specify types at compile time.
3. Based on the type of the last expression in the function.
4. No, although it can infer BCL types across functions declared in F#.
5. No.
6. Temporarily placing explicit type annotations allows you to "guide" the compiler with what your intention is; this can help track down when types are incompatible in code.

# 6

## *Working with immutable data*

**Working with immutable data is one of the more difficult aspects of functional programming to deal with, but as it turns out, once you get over the initial hurdle, you'll be surprised just how easy it is to write entire applications working with purely immutable data structures. It also goes hand in hand with many other F# features we'll see, such as expression-based development. In this lesson, you'll see: -**

- The basic syntax for working with immutable and mutable data in F#
- Some reasons for why you should consider immutability by default in software development today
- Some simple examples of working with immutable values to manage changing state

Let's start by thinking about some of the issues we come up against today, but often take for granted as simply "the way things are". Here are a few examples that I've either seen first hand or fallen foul of myself.

### **6.1.1 The unrepeatable bug**

You're developing an application, and one of the test team come up to you with a bug report. You walk over to their desk and see the problem happening. Luckily, your tester is running in Visual Studio, so you can see the stack trace and so on. You look through the locals and application state, and figure out why the bug is showing up. Unfortunately, you have no idea how the application got into this state in the first place – it's the result of calling a number of methods repeatedly over time with some shared mutable state stored in the middle.

You go back to your machine and try to get the same error, but this time you can't reproduce it. You file a bug in your work item tracking system and wait to see if you can get lucky and figure out how the application got into this state.

### 6.1.2 Multithreading pitfalls

Sound familiar? How about this one. You're developing an application, and have decided to use multi-threading because it's cool. You also recently heard about the Task Parallel Library in .NET, which makes writing multi-threaded code a lot easier, and also saw that there's a `ForEach.Parallel()` method in the BCL. Great! You've also read about locking and so on – so you carefully put locks around the bits of the shared state of your application that are affected by the multi-threaded code. You test it locally, and even write some unit tests – everything is green! You release, and two weeks later find a bug that you eventually trace to your multi-threaded code. You don't know why it happened though – it's caused by some race condition that only occurs under a specific load and a certain ordering of messages. Eventually you revert your code back to a single threaded model.

### 6.1.3 Accidentally sharing state

Here's another one. You've working on a team, and have designed a business object class. Your colleague has written some code to operate on that object. You call their code, supplying an object, and then carry on. Some time later, you notice a bug in your application – the state of business object no longer looks as it did previously! It turns out that the code your colleague wrote modified a property on the object without you realising. You only made that property public so that *you* could change it – you didn't intend or expect *other* bits of code to change the state of it! You fix the problem by making an interface for the type which exposes the bits that are "really" public on the type, and give that to consumers instead.

### 6.1.4 Testing hidden state

Or maybe you're writing unit tests. You want to test a specific method on your class, but unfortunately to run a specific branch of that method, you need to get the object into a specific state first. This involves mocking a bunch of dependencies that are needed to run the *other* methods; only then can you run your method. Then, you try to assert whether the method worked – but the only way to prove that the method worked properly it is to access some shared state that is private to the class. Your deadlines are fast approaching, so you change the accessibility of the private field to be Internal, and make internals visible to your test project.

*All* of these problems are real issues that occur on a regular basis, and they're nearly always due to mutability. The problem is often that we simply assume that mutability is a way of life, something that we can't escape, and so look for other ways around these sorts of issues – things like encapsulation, hacks like `InternalsVisibleTo` or one of the many design patterns that are out there. It turns out that working with *immutable* data solves many of these problems in one fell swoop.

## 6.2 Being explicit about mutation

So far, we've only looked at simple values in F#, but even these show us that by default, values are immutable. As you'll see in later lessons, this also applies to your own custom F# types e.g. Records.

### 6.2.1 Mutability Basics in F#

Let's see immutability in action here by opening a script file and entering the following code: -

#### **Listing 6.1 Creating immutable values in F#**

```
let name = "isaac" ①
name = "kate" ②
```

- ① Creating an immutable value
- ② Trying to assign "kate" to name

You'll notice when you execute this code that you receive the following output in FSI: -

```
val name : string = "isaac"
val it : bool = false
```

The false does not mean that the assignment has somehow failed. It's because in F#, the = operator represents equality, just like == does in C#. So all we've done is compare "isaac" with "kate", which is obviously false.

So how do we "update" or mutate a value? The answer is that we use the assignment operator, <-. Unfortunately trying to insert that into our code will lead to an error: -

#### **Listing 6.2 Trying to mutate an immutable value**

```
name <- "kate"
error FS0027: This value is not mutable
```

Oops! This still doesn't work. It turns out that there's one final step we need to do to make a value mutable, which is to use the `mutable` keyword: -

#### **Listing 6.3 Creating a mutable variable**

```
let mutable name = "isaac" ①
name <- "kate" ②
```

- ① Defining a mutable variable
- ② Assigning a new value to the variable

If you installed and configured Visual F# Power Tools, you'll notice that the name value is now automatically highlighted in red as a warning that this is a mutable value. You can think of this as the inverse of C# and VB .NET, whereby we use *variables* by default, and explicitly mark individual items as immutable *values* using the `readonly` keyword.

The reason that F# makes this decision is to help guide us down what I refer to as the “pit of success”; we *can* use mutation when needed but must be explicit about it, and should do so in a carefully controlled manner. However, by default we should go down the route of adopting immutable values and data structures.

As it turns out, you can pretty easily develop (and I have done) entire applications (with web front ends, SQL databases etc.) using only immutable data structures – and you’ll be surprised when you realise how little you actually need mutable data, particularly in request / response style applications e.g. web applications, which are inherently stateless.

### 6.2.2 Working with mutable objects

Before we move onto working with immutable data, here’s a quick primer on the syntax for working with mutable *objects*. I don’t recommend you create your own mutable types, but working with the BCL is a fact of life as a .NET developer, and the BCL is inherently OO-based and filled with mutable structures, so it’s good to know how to interact with them.

#### Now you try

Let’s start by creating a good old Windows Form, displaying it, and then setting a few properties of the window.

#### **Listing 6.4 Working with mutable objects**

```
open System.Windows.Forms
let form = new Form() ①
form.Show()
form.Width <- 400 ②
form.Height <- 400
form.Text <- "Hello from F#!"
```

- ① Creating the form object
- ② Mutating the form using the <- operator

#### **Mutable bindings and objects**

Most objects in the BCL are inherently mutable – such as a Form object. Notice that the binding form symbol is *immutable*, so the binding symbol itself cannot be changed – but the object it refers to is *itself* mutable, so properties on that object can be changed!

Notice that you can see the mutation of the form happen through the REPL i.e. if you execute the first three lines, you start with an empty form, but after executing the final line, the title bar will immediately change: -



**Figure 6.1 Creating a simple Form from an F# script**

F# also has a shortcut for creating mutable data structures in a way that assigns *all* properties in a single action, somewhat similar to object initializers in C#, except in F# it works simply by making properties appear as optional constructor arguments: -

#### **Listing 6.5 Shorthand for creating mutable objects**

```
open System.Windows.Forms
let form = new Form(Text = "Hello from F#!", Width = 300, Height = 300) ①
form.Show()
```

① Creating and mutating properties of a Form in one expression

Of course, if there are actual constructor arguments required as well, you can put them in there at the same time (VS2015 sadly doesn't give intellisense for setting mutable properties in the constructor).

#### **Quick Check**

1. What keyword do you use to mark a value mutable in F#?
2. What is the difference between = in C# and F#?
3. What keyword do we use in F# to update the value of a mutable object?

### **6.3 Modelling state**

Let's now look at the work needed to model data with state without resorting to mutation.

#### **6.3.1 Working with mutable data**

Working with mutable data structures in the OO world essentially follows a simple model – you create an object, and then modify its state through operations on that object.

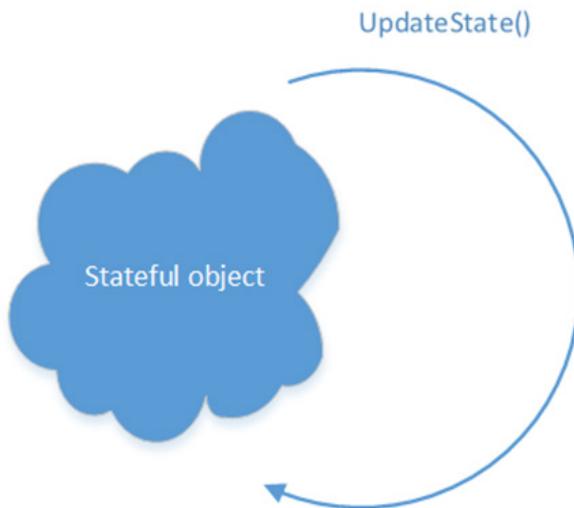


Figure 6.2 Mutating an object repeatedly

What's tricky about this model of working is that it can be hard to reason about your code. Calling a method like `UpdateState()` above will generally have no return value; the result of calling the method is a *side effect* that takes place on the object.

### **Now you try**

Let's now put this into practice with a simple example – driving a car. We want to be able to write some code that allows us to `drive()` a car, tracking the amount of petrol used; depending on the distance we drive, we should use up a different amount of petrol.

#### **Listing 6.6 Managing state with mutable variables**

```
let mutable petrol = 100.0 ①
let drive(distance) =
 if distance = "far" then petrol <- petrol / 2.0
 elif distance = "medium" then petrol <- petrol - 10.0
 else petrol <- petrol - 1.0
drive("far") ③
drive("medium")
drive("short")
petrol ④
```

- ① Initial state
- ② Modify state through mutation
- ③ Repeatedly modify state
- ④ Check current state

Working like this, it's worth noting a few things: -

1. Calling `drive()` has no outputs. We call it, and it silently modifies the mutable `petrol` variable – we can't know this from the type system.
2. Methods are not deterministic. You can't know what the behaviour of a method is without knowing what the (often hidden) state is. If you call `drive("far")` 3 times, the value of `petrol` will change every time, depending on the previous calls.
3. We have no control over the ordering of method calls. If you switch the order of calls to `drive()`, you'll get a different answer.

### 6.3.2 Working with immutable data

Let's now compare that with working with immutable data structures.

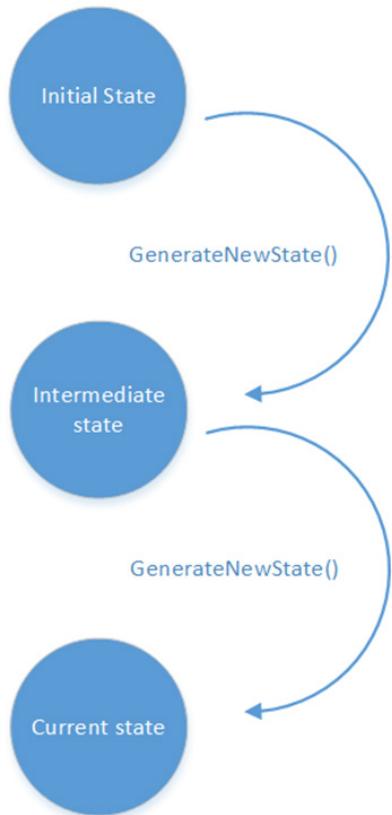


Figure 6.3. Generating new states working with immutable data

In this mode of operation, we can't mutate data. So instead, we create *copies* of the state with updates applied, and return that back out for the caller to work with; that state may be passed in to other calls that themselves generate new state.

### Performance of immutable data

I often hear this question asked – isn't it much slower to make copies all the time rather than modify a single object? The answer is yes and no. Yes, it is slower to copy an object graph than simply make an in place update. However, unless you're in a tight loop performing millions of mutations, the cost of doing so is negligible compared to e.g. opening a database connection. Plus, many languages (including F#) have specific data structures designed to work with immutable data in a highly performant manner.

Let's now rewrite our code to use immutable data.

#### **Listing 6.7 Managing state with immutable values**

```
let drive(petrol, distance) = ①
 if distance = "far" then petrol / 2.0
 elif distance = "medium" then petrol - 10.0
 else petrol - 1.0

let petrol = 100.0
let firstState = drive(petrol, "far") ②
let secondState = drive(firstState, "medium") ③
let finalState = drive(secondState, "short") ④
```

- ① Function explicitly dependent on state - takes in petrol and distance, and returns new petrol
- ② Initial state
- ③ Storing output state in a value
- ④ Chaining calls together manually

We've made a few key changes to our code. The most obvious is that we aren't using a mutable variable for our state any longer, but a set of immutable *values*. We "thread" the state through each function call, storing the intermediate states in values which are then manually passed to the next function call. Working in this manner, we gain a few benefits immediately: -

1. We can reason about behaviour much more easily. Rather than hidden side effects on private fields, each method or function call can return a new version of the state which we can easily understand. This makes unit testing much easier for example.
2. Function calls are repeatable. We can call `drive(50, "far")` as many times as we want, and it will always give us the same result. This is because the the only values that can affect the result are supplied as input arguments – there's no "global state" that's implicitly used. This is known as a **pure function**. Pure functions have some very nice properties such as being able to be cached or pre-generated, as well as being easier to test.

3. The compiler is able to protect us in this case from accidentally mis-ordering function calls, because each function call is explicitly dependent on the output of the previous call.
4. We can also see the value of each intermediate step as we “work up” towards the final state.

### **Passing immutable state in F#**

In this example, you'll see that we're manually storing intermediate state and explicitly passing that to the next function call. That's not strictly necessary, and we'll see in future lessons how F# has language syntax to avoid having to do this explicitly.

#### **Now you try**

Let's try to make some changes to our drive code.

1. Instead of using a string to represent how far we've driven, use an integer.
2. Instead of “far”, check if the distance is more than 50.
3. Instead of “medium”, check if the distance is more than 25.
4. If the distance is > 0, reduce petrol by 1.
5. If the distance is 0, make no change to the petrol consumption. In other words, simply return back out the same state that was provided.

#### **6.3.3 Other benefits of immutable data**

A few other benefits that aren't necessarily obvious from the above sample: -

1. When working with immutable data, encapsulation isn't necessarily as important as it is when working with mutable data. There are certainly times where encapsulation is still valuable e.g. as part of a public API - but there are occasions where simply by making your data read-only, the need to “hide” your data goes away.
2. Multi-threading. We'll see more of this later, but one of the other benefits of working with immutable data is that you don't need to worry about locks within a multi-threaded environment. Since there's never any shared *mutable* state, you never have to be concerned with race conditions – every thread can access the same piece of data as often as it likes as it can never change.

#### **Quick Check**

4. How do we handle changes in state when working with immutable data?
5. What is a pure function?
6. What impact does working with immutable data have with multi-threading code?

## 6.4 Summary

In this lesson: -

- We started by identifying some areas where mutable data structures can cause problems
- We saw how immutable data can act as a form of state through copy-and-update that works particularly well with pure functions, whilst avoiding *side effects* to allow us to more easily reason about our code.
- You saw a simple example of how we can create and work with immutable data in F#

This is only the beginning, and you'll see more examples of how immutable data is a core part of F# throughout this book. Also important is that F# encourages us to work with immutable data *by default* – but as F# is a pragmatic language, it always allows you to “opt out” of this by using the `mutable` keyword and `<-` operators. This is particularly useful when working with types from the BCL and / or other libraries written in C# or VB .NET that are inherently mutable. However, just like working with *immutable* data in C# is a bit of extra work and not necessarily idiomatic – so the inverse is true in F#.

### Try This

1. Try modelling another state machine with immutable data – for example, a kettle that can be filled with water, poured out into a teapot or directly into a cup.
2. Look at working with some BCL classes which are inherently mutable e.g. `System.Net.WebClient`. Explore the different ways to create and modify them.

### Quick Check Answers

1. The `mutable` keyword.
2. In F#, `=` performs an equality between two values. It can also be used for binding a value to a symbol. In C#, `=` always means assignment.
3. F# uses the `<-` operator to update a mutable value.
4. By creating copies of existing data with applied changes.
5. A function which only varies based on the arguments explicitly passed to it.
6. Immutable data does not need to be locked when working across multiple threads.

## 7

## *Expressions and Statements*

**Expressions and statements are two aspects of programming that we use often, and generally take for granted, but in F# the distinction between the two is much starker than you might be used to. In this lesson: -**

- You'll understand the differences between statements and expressions
- Understand the pros and cons of both
- See how expressions in combination with the F# type system and compiler can help you write code that is more succinct as well as easier to reason about

### **7.1 Comparing Statements and Expressions**

Before we dive in, let's quickly recap what statements and expressions are. Here are two definitions taken directly from the C# documentation on MSDN – first statements, then expressions.

The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition.

<https://msdn.microsoft.com/en-us/library/ms173143.aspx>

An expression is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace.

<https://msdn.microsoft.com/en-us/library/ms173144.aspx>

One of these is written in relatively plain (but somewhat verbose) English - the other is plain confusing (to me, at least!). So, let's redefine the two terms more succinctly and appropriately for this purposes of this lesson: -

**Table 7.1 – Statements and Expressions compared**

|             | Returns something? | Side-effectful? |
|-------------|--------------------|-----------------|
| Statements  | Never              | Always          |
| Expressions | Always             | Rarely          |

In a nutshell, that's it. In C#, we're used to methods *sometimes* returning values, and a few operators such as + - / etc. or null coalesce (??) etc. But we're not used to handling *program flow* as expressions. Instead, language constructs in C# are generally all statement based, and rely on side-effects to make changes in the system – something we've already shown as potentially being difficult to reason about. But how can we write applications when program flow *constructs* are expressions?

### 7.1.1 Difficulties with Statements

When working in languages such as C# and VB .NET, we often don't think about the difference between statements and expressions, as these languages mix and match both features throughout the language. Indeed, I would suggest that both of these languages are primarily *statement-based languages* in that statements are easy to achieve, but expressions are not.

Here's a simple example to help us compare and contrast statements and expressions. We'll start with a method that takes in someone's age, and tries to print out a string that describes the person.

#### **Listing 7.1 Working with statements in C#**

```
public void DescribeAge(int age)
{
 string ageDescription = null; ①
 var greeting = "Hello"; ②

 if (age < 18)
 ageDescription = "Child!"; ③
 else if (age < 65)
 greeting = "Adult!"; ④

 Console.WriteLine($"{greeting}! You are a '{ageDescription}'.");
}
```

- ① Initialise a mutable variable to some default
- ② Creating a mutable variable to use later
- ③ First if branch
- ④ Second if branch

There are several issues with this code, all caused by the fact that if / then in C# is a way of controlling program flow with a set of arbitrary, *unrelated* statements: -

1. There's no handler for the case when `age >= 65`. We're not actually accessing any properties on the string, so we won't get any null reference exceptions; instead we'll just print out null. If you deliberately avoid setting `ageDescription` to *any* value, the compiler will actually give you a warning regarding using an uninitialised variable – but initialising it to `null` satisfies the compiler!
2. We've accidentally assigned the string to `greeting`, rather than `ageDescription`, in the second case.
3. We had to declare `ageDescription` with a default value before assigning it. This opens up the possibility of all sorts of bugs for more complex logic.

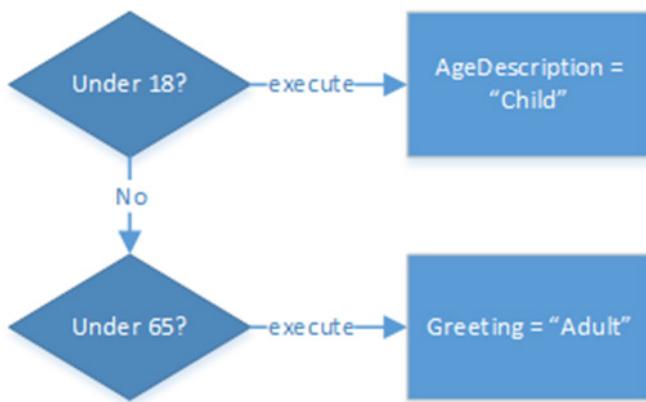


Figure 7.1 – Illustrating the flow of an if / else statement

Your initial instinct might be simply to say that no-one makes mistakes like that, and that this is a strawman example. But you'd be amazed how many bugs creep in from situations just like this – particularly as a codebase grows in size and these sorts of code smells begin to manifest themselves in strange ways.

### 7.1.2 Making life better through Expressions

In the previous example, we saw a number of issues that are all valid C# code, yet are mistakes that you were (hopefully!) able to identify quickly and easily. Why can't the compiler fix these things for us? Why can't it help us get these things right first time?

The answer is that statements are *weak* - compilers simply have no understanding that there's any *relationship* between all of the branches of the if / else block. Instead, they're simply different paths to go down and execute - the fact that they're all supposed to assign a value to the same variable is *purely coincidental*.

We need a construct that's a little bit more powerful in order for the compiler to understand what we're trying to achieve – and as it turns out, we can fix all of these problems in one fell swoop in C# by rewriting our code as follows: -

### Listing 7.2 Working with expressions in C#

```
private static string GetText(int age) { ①
 if (age < 18) return "Child!";
 else if (age < 65) return "Adult!";
 else return "OAP!";
}

public void DescribeAge(int age) {
 var ageDescription = GetText(age); ②
 var greeting = "Hello";
 Console.WriteLine($"{greeting}! You are a '{ageDescription}'.");
}
```

- ① Expression with signature int -> string
- ② Callsite to function

We've now split our code into two methods – one which has the single responsibility of generating the description, and the other that calls it and uses the result later on. There's the “obvious” benefit that moving the code into a separate method might improve readability, but the real benefits are now shown by the way we're naturally forced to structure our code: -

1. We can no longer omit the “else” case when generating the description; if we do this, the C# compiler will stop us with the error `not all code paths return a value`.
2. We can't accidentally assign the description to the wrong variable in half of the cases, because the assignment to `ageDescription` is only performed in one location.
3. We don't need to have a null-initialised variable floating around now either.

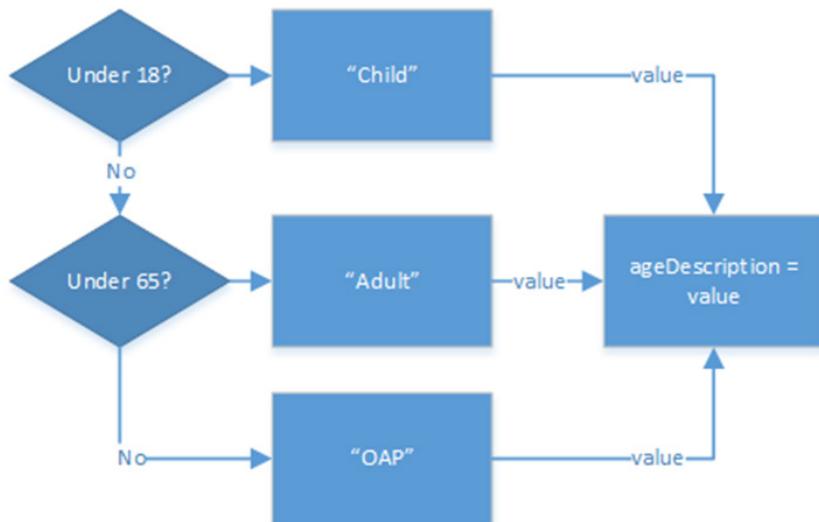


Figure 7.2 – Illustrating the flow of an if / else expression

### Quick Check

1. How often do Expressions return a value?
2. How often do Statements utilize side-effects?
3. What is the smallest unit of expression in C#?

## 7.2 Expressions in F#

### 7.2.1 Working with Expressions

F# firmly encourages expressions as the default way of working. In fact, virtually everything in F# is an expression! For example: -

- There is no notion of a void function in F# - every function must return something (although, once again, there's a nice escape hatch if you need to write code that has no result)
- All program flow branching mechanisms are expressions
- All values are expressions

This illustrates why F# doesn't need a return keyword at the end of a function – because *everthing* is an expression, the last expression within a function *must* be the return value. Let's now rewrite our original C# sample in F# and see the differences: -

#### **Listing 7.3 Working with expressions in F#**

```
open System
let describeAge age =
 let ageDescription = ①
 if age < 18 then "Child!" ②
 elif age < 65 then "Adult!"
 else "OAP!"

 let greeting = "Hello"
 Console.WriteLine("{0}! You are a '{1}'.", greeting, ageDescription)
```

- ① Value binding  
 ② if / else expression branches

The key thing to observe in this code sample is that the if / then block of code has a *result* which is assigned to `ageDescription`. This is different from within the C# block – in this case, it acts more like a function in that it has an (implicit) input i.e. `age` and an explicit result i.e. either "Child", "Adult" or "OAP", which is then assigned to `ageDescription`. By moving to this way of working with expressions, we get the same benefits that we did in C# except here they're a first-class part of the language – we don't have to move our code to extra methods to benefit from the extra safety that expressions provide. As a further benefit, also notice that we are no longer relying on mutable data any more – both string values are immutable by default, which fits very nicely with this expression-based mode of development.

## 7.2.2 Composability

A further benefit of expressions is that they encourage *composability*. Imagine that we wanted to modify the original C# method in order to write the result to disk rather than print to the console. Because we're simply writing to the console as part of the whole method (and therefore returning void), there's nothing to "act on". But if we separate the method into two parts – one that generates a string and another that outputs to the console – we can reuse the first part much more easily (as well as make unit testing simpler).

### Listing 7.4 Composability through expressions in C#

```
public string DescribeAge(int age) { ①
 var ageDescription = default(string);
 // logic elided
 return ageDescription;
}

public void DescribeAndPrint(int age) {
 var description = DescribeAge(age); ②
 Console.WriteLine($"{greeting}! You are a '{ageDescription}'.");
}
```

- ① Reusable business logic – now returns string
- ② Compose business logic with console output

## 7.2.3 Introducing Unit

I mentioned that F# does not allow methods to return void. How on earth does this work in F# then – particularly when the BCL probably has thousands of methods that return void? The answer is that F# has a type called `unit`. In fact, you've probably already seen this term floating around in intellisense occasionally. The `unit` type is found in place of any method that would in C# normally return void, but unlike void, appears in F# to be a regular object that can get returned from any piece of code. In this way, we can say that every function returns a value – even if that value is `unit`. Similarly, every function can be thought of as always taking in at least one input value, even if that value is `unit`.

Let's look at some functions and methods from both the code above and the BCL. You can even bind the value of `unit` to a symbol, just like any other "normal" value!

### Listing 7.5 Replacing void methods with functions that return unit

```
describeAge : age:int -> unit
System.Console.WriteLine : unit -> unit
"Test".GetHashCode : unit -> int
let x = describeAge 20 // val x : unit = ()
```

### Now you try

Let's quickly look at `unit` with a few practical examples.

1. Create an instance of unit using standard let binding syntax – the right-hand-side of the equals just needs to be () .
2. Call the describeAge function and assign the result of the function call to a separate value.
3. Check if the two values are equal to one another. What is the result?

Why is all this `unit` business important? What's wrong with `void`? One reason is that `void` is a “special case” within the C# type system. Normal rules don't apply to it, which is why you have many situations where you have two versions of the same type in the BCL – a good example being the `Task` type in .NET, which has both `Task` and `Task<T>`. In F#, all you would really need is `Task<T>`, because even a function that returns nothing would be `Task<unit>`.

#### **Unit isn't quite an object**

Unfortunately, despite unifying the type system, at runtime `unit` doesn't behave quite like a proper .NET object – for example, don't try to call `GetHashCode()` or `GetType()` on it – you'll get a null reference exception. Hopefully a future version of F# will fix this, but you can still think of `Unit` as a singleton object if it helps you to visualise it.

### **7.2.4 Disarding results**

F# also helps tell us that we might be doing something wrong if we call a function and don't utilise the return value.

#### **Listing 7.6 Discarding the result of an expression**

```
let writeTextToDisk text = ①
 let path = System.IO.Path.GetTempFileName()
 System.IO.File.WriteAllText(path, text)
 path

let createManyFiles() = ②
 writeTextToDisk "The quick brown fox jumped over the lazy dog"
 writeTextToDisk "The quick brown fox jumped over the lazy dog"
 writeTextToDisk "The quick brown fox jumped over the lazy dog"

createManyFiles() ③
```

- ① Writes text to disk
- ② Writes several files to disk
- ③ Calls the function

You'll notice a warning in the Visual Studio code editor under the first two calls in the `createManyFiles()` function: -

This expression should have type 'unit', but has type 'string'. Use 'ignore' to discard the result of the expression, or 'let' to bind the result to a name.

In other words, the compiler is warning you that the `writeToDisk()` function is returning something (in this case, the generated file name) and we're simply discarding it! In our case,

we might decide to now change the code to make a note of all the filenames that were generated and collate them together into a list to return to the caller.

In a pure functional language, such as Haskell, it wouldn't make sense to discard the value of a function call, because everything is an expression and there are no side effects. In the impure .NET world, this isn't the case – there are many functions that perform side effects such as file IO, database access etc.

In our case, perhaps we aren't interested in the resulting filename. Or replace this call with one to ADO .NET that performs a SQL command – the result is the number of rows updated. Again, perhaps you're not interested in this. So, we can remove the warning above by explicitly wrapping the result in the `ignore` function. `ignore` simply takes in a value and discards it, before simply returning `unit` – and because F# allows us to silently ignore expressions that return `unit`, the warning goes away.

#### **Listing 7.7 Explicitly ignoring the result of an expression**

```
let createManyFiles() =
 ignore(writeTextToDisk "The quick brown fox jumped over the lazy dog")
 ignore(writeTextToDisk "The quick brown fox jumped over the lazy dog")
 writeTextToDisk "The quick brown fox jumped over the lazy dog"
```

#### **Quick check**

4. What is the difference between a function returning `unit` and a void method?
5. What is the purpose of the `ignore` function in F#?

### **7.3 Forcing statement-based evaluation**

Moving to expressions means that we can't get away any longer with things like unfinished if / else branches, or even early return statements from functions. There are times that you might need to work with statement-like evaluation (although it should definitely be the exception to the rule). We can do this in F# by ensuring that code in a given branch etc. returns `unit`:

#### **Listing 7.8 Forcing statement-based code with Unit**

```
let now = System.DateTime.UtcNow.TimeOfDay.TotalHours

if now < 12.0 then Console.WriteLine "It's morning" ①
elif now < 18.0 then Console.WriteLine "It's afternoon"
elif now < 20.0 then ignore(5 + 5) ②
else () ③
```

- ① `Console.WriteLine` returns `unit`
- ② ignoring an expression to return `unit`
- ③ Optional - explicitly returning `unit` for the final case

By the way, the `else` branch here is actually optional: Because the first three branches all simply return `unit`, F# allows us to implicitly "ignore" the `else` branch as well and it fills it in for

us. In this way, we've turned the if/else expression into a statement – there's no result of the conditional, just a set of side effects that return unit.

Of course, going with a statement-based approach means that we're right back where we started, with no type checks around dealing with all cases, and leading us down the path of relying on mutable data.

### Cryptic compiler errors

One aspect of F# 4 that I'm *not* especially fond of is the error messages that it spits out, which are a throwback to F#'s OCaml roots. For example, if you create an if / else expression that returns a string value for the if branch, but forget to handle an else branch, you'll see an error similar to this:

```
This expression was expected to have type string but here has type unit
```

What the compiler is really saying is that you're missing the "else" case, so please add one that returns a string. Thankfully there's a concerted effort being put forward by the community to contribute a number of changes to the error messages in the compiler to improve this situation, and this situation should be much improved in time for the next release of F#.

### Quick Check

6. Is it possible to work with statements rather than expressions in F#?

## 7.4 Summary

In this lesson: -

- You learned about the differences between statements and expressions.
- You saw how by moving from statements to expressions we benefit from being able to better reason about our code, with the added bonus that the compiler can catch more bugs for us at an earlier stage.
- You saw how expressions in F# are a fundamental feature of the language, whereas statements are shied away from – leading us down the road of writing code in a manner that is less likely to result in bugs.

As we move through the next set of lessons, you'll see expressions more and more in the language.

### Try This

Try to port some statement-oriented code you've written in C# to F#, making it expression-based in the process. What's the impact it had? Then, create a program that can read the user's full name from the console and print back out their first name only. Thinking about

expressions, can you write the application so that the main logic is expression based? What impact does this have on coupling to the Console?

### **Quick Check Answers**

1. Always.
2. Always.
3. A method is generally the smallest way to create an expression in C#.
4. Unit is a type that represents the absence of a specific value. Functions can be return unit and take it in as an argument. Void is a custom feature in the C# language for methods that have no return type.
5. Ignore allows us to explicitly discard the result of a function call.
6. Yes, using tricks such as ignore to ensure that branches return unit.

# 8

## Capstone 1

This lesson is a slightly different one to what we've covered so far. Instead of covering a new language feature, we'll try to solve a larger exercise that's designed to bring together all of the lessons that we've covered so far in the book. So, in this lesson you'll be expected to: -

- Make changes to an existing F# application in Visual Studio
- Use the REPL as a development playground to help you develop solutions
- Port code from scripts into an F# application that is compiled into an assembly
- Write code using expressions and immutable data structures

### 8.1 Defining the problem

For this exercise, we're going to work on a codebase that builds on our "petrol car" sample from earlier in this unit. The objective will be to write a simple application that can drive the car to a number of destinations without running out of petrol. A basic application structure has already been written for you for the console runner, but the implementation of the actual core code needs to be done.

1. Your car starts with 100 units of petrol.
2. You can drive to one of four different destinations. Each destination will consume a different amount of petrol: -
  - Home – 25 units
  - Office – 50 units
  - Stadium – 25 units
  - Gas – 10 units
3. If the user tries to drive anywhere else, the system will reject the request
4. If the user tries to drive somewhere and does not have enough petrol, the system will reject the request

5. When the user travels to the Gas station, the amount of petrol that they have should increase by 50 units

## 8.2 Some advice before you start...

Here's a few tips before starting this exercise: -

1. Use the REPL and a script file to explore different ideas. See if you can build up everything in script form, before looking to build a standalone application. If you get anything wrong – don't worry! The whole point of a script is to allow you to cheaply try out ideas – if an idea doesn't work, just try again. You want to explore the domain in a care-free manner. Once you find something that feels right, move on to the next stage.
2. Avoid mutation by default.
3. Favour expressions and pure functions over statements.
4. Don't worry if your code feels more "procedural" than "functional" at this stage. Given the limited amount of lessons we've gone through so far, that's not a problem.

## 8.3 Starting small

The easiest way to get something up and running quickly is, as usual, to start in the REPL. One of the core differences to how I tend to approach problems in the FP world compared to the OO world is to simply start by implementing small functions that I know are more or less correct, without worrying too much about how they'll be used later on. As long as the functions don't rely on any shared external state, they can be used just about anywhere without a problem. With that in mind, let's start!

### 8.3.1 Solution overview

You'll see in the `src/code-listings/lesson-08` folder is a pre-built `Capstone1.sln` solution for you to open. This contains a few files: -

- `Program.fs` – the console "runner" of the application. This has already been written for you, and we'll briefly review it shortly.
- `Car.fs` – contains the logic that you'll create to implement the rules above.
- `Scratchpad.fsx` – Will be used to explore the domain and experiment with some code.

There's also a `Car - Solution.fs` file which contains a suggested solution. Don't look at this unless you really need to!

Let's now review the `Program.fs` file. As we've already seen, this contains a `main` function which takes in some arguments (`argv`) – we won't use them here though.

#### **Listing 8.1 – The Main routine in our Program**

```
while true do
 try ①
 let destination = getDestination() ②
```

```

printfn "Trying to drive to %s" destination
petrol <- driveTo(petrol, destination) ③
printfn "Made it to %s! You have %f petrol left" destination petrol
with ex -> printfn "ERROR: %s" ex.Message ④
0 ⑤

```

- ① Start of a try / with exception handling block
- ② Get the destination from the user
- ③ Get updated petrol from core code and mutate state
- ④ Handle any exceptions
- ⑤ Return code

There are some interesting bits to mention here. Firstly, you'll notice the try / with block, F#'s equivalent of try / catch. It works in pretty much the same way, with support for the equivalent of exception filters and so on. In general, functional programmers tend to shy away from exceptions, especially as a way of managing control flow, so I don't want you to think about this sample as instructive – rather it's because we haven't learned all the functional tools yet to give us a valid alternative!

Secondly, you'll notice we're using a while loop here. This isn't exactly idiomatic F#, and using a while loop essentially forces us to use a mutable variable. However, note that our mutable data is *isolated* – it's in a single place, and Visual Studio highlights it in a different colour to warn us. This is a key point of working in F# – in many applications, there *will* be a few places where mutation (or a side effect) is difficult to get rid of – and there's nothing inherently wrong with that. What *is* important is that you try to *restrict* mutation to *just those places*, and favour immutability everywhere else. We'll learn tricks in throughout the rest of this book how to avoid common mutation pitfalls, including the above "while loop over state" challenge.

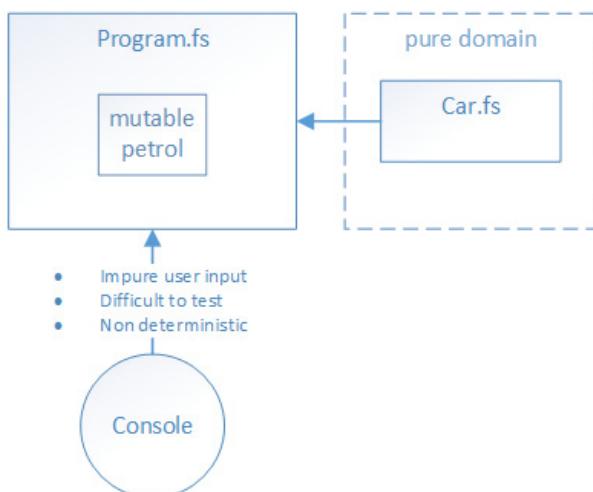


Figure 8.1 – Separation between impure Program.fs and pure business logic

By doing this, we can design our core domain to be entirely pure, easy to reason about and test, and isolate the bits of impurity and untestability to the console runner.

## 8.4 Implementing core logic

To implement our solution, we'll start by working from a script file. We'll test out individual functions, building up to larger functions, before eventually migrating over to a full-blown application. To that end, open up `scratchpad.fsx` in the solution in VS.

### 8.4.1 Our first function – calculating distances

Let's start by implementing a function that can figure out how many units of petrol will be used by driving to a specific location - simple. This function should have one input – the destination to drive to – and should return the amount of petrol required to get there (don't worry – in this example, the distance needed is always the same, irrespective of where you are). Here's a stub function to get you going.

#### **Listing 8.1 – Creating a function to calculate distances**

```
// Gets the distance to a given destination
let getDistance (destination) =
 if destination = "Gas" then 10
 /// remaining implementation elided...
 else failwith "Unknown destination!"
```

- ➊ Function definition
- ➋ Checking the destination and returning an int as an answer
- ➌ Throwing an exception if we can't find a match

You can use the `elif` keyword for the other custom branches of code. The `failwith` keyword is a quick way to throw an exception with a message – although there are other ways (such as the `raise` keyword for custom exception types).

Now that you've written the function, you should test it out. We don't need to worry about unit tests and the like at this point – we'll just use the script itself to check it! There are a couple of example test calls already included in the script for you.

1. First, compile the `getDistance` function by highlighting it and hitting ALT + ENTER.
2. Execute both test cases one at a time - they should both return `true`.

On its own, this function is pretty useless – so let's continue writing some more functions which will then put us in a position to tie it all together.

### 8.4.2 Calculating petrol consumption

Next up, we'll need a function that can calculate the amount of petrol remaining after driving a specific amount. This should be another pretty simple function to do; here's an example function definition: -

```
let calculateRemainingPetrol(currentPetrol:int, distance:int) : int = ...
```

1. As long as the petrol is greater than or equal to the distance needed, this should return the new petrol amount.
2. Otherwise, it should throw an exception with the message "Ooops! You've run out of petrol!".
3. Again, once you've developed this function, you should first test it in isolation.

#### 8.4.3 Composing functions together

Now that we've created a couple of useful (albeit limited) functions, let's build a larger function to pull them together. First, let's just test out in the script that we can call the functions together.

##### **Listing 8.2 – Testing out orchestration of several functions**

```
let distanceToGas = getDistance("Gas") ①
calculateRemainingPetrol(25, distanceToGas) // should return 15
calculateRemainingPetrol(5, distanceToGas) // should throw
```

- ① Calling the `getDistance` function

If that all worked, we can now build a proper function that will orchestrate the two functions together, `driveTo`. This function should take in some current petrol and a target destination. Next, work out the distance using `getDistance` and use that to call `calculateRemainingPetrol`, the result of which you should return. Your function definition should look like this: -

```
let driveTo (petrol:int, destination:string) : int = ...
```

#### 8.4.4 Stopping at the Gas station

The last part we need to do is to add in some extra logic to check if we went to the gas station, and if we did, to increase total petrol by 50. You can simply write this logic directly into the `driveTo` function after you've called `calculateRemainingPetrol`; if the destination was "Gas", add 50 onto the output, otherwise just return what was output by `calculateRemainingPetrol`. Note to add the 50 units of petrol on after you've safely driven to the gas station and not before – you have to drive there first!

### 8.5 Testing in scripts

Before we move over to a full-blown application, we can test this all out in isolation in our script. Here's a simple test case you can try out: -

##### **Listing 8.3 – A test case in script**

```
let a = driveTo(100, "Office") ①
let b = driveTo(a, "Stadium")
```

```
let c = driveTo(b, "Gas")
let answer = driveTo(c, "Home") ②
```

- ① A number of chained calls to our top-level function
- ② Answer should be 40.

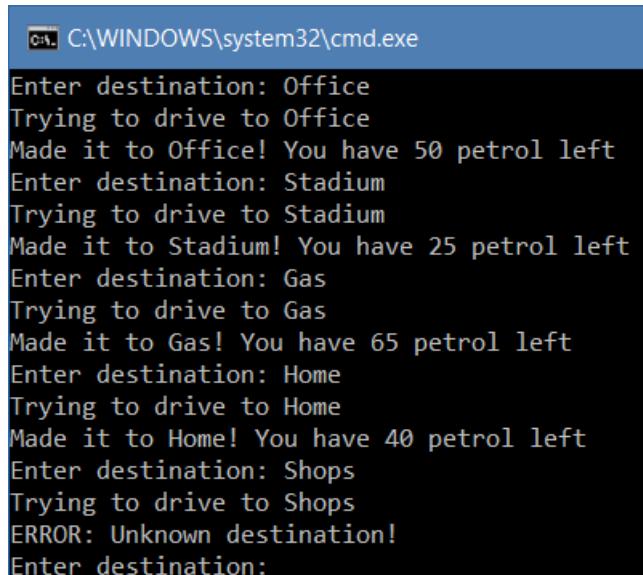
Observe that everything we've written so far is entirely *pure*. There's no shared state, no mutation. We can call these functions easily in isolation, and build them up into larger functions.

## 8.6 Moving to a full application

Now, once you've tested this all out, you should be ready to move this code into a full application.

1. In the solution, open up `Car.fs`.
2. Copy across the two helpers you wrote, `getDistance` and `calculateRemainingPetrol`.
3. Make sure that you paste them *above* the stub function for `driveTo`.
4. Lastly, copy across the implementation for `driveTo` from your script into the file, replacing the existing stub implementation.

You should now be in a position to compile the application and run it: -



```
C:\WINDOWS\system32\cmd.exe
Enter destination: Office
Trying to drive to Office
Made it to Office! You have 50 petrol left
Enter destination: Stadium
Trying to drive to Stadium
Made it to Stadium! You have 25 petrol left
Enter destination: Gas
Trying to drive to Gas
Made it to Gas! You have 65 petrol left
Enter destination: Home
Trying to drive to Home
Made it to Home! You have 40 petrol left
Enter destination: Shops
Trying to drive to Shops
ERROR: Unknown destination!
Enter destination:
```

Figure 8.2 – Sample output from the console runner

## 8.7 Summary

You made it - hopefully that wasn't too taxing! This capstone exercise should have shown you a number of elements from lessons that you've already seen, including: -

- Working with scripts as a way of exploring a domain and developing code
- Writing expression-based, pure functional code
- Migrating from scripts to console applications

# Unit 3

## *Types and Functions*

So far, in terms of F# the language, we've looked at type inference, structuring code with expressions, and working with immutable data. But we've only dealt with simple data values – ints, strings and so on. What about classes? Do we still have them in F#? If not, what else do we use? And what about functions – F# is supposed to be a functional programming language, and yet we've barely covered them!

Don't worry – this unit will deal with all of those topics. We'll see how F#'s approach to separating data and behaviour works, and why it generally means that classes are undesirable within a functional-first system. In fact, we won't be focusing on them at all in this unit – but we *will* look at alternative ways of modelling problems without needing to resort to classes (and believe me when I say I've written entire full-stack applications in F# without the need to write a single class!).

You'll also see how the rules we learned about immutable data still apply, even when working with larger data structures. We'll also learn more about F# functions – how they are much more powerful than the simple methods that we're used to, and how we can often use *functions* instead of *classes*. And, as if that's not enough, we'll also cover how to construct larger applications through namespacing and *modules*.

There'll be more and more F# code in the coming lessons, so make sure you have an open copy of Visual Studio at the ready with a blank .fsx file so that you can code as you go!

**NOTE** Pay attention in this unit – after it, there'll be a slightly larger exercise for you to work through that's designed to get you using *all* of the elements you'll have learned thus far in a single, coherent application. Think of it as the end-of-level boss, and these lessons as the power ups you need to gain in order to beat it!

## 9

## *Shaping data with Tuples*

We'll start this unit by looking at the simplest data structure in F#, the Tuple. Tuples are a great way to quickly pass small bits of data around your code when classes or similar feel like overkill.

In this lesson: -

- You'll see how tuples are used within F#
- You'll understand when to use and not use it
- You'll see how tuples work together with type inference to enable succinct code
- You'll see how tuples relate to the rest of .NET

Let's start by considering an example that seems trivial and yet gets us in all sorts of contortions nearly every day. The following method takes in a string that contains an individual's name e.g. "Isaac Abraham" and splits it out into its constituent parts, returning **both** the forename and surname.

### **Listing 9.1 Returning arbitrary data pairs in C#**

```
public ??? ParseName(string name) {
 var parts = name.Split(' ');
 var forename = parts[0];
 var surname = parts[1];
 return ???; }
```

What should this method return? There are a few options, none of which are particularly satisfying (I should point out that these are all real answers that have been suggested to me when I've posed this question!): -

1. Create a dedicated DTO called Name, with properties Forename and Surname. This works, but it's a pretty heavyweight approach for something as small as this one-off function. And, what if we have a second method that returns forename, surname and age? Very quickly you'll end up with many different DTOs, all of which are very similar,

and probably have to map between them since C# doesn't allow you to compare objects that have the same *structure* rather than *type*.

2. Return an anonymous type? Unfortunately, C# doesn't allow anonymous types to escape method scope – so instead we can return it as a weakly-typed Object, and then use reflection or similar to get at the data. Obviously doing something like this effectively removes us from the world of the C# type system and into the world of runtime checking. Also, anonymous types are internal, so this solution doesn't work across assemblies.
3. The same as point 2, but this time use dynamic to avoid reflection.
4. Return an array of strings. Again, this isn't ideal and means that the type system isn't working for us. If we want to return a mixture of types, we're again stuck.
5. Out parameters. Everyone hates these! You need to first explicitly declare a variable before calling a method, and the syntax is somewhat ugly. There's actually work going on in the C# team to try to improve this in the future for C#.

The one option that I rarely hear is to return a *Tuple*. Tuples are exactly what we need in this case, but C# currently has no specific *language* support for them - so you have to rely on the raw BCL type, System.Tuple. Here's what it looks like (I've used explicit typings here for documentation purposes, but it's not necessary – I would normally use var): -

### **Listing 9.2 Returning arbitrary data pairs in C#**

```
public Tuple<string, string> ParseName(string name) {
 string[] parts = name.Split(' ');
 string forename = parts[0];
 string surname = parts[1];
 return Tuple.Create(forename, surname); }

Tuple<string, string> name = ParseName("Isaac Abraham"); ①
string forename = name.Item1; ②
string surname = name.Item2;
```

① : Calling a method that returns a Tuple of string, string

② : Manually deconstructing the Tuple into meaningful variables.

Tuples are nice in that they allow us to pass arbitrary bits of data around, temporarily grouped together. Tuples also support equality comparison by default, so we can compare arbitrary tuples against one another (provided their generic types are the same and each type itself supports equality comparison). The problem is that the properties show up as Item1, Item2, ItemN etc. – so we lose all semantic meaning. We usually either comment them to explain meaning, or immediately “deconstruct” the tuple into its constituent parts with meaningful variable names.

## 9.1 Tuples Basics

F# on the other hand has *language* support for tuples. This means that we can rewrite the code above as follows: -

### **Listing 9.3 Returning arbitrary data pairs in F#**

```
let parseName(name:string) =
 let parts = name.Split(' ')
 let forename = parts.[0]
 let surname = parts.[1]
 forename, surname ①
let name = parseName("Isaac Abraham") ②
let forename, surname = name ③
let fname, sname = parseName("Isaac Abraham")
```

① : Creating a tuple of forename and surname

② : Calling a function that returns a Tuple

③ : Deconstructing a tuple into meaningful values

④ : Deconstructing a tuple directly from a function call

Let's look at this in a little more detail. Most of this should translate quite easily from the C# example that we saw, but the key parts of how we interact with tuples will be new.

Firstly, instead of having to explicitly call the `Tuple.Create` function, we can create tuples simply by separating values with a comma. Secondly, we can also *deconstruct* a tuple back into separate parts by assigning them to different values, again with a comma. Tuples can also be of arbitrary length and contain a mixture of types – so to create a tuple of three values, we would use syntax such as `let a = "isaac", "abraham", 35`.

### **Tuples and decimal numbers**

Some countries (mostly those in mainland Europe) use commas to express decimals e.g. 10,5 rather than 10.5 – if you're living in one of those countries, don't get confused here! In F#, you don't need the space separator, so `let y = 10,5` is a tuple of two numbers. It's **not** a decimal separator.

### **Now you try**

Let's now do a bit of hands-on work with Tuples ourselves.

1. Open a blank fsx file for experimenting.
2. Create a new function, `parse`, which takes in a string `person` which has the format "playername game score" e.g. "Mary Asteroids 2500".
3. Split the string to split the different parts of the string into separate values.
4. Convert the third element to an integer. You can either use `System.Convert.ToInt32()`, `System.Int32.Parse()` or the F# alias function for it, `int()`.
5. Return back a three-part tuple of name, game and score and assign to a value.

6. *Deconstruct* all three parts into separate values using `let a,b,c =` syntax.
7. Notice that you can choose arbitrary names for each element.

### The history of Tuples in .NET

Tuples were introduced in the BCL in .NET 4, but they were actually part of the `FSharp.Core` library since much earlier. Indeed, if you run any F#2 code, you see that `FSharp.Core` has a version of the `Tuple` type. Since F#3, this type no longer exists as F# uses the BCL `System.Tuple` instead.

C#7 will also have a similar kind of language-level Tuple support. However, for performance reasons, it's likely that a new `Tuple` type will be created which is a value type (rather than `System.Tuple`, which is a reference type). F# will probably have a new `struct` keyword to allow us to choose which version of `Tuple` we wish to use.

#### 9.1.1 When should I use tuples?

Tuples are a lightweight data structure. They are easy to create, with native language support. As such, they're great for internal helper functions and for storing intermediary state. You can imagine using them within a function as a way to "package up" a few values in order to pass them to another section of code easily – or as a way of specifying intent i.e. that two values are somehow "bound" to one another e.g. first name and surname, sort code and account number etc. etc.

### Tuple Helpers

F# also has two built-in functions for working with two-part tuples – `fst` and `snd`. As the name suggests, they take in a two-part tuple and return either just the first or second element in the tuple.

### Quick Check

1. How would you separate values in a tuple in F#?
2. What is the main distinction between tuples in F# and C# 6?

#### 9.2 More complex tuples

Let's briefly expand upon the above with a slightly more detailed look into tuples and how they fit into the F# type system.

#### 9.2.1 Tuple Type Signatures

It's worth understanding Tuple notation in F#, which is `type * type * type`. So, a three-part tuple of two strings and an int would be notated as `string * string * int`. Here's a simple example of that: -

```
let nameAndAge = "Joe", "Bloggs", 28
 val nameAndAge : string * string * int
```

Figure 9.1 – Creating a three-part tuple in F# and Visual Studio 2015

## 9.2.2 Nested Tuples

We can also nest, or group, tuples together. In the example above, we treat all three elements as siblings; there's no grouping of the name elements. We can fix that by creating a "nested" tuple within a larger tuple by grouping together the "inner" part with brackets: -

### **Listing 9.4 Returning arbitrary data pairs in F#**

```
let nameAndAge = ("Joe", "Bloggs"), 28 ①
let name, age = nameAndAge ②
let (forename, surname), theAge = nameAndAge ③
```

- ① Creating a nested tuple
- ② Deconstructing a tuple
- ③ Deconstructing the same tuple, including the nested component

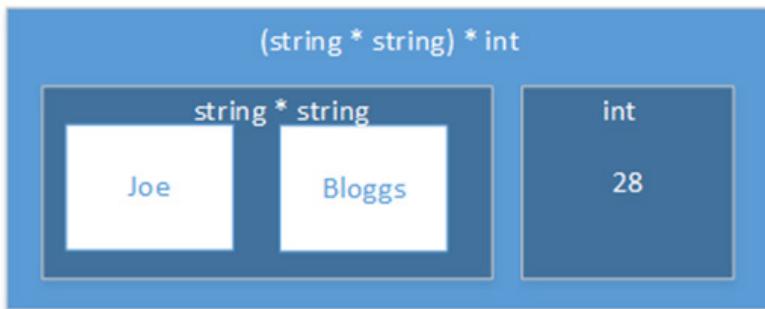


Figure 9.2 - A tuple containing a nested tuple

Just to confirm – the type signature for `nameAndAge` here is `(string * string) * int`. In other words, a two-part tuple, the first part of which is *itself* a tuple of two strings, whilst the second part is an int. If you were to use the raw `System.Tuple` type here (and you can prove it by calling `GetType()` on `nameAndAge`), it would look something like `Tuple<Tuple<String, String>, Int32>`. I know which syntax I prefer!

## 9.2.3 Wildcards

If there are elements of a tuple that you're not interested in, you can discard them whilst deconstructing a tuple by assigning those parts to underscore symbol: -

**Listing 9.5 Using wildcards with Tuples**

```
let nameAndAge = "Jane", "Smith", 25
let forename, surname, _ = nameAndAge ①
```

- ① Discarding the third element of the tuple

This is particularly useful when you only want to pull out a certain section of a tuple, and is better than simply assigning to arbitrary values such as `x` or `y` – the underscore is an actual symbol in F# and tells the type system (and the developer) that we explicitly do not want to use this value. We'll also see in the coming lessons how wildcards are useful when *pattern matching* – a form of conditional logic checking which replaces `switch / case`.

**9.2.4 Type Inference with Tuples**

F# can infer tuples, just like it does with “simple” value – even for function arguments: -

**Listing 9.6 Type Inference with Tuples in F#**

```
let explicit : int * int = 10, 5 ①
let implicit = 10,5 ②

let addNumbers arguments = ③
 let a, b = arguments
 a + b
```

- ① : Explicit type signature  
 ② : Type inferred to be `int * int`.  
 ③ : “arguments” inferred to be `int * int`.

Notice also that F# will also automatically genericise tuples within functions if a tuple element is unused within a function: -

**Listing 9.7 Genericised functions with tuples**

```
let addNumbers arguments =
 let a, b, c, _ = arguments ①
 a + b
```

- ① Deconstructing a four-part tuple

What is the signature of `arguments` in this case? The compiler can infer the types of `a` and `b` as integers, but there are two other elements in the tuple – `c` and the wildcard value. So, the compiler will automatically make them generic type arguments to the `addNumbers` function: `int * int * 'a * 'b`. In other words, this is a four-part tuple, of which the first two elements are integers, the third is of type `'a` and the fourth of type `'b`.

**Quick Check**

3. What is the type signature of `nameAndAge` in the listing 9.4? Why?

4. How many elements are there in `nameAndAge`?
5. What is the purpose of the wildcard symbol?
6. How many wildcards can you use when deconstructing a tuple?

## 9.3 Tuple best practices

So, we've seen what Tuples can do – but F# has other data structures, such as Records (Lesson 9) and Discriminated Unions. So, this section will briefly outline some best practices for using Tuples to allow you understand how to best utilise them.

### 9.3.1 Tuples and the BCL

One of the nicest parts of F# is that it handles interoperability between F# and C# / VB. There's plenty to see on this later on in the book, but as an example, we'll see now how F# uses tuple language support to elegantly remove the need for `out` parameters. Here's an example of trying to parse a number stored in a string using C# and the `Int32.TryParse` function: -

#### **Listing 9.8 Implicit mapping of out parameters to Tuples**

```
var number = "123";
var result = 0; ①
var parsed = Int32.TryParse(number, out result); ②
let result, parsed = Int32.TryParse(number); ③
```

- ① : Declaring the “out” result variable with a default value.
- ② : Trying to parse number in C#
- ③ : Replacing out parameters with a tuple in a single call in F#

Here, the same BCL call that we would normally have to call as a two-stage process with an `out` parameter is replaced with a single call and both the parsed value *and* parsing result are returned as a tuple – much nicer!

### 9.3.2 When not to use Tuples

F# has other types of data structures in addition to Tuples, so clearly Tuples aren't the only way of passing data around. So when wouldn't you use them?

Despite their ease-of-use, Tuples are not generally a great fit for public APIs *except* where the tuple size is small – typically two or at most three elements wide; anything more quickly becomes difficult to reason about. Why is this? Remember that tuple fields have *no specific names* – the *client* of the tuple can choose arbitrary names for tuple fields when deconstructing them – so it's not easily possible to encode the semantic meaning behind a tuple. Imagine you have a function that returns a tuple of `string * string`. What does this represent? Forename and Surname? City and Country? It could be anything.

So, whilst tuples are very useful (as we'll see, they can simplify complicated type signatures), you'll still need to think a little about e.g. naming functions that return tuples

intelligently so that it's obvious what the parts of the tuple represent. In the following example, all three functions return the same data: a `string * string` tuple. But what do `a` and `b` represent?

#### **Listing 9.9 Intelligently naming functions**

```
let a, b = getData() ①
let a, b = getBankDetails() ②
let a, b = getSortCodeAndAccountNumber() ③
```

- ① : Poor naming
- ② : Improved naming
- ③ : Better naming

In situations like this, where you have a clear contract for a DTO that's fairly stable, you'll probably prefer to use a Record, which we'll cover in the next lesson.

#### **Quick Check**

7. What's generally considered the maximum size you should use for a tuple?
8. When should you be cautious of using tuples?

## **9.4 Summary**

In this lesson: -

- You saw the most basic data structure in F#, the Tuple.
- You understand that Tuple support in F# is simply language support over the `System.Tuple` type that exists in the BCL.
- You've seen how type inference works with Tuples.
- You understand when and where Tuples are best suited to being used.

#### **Try This**

Look at methods in common BCL namespaces; try to find some that you think should be "tupled" (hint – ones with `out` parameters are a good start!). Then, write a function to load a filename and last modified date from the file system, using a tuple as the return type.

#### **Quick Check Answers**

1. You use the comma to separate values.
2. F# has language support for tuples; C#6 does not.
3. `(string * string) * int` i.e. a tuple containing firstly a nested tuple of 2 strings, and then an int.
4. 2 elements – `(string * string)` and `(int)`
5. To explicitly discard unneeded elements of a tuple.
6. There is no limit – you can discard as many elements of a tuple as you wish.

7. Two or three elements.
8. In public contracts, particularly where different elements of the tuple are of the same type and open to misinterpretation.

# 10

## *Shaping data with Records*

In the previous lesson, we looked at a lightweight, simple way of packaging data pieces together with the Tuple. We also saw that Tuples are great in some situations, but in others not so much. Now we'll look at F#'s secondary data structure, the Record – a more fully-featured data structure more akin to a Class. In this lesson: -

- You'll see what Records are within F#
- You'll understand how Records compare with C# / VB classes.
- You'll learn how to affect changes to records whilst still retaining immutability
- You'll see some tips for when working with records.

Let's start by continuing where we left off in the previous lesson by describing a situation where Tuples aren't suitable for exposing data – for example, a public contract of some sort where we want explicit named fields, or somewhere that you need to expose more than just two or three properties. Here's a simple example of a Customer type in C#: -

### **Listing 10.1 A basic DTO in C#**

```
public class Customer { ①
 public string Forename { get; set; } ②
 public string Surname { get; set; }
 public int Age { get; set; }
 public Address Address { get; set; }
 public string EmailAddress { get; set; }
}
```

- ① Type definition  
 ② Public, mutable properties

This is often referred to as a POCO (Plain Old C# Object) or DTO (Data Transfer Object) – essentially a class that's used for the purposes of storing and transferring data, but not

necessarily any behaviour. I've omitted the Address class for brevity, but it's just another DTO.

That POCO is pretty nice - but there *are* several issues with this sort of approach, all around data integrity. Firstly, there's no way to guarantee that you'll always create a valid object - for example, you might forget to set the Address property. You can also modify this after the object is created; in fact, anyone could! You'll probably want to enforce construction and lifetime of Customers in a safer way: -

### **Listing 10.2 “Near-immutable” DTOs in C#**

```
public class Customer {
 public string Forename { get; private set; } ①
 public string Surname { get; private set; }
 public int Age { get; private set; }
 public Address Address { get; private set; }
 public string EmailAddress { get; private set; }

 public Customer(string forename, string surname, int age, Address address, string
 emailAddress) { ②
 Forename = forename;
 Surname = surname;
 Age = age;
 Address = address;
 EmailAddress = emailAddress;
 }
}
```

- ① Public read-only, private mutable properties.
- ② Non-default constructor guarantees safe initialisation of object

### **Trusting the compiler - again**

You may think to yourself “I never forget to set all the properties when creating this class”, or “it only happens in one place”. You’ll be surprised how often mistakes actually do happen – particularly once you start sharing a DTO across multiple parts of a system. It’s far better to let the compiler enforce these sorts of rules on us – the problem is that it’s a pain to write all the boilerplate in C#. That’s why tools such as Resharper or CodeRush are so common – they essentially do a lot of this for us. As we’re starting to see, in F# many of these things are baked directly into the language and compiler.

This is a definite improvement, but there's a lot of boilerplate here! Even worse, we're still not guaranteeing that this type is immutable – the class itself could change its own state later on in e.g. a method. If we wanted to **truly** make this DTO immutable, we'd have to manually create a readonly backing field, and then make a public getter for it. In the interest of time and space (and our sanity), I'm not going to show out that version here.

There's another issue with this that we tend to once again simply take for granted as “the way things are”. Let's say that we want to check if two customers have the same address. How would we check that? Here's what we'd *like* to be able to do: -

**Listing 10.3 Comparing objects of the same type in C#**

```
public class Address { ①
 public string Street { get; set; }
 public string Town { get; set; }
 public string City { get; set; }
}
var sameAddress = (customerA.Address == customer.Address); ②
```

- ① Example Address type  
 ② Comparing two address objects

Unfortunately, this will almost certainly return false, even if both addresses contain the same address values. That's because .NET classes perform reference equality checks by default. In other words, only if both addresses are *the same object*, existing in *the same space in memory*, will this check return true. That's not what we want! What we are looking for is a form a *structural equality* checking. You *can* do this in C# or VB .NET, but as it turns out, it's rather a lot of work – you need to: -

- Override `GetHashCode()`
- Override `Equals()`
- Write a custom `==` operator (otherwise `Equals` and `==` will give different behaviour!)
- Ideally implement `System.IEquatable`
- Ideally implement `System.Collections.Generic.IEqualityComparer`

Try implementing all of this by hand for all fields in the class (and the associated unit tests) and you'll discover that suddenly our POCO is no longer a "plain" C# object but a COCO – a Complex Old C# Object!

You'd be surprised how often we actually want some form of structural equality rather than referential equality – and there's a reason that we use tools like Resharper to generate this for us – it's not fun, and prone to errors. Of course, even with a tool like Resharper, the code that is generated needs to be maintained – imagine that you add a property to an object and forget to regenerate the equality checking code. This can lead to the worst kinds of bugs that only occur at runtime in specific circumstances depending on the objects being compared.

## 10.1 POCOs done right - Records in F#

F# Records are best described as simple-to-use objects designed to store, transfer and access immutable data that have named fields – essentially the same thing we've just tried to achieve with a C# POCO.

### 10.1.1 Record Basics

Let's try to implement the same Address type in F# so that it supports both immutability as well as implementing structural equality checking: -

**Listing 10.4 Immutable and Structural Equality record in F#**

```
type Address =
 { Street : string
 Town : string
 City : string }
```

Believe it or not, that's everything you need. For this, you'll get: -

- A constructor that requires all fields to be provided
- Public access for all fields (which are themselves read-only)
- Full structural equality, throughout the *entire object graph*

**Declaring records on a single line**

You can also define the record on a single line (useful for very simple records) using a semicolon as separator e.g. `type Address = { Line1 : string; Line2 : string }.` Note that for multiline declarations, just like the rest of the language, you need to ensure that all fields start on the same column.

I suggest you allow Visual F# Power Tools take care of formatting for you though – at least whilst you're taking your first steps with the language. You can highlight any declaration and choose **Edit -> Advanced -> FormatSelection** (or **FormatDocument**) to have it format your code in a consistent manner.

**10.1.2 Creating Records**

Create records in F# is super easy – record constructors again have specific language support. Let's create an instance of an Address, and then create a Customer with that address (reuse the code sample from earlier for the declaration of the Address type): -

**Listing 10.5 Constructing a nested record in F#**

```
type Customer = ①
 { Forename : string
 Surname : string
 Age : int
 Address : Address
 EmailAddress : string }
let customer = ②
 { Forename = "Joe"
 Surname = "Bloggs"
 Age = 30
 Address =
 { Street = "The Street"
 Town = "The Town"
 City = "The City" }
 EmailAddress = "joe@bloggs.com" }
```

- ① Declaring the Customer record type
- ② Creating a Customer with Address inline.

Easy! Notice how I've defined the address *inline* whilst creating the customer – of course, I could have also defined that separately if I'd wanted to as a separate let binding. Now that you've created a record, you can start to access fields on the record just like normal C# objects.

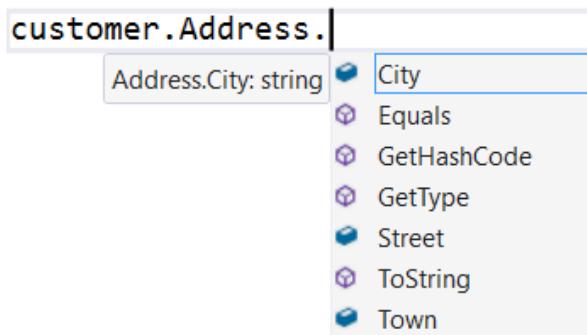


Figure 10.1: Accessing fields on an F# record.

You'll notice that, just like with a constructor that requires all properties, you cannot "miss out" any of the fields when declaring an instance of a record: -

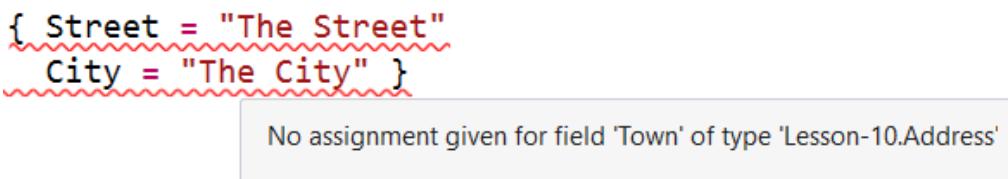


Figure 10.2: Compiler error when omitting the Town field from an Address record.

### All your fields belong to us

One nice thing about having to eagerly set all fields of a record is that when you decide to add a new field to a record, the compiler will instantly warn you of every location where you create an instance of that record, so that you can ensure that you never accidentally create a record with half of it uninitialized. You might wonder how you would deal with cases when we only have *some* values of a record up-front – or perhaps you only want to use some of the fields in the record some of the time, and other times use all of them. In C#, it's quite normal to "re-use" a POCO for multiple purposes by simply omitting setting some fields.

This is again part of F# trying to guide us down the road of being explicit about this and ultimately encoding these sorts of business rules or situations *within the type system*. We'll cover the most common answer in a later lesson (Discriminated Unions), but the main point is that the compiler forces you to populate all fields when creating a record value.

### Now you try

Let's have a look at creating our own record type now.

1. Define a record type in F# to store data on a Car, such as manufacturer, engine size, number of doors etc.
2. Create an instance of that record.
3. Experiment with formatting of the record – use Power Tools to automatically format the record for you etc.

### Quick Check

1. What is the default accessibility modifier for fields on records?
2. What is the difference between referential and structural equality?

## 10.2 Doing more with Records

### 10.2.1 Type Inference with Records

You'll notice that the code used to create an instance of a record (such as `address` and `customer`) looks somewhat similar to how you declare objects in javascript – a dynamic language. Don't be fooled! The compiler knows that these are static types, rather than a dynamic object. It's that once again the compiler has inferred the types based on the *properties that have been assigned to the object*. Of course, you can be explicit about this in a couple of ways – either by specifying the type of the left-hand side binding, or prefixing fields with the type name: -

#### **Listing 10.6 Providing explicit types for constructing records.**

```
let address : Address = ①
 { Street = "The Street"
 Town = "The Town"
 City = "The City" }

let addressExplicit =
 { Address.Street = "The Street" ②
 Town = "The Town"
 City = "The City" }
```

- ① : Explicitly declaring the type of the address value.  
 ② : Explicitly declaring the type that the Street field belongs to.

I would encourage you to avoid using explicit types unless you really need to, but one benefit of choosing to prefix a field with the type is that the compiler will also give you some intellisense immediately: -

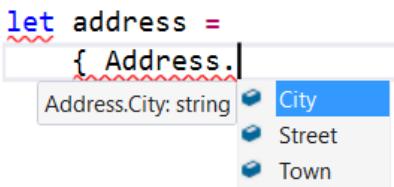


Figure 10.3: Creating a record in F# with field-level intellisense.

F# will also infer a record type based on *usage* of an instance. Here's an example of a function that takes in a customer as an argument – once you've "dotted into" the object once and accessed the first field, the compiler will kick in and realise what you're doing: -

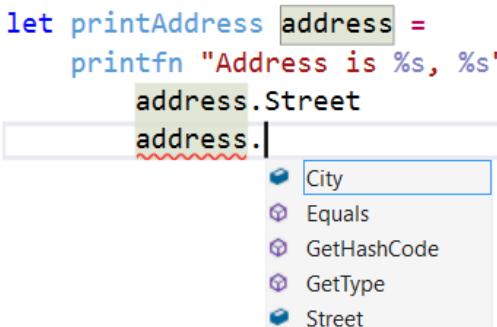


Figure 10.4: Type inference correctly identifies the type of the address object.

Note that until you access at least one property, the compiler (obviously) won't be able to deduce the type – so it'll normally just show as an object (or perhaps a generic type).

### Members on Records

You'll notice that Records have the standard `ToString()`, `GetHashCode()` etc. members – that's because Records actually compile down to Classes. And, just like classes, you *can* create member methods on them. However, this is not something that you'll normally need to do in F# – remember that Records should act as DTOs – and won't be covered in this lesson.

## 10.2.2 Working with immutable Records

Just like regular value bindings, fields on F# records are *immutable* by default. But we know that in the real world, we sometimes have to model the context of a value changing its state over time. How can we do this without mutating fields on a record? F#'s answer to this is to

provide what is known as “copy-and-update” syntax. Let’s try to change a customer’s email address and their age: -

#### **Listing 10.7 Copy-and-update record syntax**

```
let updatedCustomer = { customer with Age = 31; EmailAddress = "joe@bloggs.co.uk" } ①
```

- ① Creating a new version of a record using the ‘with’ keyword.

The idea behind this is: you provide a record **with** the *modifications* that you wish to perform on the record - F# will then create a *copy* of the record with those changes applied. So here we’ve provided `customer` with *two modifications*. In this way, you can get the best of all worlds: -

- You can provide records to other sections of code without having to worry about their values being implicitly modified without your knowledge.
- You can still easily “simulate” mutation through copy-and-update.
- If you want to write a function that does modify a record, you simply have it take in the original version as an argument and return the new version as the output of the function. This is exactly how LINQ works with collections (and it should not come as a surprise to learn that the LINQ featureset is basically a subset of functional programming for C#)

This way of working is a great fit for event-based architectures, where you record all changes to data over time as immutable events and versions of records.

If you *absolutely have to* – and this really should be the exception to the rule – you *can* override immutability behaviour on a field by field basis by adding the `mutable` modifier. The sort of situation that you might want to do this on is if you have a record which will be used in a tight loop, mutating itself thousands of times a second. Because records are reference types (although it’s looking increasingly like the next version of F# will allow struct records), every copy-and-update causes a new object to be allocated on the heap, so GC pressure could cause performance issues in such a situation. However, I would recommend that the default should be to use immutable data structures initially, test performance, and only if you see an issue, reconfigure the definition of the record. Certainly in all applications I’ve written this has never been an issue for me – bottlenecks are far more likely to occur with other parts of your application such as e.g. database connectivity etc.

#### **10.2.3 Equality Checking**

You can safely compare two F# records of the same type with a single `=` for full, deep structural equality checking: -

#### **Listing 10.8 Comparing two records in F#**

```
let isSameAddress = (address = addressExplicit) ①
```

**1 Comparing two records using the '=' operator.**

You *can* override this behaviour with a few attributes that you can place on a record, but I'd advise you to avoid looking them up unless you really need to as you then need to fall back to implementing `GetHashCode()` manually etc. – but it's worth knowing that there *is* an escape hatch if needed.

### Now you try

Let's practically explore some of these features of Records.

1. Define a record type, such as the `Address` type shown earlier.
2. Create two different instances of the record that have the same values.
3. Compare the two objects using `=`, `.Equals`, and `System.Object.ReferenceEquals`.
4. What are the results of all of them? Why?
5. Create a function that takes in a customer and, using copy-and-update syntax, sets their `Age` to a random number between 18 and 45.
6. The function should then print out the customer's original and new age, before returning the updated customer record.

Let's now wrap up this section by comparing classes and records.

**Table 10.1 - Comparing Classes and Records**

|                            | NET Classes        | F# Records            |
|----------------------------|--------------------|-----------------------|
| Default mutability of data | Mutable            | Immutable             |
| Default Equality behaviour | Reference equality | Structural equality   |
| Copy and update syntax?    | No                 | Rich language support |
| F# type inference support? | Limited            | Full                  |
| Guaranteed initialisation  | No                 | Yes                   |

### Quick Check

3. At runtime, what do Records compile into?
4. What is the default type of equality checking for Records?

## 10.3 Tips and tricks with Records

Let's now briefly discuss a few extra tips on working with records.

### 10.3.1 Refactorings

Don't forget that VFPT has support for a number of useful refactoring tools to make life even easier. One such feature is rename refactoring on record fields (**F2**). Another is to automatically populate all fields in a record when creating one which you can then fill in: -

1. Start creating an instance of a record.
2. Set at least one field on the record.
3. Move the caret to the start of the field declaration and wait for the lightbulb to appear.
4. Hit **CTRL + .** and choose **Generate record stubs** from the pop-up menu.

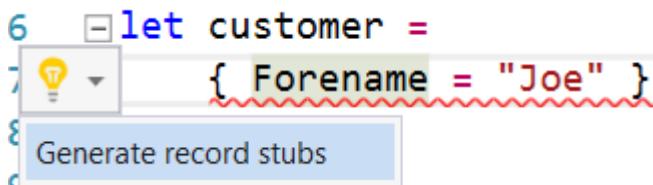


Figure 10.5: Automatically generating record stubs through VFPT.

You can configure VFPT on how to fill in missing fields through **Tools / Options / F# Power Tools / Code Generation** : -

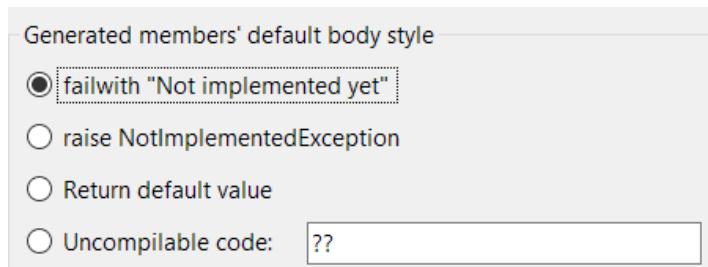


Figure 10.6: Configuring how VFPT auto-generates record stubs

### 10.3.2 Shadowing

Copy-and-update is a common feature to use in F# when working with records, but it doesn't necessarily feel right to create different names for value bindings every time we update a record. That's why F# allows us to *reuse* existing named bindings, called *shadowing*. This means that we can write code as follows: -

```
let myHome = { Street = "The Street"; Town = "The Town"; City = "The City" }
let myHome = { address with City = "The Other City" }
let myHome = { address with City = "The Third City" }
```

Notice here that we have simply reused the binding `myHome` rather than `myHome1`, `myHome2` and `myHome3` etc. etc. This is *not* the same as mutating `myHome` – instead, we’re reusing the *same symbol* with a *new value*. You can observe this in VS by highlighting a symbol and witnessing how the editor highlights references to the *instance* of the symbol as shown in Figure 10.6.

```
let myHome = { Street = "Th"
let myHome = { address with
printfn "%s" myHome.City
let myHome = { address with
printfn "%s" myHome.Street
()
```

Figure 10.7 – Visual Studio highlighting referencing to a specific instance of a symbol.

### 10.3.3 When to use records

Records are probably the most common type of data structure in F#. They’re more powerful than tuples, with the ability to explicitly name fields as well as a neat copy-and-update syntax that Tuples do not have. For a C# / VB .NET developer, they’re a natural fit for all the times we’ve needed simple DTOs within our applications. Lastly, as Records compile down into classes, they can be consumed very easily in other .NET languages and systems expecting classes – because that’s all that they really are; it’s just that F# wraps over them with an extremely smart compiler.

However, Tuples still have their place within F#, particularly when you’re working with small bits of short-lived data, and especially if they’re used in a context where the data is not exposed publicly.

#### Quick Check

5. What is shadowing?
6. When should you use records?

## 10.4 Summary

In this lesson: -

- You saw the most common data structure in F#, the Record.
- You understand the typical use cases that F# records are designed to solve.
- You’ve seen how type inference works with Records.
- You’ve learned about a powerful alternative to working with mutable data called copy-and-update.

**Try This**

1. Try to model the Car example from Lesson 6, but using Records to model the state of the Car.
2. Take an existing set of classes that you have in an existing C# project and map as Records in F#. Are there any cases that don't map well?

**Quick Check Answers**

1. Public
2. Referential equality compares that two records are actually the same object in memory; structural equality compares the contents of two records.
3. Classes
4. Structural Equality
5. The ability to re-use an existing symbol for a new value.
6. For public contracts, typically where Tuples are not a good fit due to number of fields.

# 11

## *Building composable functions*

**It seems a little strange to consider that we're on lesson eleven and haven't spent much time talking about functions yet! However, you've already seen (and built your own) functions by now, so you've gained a little exposure to them already. In fact, you can already do pretty much the same sort of things you'd do with methods in C# / VB .NET. What we'll do in this lesson is dig into a bit more depth as to just how powerful functions in F# really are: -**

- You'll gain a proper understanding of F# functions compared to methods
- You'll learn about a powerful technique called partial application
- You'll learn about two important operators in F# that help build larger pieces of code - Pipeline and Compose

Have an energy drink before you start this lesson, as it's probably the one lesson in this book that will throw the most at you in terms of F# features and syntax!

We tend to think of functions and methods as interchangeable terms. However, "let bound" functions in F# are actually an entirely different beast to methods. By "let bound", I'm referring to the sorts of function that you've already been defining so far – bound to a value through the `let` keyword (you *can* create classes with methods in F#, although I've not shown the syntax for this yet). Let's see a quick comparison between *methods* and *functions*: -

**Table 11.1 – Comparing Methods and Functions**

|             | C# Methods                         | F# let-bound functions                   |
|-------------|------------------------------------|------------------------------------------|
| Behaviour   | Statements or expressions          | Expressions by default                   |
| Scope       | Instance (object) or Static (type) | Static (module level or nested function) |
| Overloading | Allowed                            | Not supported                            |
| Currying    | Not supported                      | Native support                           |

Some of these points above we've already covered in earlier lessons (for example, functions always return something, even if that something is the "unit" object), but there are some points above that you'll almost certainly be surprised by. Let's start by explaining the one term above that you might not know yet - currying.

## 11.1 Partial function application

Partially applied functions are one of the most powerful parts of the function system in F# compared to C#, and opens up all sorts of interesting possibilities for working with functions. Let's start by clarifying something you'll probably have already noticed from previous examples – the following two functions appear to do the same thing, except that one uses brackets (parentheses) and commas for input arguments (like C#) and one doesn't. The former is referred to as *tupled* form and the latter as *curried* form: -

### **Listing 11.1 Passing arguments with and without brackets**

```
let tupledAdd(a,b) = a + b ①
let answer = tupledAdd (5,10)

let curriedAdd a b = a + b ②
let answer = curriedAdd 5 10
```

- ① : Tupled function int \* int -> int
- ② : Curried function int -> int -> int

Many developers are frightened when they see the explanation of curried functions, and to be honest, you don't really *need* to understand it. To cut a long story short, the main differences to take away from the above sample is this: -

1. *Tupled* functions force you to supply all the arguments at once (just like standard methods), and have a signature of `(type1 * type2 ... * typeN) -> result`. F# actually considers all the arguments as a *single object*, which is why the signature looks like a tuple signature – that's exactly what it is.
2. *Curried* functions allow you to supply only *some* of the arguments to a function, and get back a *new function* that expects the *remaining* arguments. They have a signature of `arg1 -> arg2 ... -> argN -> result`. You can think of these as a function that itself returns a function (please feel free to take a moment to let that sink in).

### **Listing 11.2 Calling a curried function in steps**

```
let add first second = first + second ①
let addFive = add 5 ②
let fifteen = addFive 10 ③
```

- ① Creating a function in curried form
- ② Partially applying add to get back a new function, addFive with signature int -> int
- ③ Calling addFive

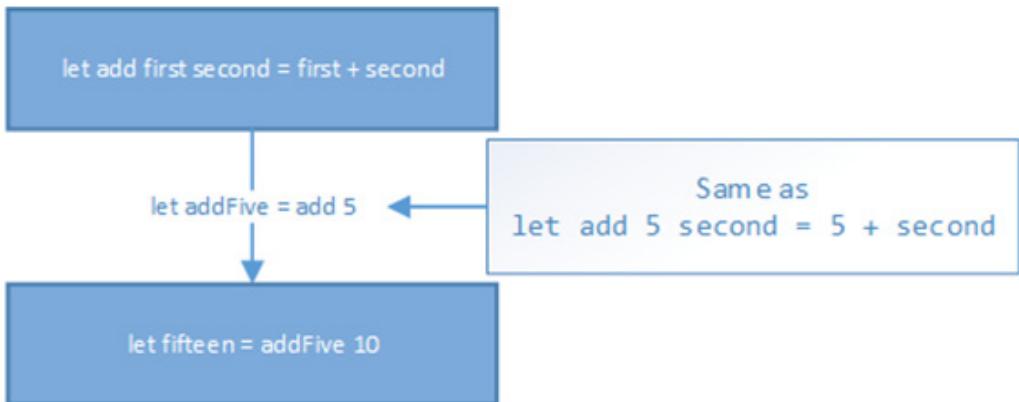


Figure 11.1 – Partially applying a function to create a new function

### Partial application and currying

You might have heard the terms “curried” and “partially applied” functions before. The two are sort of related – a curried function is a function that itself returns a function. Partial application is the act of calling that curried function to get back a new function.

So, now we know *what* curried functions are, let’s have a look into some cases where they offer practical advantages over tupled functions.

#### Quick Check

1. What is the difference between a curried and tupled function?

## 11.2 Constraining functions

One easy use case for curried functions is where you wish to create a more constrained version of a function – sometimes known as a *wrapper* function. You probably use methods like this all the time when you wish to make functions easier to call when e.g. a subset of the arguments is often the same. Let’s look at a simple set of wrapper functions that can create `DateTime` objects. The first one takes in year, month and day; the second just the month and day for this year, and the final one just the day for this year and month: -

#### **Listing 11.3 Explicitly creating wrapper functions F#**

```

open System
let buildDt year month day = DateTime(year, month, day)
let buildDtThisYear month day = buildDt DateTime.UtcNow.Year month day
let buildDtThisMonth day = buildDtThisYear DateTime.UtcNow.Month day

```

Notice how each function “cascades up” to a more generalised version. This is nice, but with curried functions we can make the two wrapper functions much more lightweight: -

#### **Listing 11.4 Creating wrapper functions by currying**

```
let buildDtThisYear = buildDt DateTime.UtcNow.Year
let buildDtThisMonth = buildDtThisYear DateTime.UtcNow.Month
```

This code is identical to Listing 11.2, except here we didn’t have to explicitly “pass through” the extra arguments to the right-hand side – F# automatically does that for us. We wouldn’t have been able to do this form of “lightweight” wrapping with a function in tupled form, because you need to pass all values of the tuple together. Of course, Visual Studio will automatically infer that these are functions, and not simple values, as you can see from the following screenshot.

```
let buildDt year month day = DateTime(year, month, day)
let buildDtThisYear = buildDt DateTime.UtcNow.Year
let buildDtThisMonth = buildDtThisYear DateTime.UtcNow.Month
```

Figure 11.2: Syntax highlighting for curried functions in Visual Studio

It’s worth remembering that partially applied functions work from *left to right* i.e. you partially apply arguments starting from the left-hand side, and work your way in. That’s why I’ve placed `year` as the first argument – it’s the most “general” argument and the one that I want to partially apply first.

#### **Now you try**

Let’s create a simple wrapper function, `writeToFile`, for writing some data to a text file.

1. The function should take in three arguments in this specific order: -
  - o `date`: The current date
  - o `filename`: A filename
  - o `text`: The text to write out
2. The function signature should be written in *curried* form i.e. with spaces separating the arguments.
3. The body should create a filename in the form `{date}-{filename}.txt`. Use the `System.IO.File.WriteAllText` function to save the contents of the file.
4. You can either manually construct the path using basic string concatenation, or use the `sprintf` function.
5. You should construct the date part of the filename explicitly using the `ToString` override e.g. `ToString("yyMMdd")`; you’ll need to explicitly annotate the type of `date` as a `System.DateTime`.

If you've done this correctly, your function should have a signature as follows. It's *really* important to look at the signature of functions like this so that you learn to understand what's happening.

```
let writeToFile (date:DateTime) filename text =
 let path = File.WriteAllText(filename, text)
```

**Figure 11.3 – Creating a curried function in F#**

The body of the function should look something like this: -

### **Listing 11.5 Creating your first curried function**

```
open System
open System.IO
let writeToFile (date:DateTime) filename text =
 let path = sprintf "%0-%s.txt" (date.ToString "yyMMdd") filename
 File.WriteAllText(path, text)
```

- ① Using %O automatically calls ToString() on the matching argument

6. You should now be able to create more *constrained* versions of this function: -

### **Listing 11.6 Creating constrained functions**

```
let writeToToday = writeToFile DateTime.UtcNow.Date
let writeToTomorrow = writeToFile (DateTime.UtcNow.Date.AddDays 1.)
let writeToTodayHelloWorld = writeToToday "hello-world"

writeToToday "first-file" "The quick brown fox jumped over the lazy dog"
writeToTomorrow "second-file" "The quick brown fox jumped over the lazy dog"
writeToTodayHelloWorld "The quick brown fox jumped over the lazy dog"
```

- ① Creating a constrained version of the function to print with today's date
- ② Creating a more constrained version to print with a specific filename
- ③ Calling a constrained version to create a file with today's date and "first-file"
- ④ Calling the more constrained version – only the final argument is required

There are many useful applications of curried functions, such as dependency injection at the function level (as we'll see in Lesson 12) but curried functions also work very well in tandem with another of F#'s functional features – pipelines.

### **Quick Check**

2. Name at least two differences between C# methods and F# let-bound functions.

### 11.2.1 Pipelines

Wrapper functions is a nice benefit of curried functions, but it's not the main beneficiary – that's where pipelines come in. Irrespective of the language we're in, we often need to call methods in an ordered fashion, where the output of one method acts as the input to the next. Let's take an example of a simple set of methods that we want to orchestrate together:

- Get the current directory
- Get the creation time of the directory
- Pass that time to a function `checkCreation`, which if the folder is older than 7 days' prints "Old" to the console and otherwise prints "New"

You might write some code that looks like this: -

#### **Listing 11.7 Calling functions arbitrarily**

```
let time =
 let directory = Directory.GetCurrentDirectory() ①
 Directory.GetCreationTime directory ②
checkCreation time
```

- ① Temporary value to store the directory  
 ② Using the temporary value in subsequent method call

This isn't bad, but you have a set of temporary variables that are used to pass data to the next method in the call. And if the chain was bigger, it'd quickly get unwieldy. We could try implicitly chaining these methods together: -

#### **Listing 11.8 Simplistic chaining of functions**

```
checkCreation(
 Directory.GetCreationTime(
 Directory.GetCurrentDirectory())) ①
```

- ① Explicitly nesting method calls

This is less code, and it's now clear that there's a specific relationship between these functions, but the problem is that the order that we *read* the code in is now the opposite to the order of *operation* – that's definitely not what we want! What we want is something that looks like this:

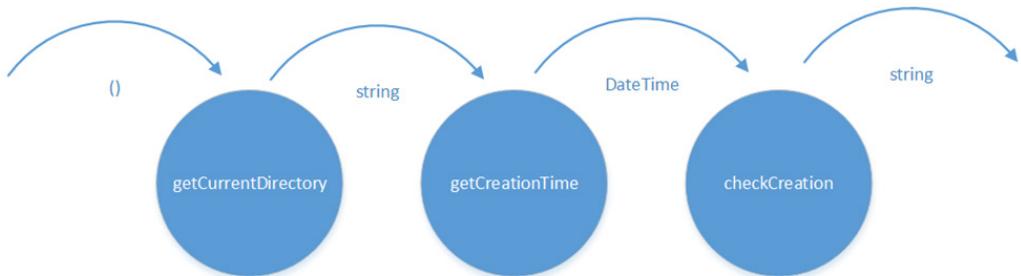


Figure 11.4 – Logical flow of functions

Luckily, F# has a special operator called the *forward pipe* which, much like e.g. currying, is a simple yet powerful feature. It looks like this: `arg |> function`. That doesn't really mean much, so let's explain it another way: -

Take the value on the *left-hand side* of the pipe, and flip it over to the *right-hand side* as the *last argument* to the function.

In other words, given a function call `addFive x`, instead of calling it as `addFive 10`, we can call `10 |> addFive` – so we can simply “flip” the argument over the pipe to the left-hand side. The actual result is the same - it's simply another way of expressing the same code. The beauty of this is that as long as the *output* of one function matches the *input* of the next one, *any function can be chained with another one*. This simple rule means that we can rewrite the above code as follows: -

#### **Listing 11.9 Chaining three functions together using the pipeline operator**

```
Directory.GetCurrentDirectory() ①
|> Directory.GetCreationTime ②
|> checkCreation ③
```

- ① : Returns a string
- ② : Takes in a string, returns a DateTime
- ③ : Takes in a DateTime, prints to the console

Now our code actually reads like it operates! Note that we could have even placed the unit argument for `GetCurrentDirectory` – the `()` object – at the head of the pipeline, but in this case, I've left it in place.

You'll find that the pipeline is extremely useful for composing code together into human-readable *domain specific languages* (or DSLs). And because pipelines operate on the *last argument* of a function, we can quickly create code that looks like this: -

#### **Listing 11.10 Sample F# pipelines and DSLs**

```
let answer = 10 |> add 5 |> timesBy 2 |> add 20 |> add 7 |> timesBy 3 ①
```

```
loadCustomer 17 |> buildReport |> convertTo Format.PDF |> postToQueue ②

let customersWithOverdueOrders =
 getSqlConnection "DevelopmentDb"
 |> createDbConnection
 |> findCustomersWithOrders Status.Outstanding (TimeSpan.FromDays 7.0)
```

① Piped function chain

② An example DSL for working with customer reports as a pipeline

This might look similar to a feature that already exists in C# and VB – *extension methods*. However, they’re not *quite* the same: –

**Table 11.2 – Extension Methods vs Curried Functions**

|                  | C# Extension Methods                                                                                                                      | F#                                                                                                     |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Scope            | Methods must be explicitly designed to be extension methods in a static class with the extension point decorated with the “this” keyword. | Any single-argument .NET method (including the BCL) and all curried functions can be chained together. |
| Extension point  | First argument in method signature                                                                                                        | Last argument in function                                                                              |
| Currying support | None                                                                                                                                      | First class                                                                                            |
| Paradigm         | Not always a natural fit for OO paradigm with private state etc.                                                                          | Natural fit for stateless functions                                                                    |

### CUSTOM FONTS

Although VS uses Consolas by default, you might want to try out the freely-available FiraCode font, which is a monospace font that supports *ligatures* – in other words, it can represent custom operators much more nicely: –

```
customers ▷ where isOver35
customers ▷ where (fun customer -> customer.Age > 35)
```

**Figure 11.5 – Using the Fira Code font in Visual Studio 2015**

Unfortunately, Visual Studio doesn’t support all of the ligatures (unlike VS Code), but most – including the pipeline operator – are rendered correctly, as per Figure 11.6

## Now you try

Let's revisit our simple driving and petrol example from Lesson 6 and see if we can make the code more elegant using pipelines. Recall that the original code looked something like this: -

### **Listing 11.11 Review of existing petrol sample**

```
let startingPetrol = 100.0
let petrol1 = drive(petrol, "far") ①
let petrol2 = drive(petrol1, "medium") ②
let petrol3 = drive(petrol2, "short")
```

- ① – State as a result first function call
- ② – Output of first function call is passed into second function call

However, we now want to consume this code using *pipelines*. Notice that in the previous pipeline example, we were working with functions that only took a *single* argument; this one takes in two arguments – the state (petrol) and the distance travelled. When we want to use pipelining, remember that the *last* argument is the one that gets flipped over to the left-hand side of the pipe.

1. Take the existing petrol function from Listing 6.7.
2. Convert the function from tupled form to curried form i.e. remove commas and brackets.
3. The data that should be piped through should be the last argument – so in this case, the amount of petrol.

Your code should now look like this and can be consumed as follows: -

### **Listing 11.12 Using pipelines to implicitly pass chained state**

```
let drive distance petrol = // code elided... ①

let startPetrol = 100.0 ②

startPetrol
|> drive "far" ③
|> drive "medium"
|> drive "short"
```

- ① drive function rewritten as a curried function with state as the final argument
- ② Starting state
- ③ Implicitly passing state in a chain

When should you use pipelines? I would generally say: whenever you have a piece of data that *logically flows* between a set of functions. That's a bit of a wooly answer, but there's an element to this that simply comes with experience (just like many features in programming languages), as well as personal preference. Some people prefer to pipeline everything – others only if you have at least several functions in the chain. Personally, my recommendation is to

trust your eyes – what “reads better” – `customer |> saveToDatabase or saveToDatabase customer?` There’s no hard and fast rule – it often depends on context – but the more you code, and the more of other peoples’ code that you see, the more confidence you’ll build up to experiment with pipelines yourself.

### Quick Check

3. Which argument to a function is one that can be “flipped” over a pipeline?
4. Can you use C# or VB .NET methods with the pipeline?

## 11.3 Composing functions together

The last element we’ll touch on in this lesson is the somewhat less common operator `>>` (called *compose*), but one that is useful to be aware of. Compose works hand in hand with the pipeline operator and lets us *build a new function* by “plugging” a set of compatible functions together.

Let’s revisit our file processing pipeline from Figure 11.4. If we start thinking about this as a *composed* functional pipeline, we could name the behavior of this entire pipeline something like `checkCurrentDirectoryAge`. In this context, the only elements that are of interest to us are the *initial input* and the *final output* – the rest is effectively intermediate state. As such, we could now rewrite our original pipeline chain from Listing 11.9 as follows: -

### Listing 11.13 Automatically composing functions

```
let checkCurrentDirectoryAge =
 Directory.GetCurrentDirectory
 >> Directory.GetCreationTime
 >> checkCreation ①
let description = checkCurrentDirectoryAge() ②
```

- ① Creating a function by composing a set of functions together
- ② Calling the newly-created composed function

This code essentially does the same as Listing 11.9 – except here you can view this as “plug these three functions together, and give me back a *new function*”. As long as the **result** of the **previous** function is the *same type* as the **input** of the **next** function, you can “plug them” together indefinitely.

```
let checkCurrentDirectoryAge =
 Director
 >> Director
 >> check
```

val checkCurrentDirectoryAge : (unit -> string)

Full name: Lesson-11.checkCurrentDirectoryAge

Figure 11.6: Composing together three functions to create a new function.

As you start out with F#, you'll probably not find yourself using the compose operator a great deal. However, it's worth knowing, and once you get more comfortable with it, will allow you to generate extremely succinct and readable code.

### **Quick Check**

5. What operator do we use for composing two functions together?
6. What rule do you need to adhere to in order to compose two functions together?

## **11.4 Summary**

Another sizeable lesson! Don't worry - for the next couple of lessons, we'll slow down a little bit and let you catch your breath! In this lesson: -

- You learned about the difference between *methods* and *functions* in F#.
- You've seen what the difference between curried and tupled functions are.
- You learned about the *pipeline* operator.
- You learned how F# allows us to natively build larger functions from smaller functions using the *compose* operator.

### **Try This**

Take an existing .NET method in the BCL, or your existing code. Try porting the code to F#, and seeing what impact it has when the function is curried as opposed to tupled. Then, try looking through an existing project where you are composing methods together manually by calling one method and immediately supplying the result to the next method. Try to create a composed function that does the same thing.

### **Quick Check Answers**

1. A tupled function behaves as per C# functions – all arguments must be supplied. Curried functions allow you to supply a subset of the arguments, and get back a new function that expects the remaining arguments.
2. Functions are always static; methods can be instance level. Functions don't support overloading, but do support currying.
3. The last argument to a curried function.
4. Yes, if they only take in a single argument.
5. The >> operator is used for composition.
6. The output of the first function must be the same type as the input of the second function.

# 12

## *Organising code without classes*

We've so far learned all about relatively "low level" elements of F#: language syntax, Tuples, Records and Functions. We've not yet looked at how to organise larger amounts of code that should logically be grouped together. In this lesson: -

- We'll review namespaces in F#
- We'll cover F# modules – a way to statically group behaviours together in a library
- We'll see how to use both within a standalone application

Organising code elements is often a somewhat tricky effort in the OO world, not just in terms of namespacing, but in terms of *responsibilities*. We often spend a lot of time looking at whether classes obey concepts such as "single responsibility", moving methods from one class to another along with associated state etc. I think that the way things work in F# is much, much simpler. We're not so fussed with classes or inheritance or behaviours and state. Instead, because we're typically using stateless functions operating over immutable data, we can use some alternative sets of rules for organising code. By default, follow these simple rules: -

1. Place related types together in namespaces.
2. Place related stateless functions together in modules.

That's pretty much it for many applications. So, the obvious questions are "what are namespaces in F#?" and "what are modules?" Let's take a look.

### **12.1 Namespaces and Modules**

#### **12.1.1 Namespaces in F#**

Namespaces in F# are essentially identical to those in C# and VB in terms of functionality. We use namespaces to *logically* organise data types, such as records (like a `Customer` type), as well as *modules*. Namespaces can be nested underneath other namespaces in a hierarchy –

again, this should be nothing new for you. Namespaces can also be opened in order to avoid having to fully-qualify types or modules, and you can share namespaces across multiple files. Of course, Visual Studio will provide intellisense for types as you “dot into” namespaces etc., just like in C# or VB .NET.

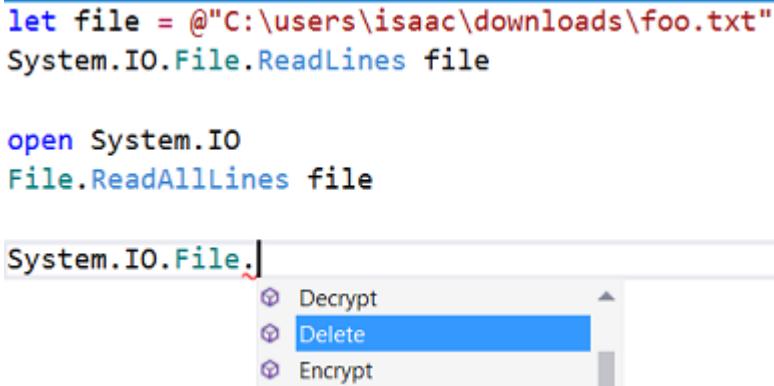


Figure 12.1: Accessing System.IO namespace functionality.

You can observe in Figure 12.1 that we can manually access functions through a fully-qualified namespace, or simply open the namespace after which we can access the static class File directly. Essentially, this is the same as what you’ll already know in C#.

### 12.1.2 Modules in F#

One thing that namespaces cannot hold are functions – just types. We use *modules* in F# to hold let-bound functions. But in F# modules can also be used like namespaces in that they can store types as well. Depending on your point of view, you can think of F# modules in one of two ways: -

- Modules are like *static classes* in C#
- Modules are like namespaces but can also store functions

You can create a module for a file by simply using the `module <my module>` declaration at the top of the file, e.g. `module MyFunctions`. Any types or functions declared underneath this line will live in the `MyFunctions` module.

Just like static classes, modules can live *within* an enclosing namespace (which can be nested). **Important:** In F# you can declare both the namespace and module simultaneously. So `module MyApplication.BusinessLogic.DataAccess` means that we have a module `DataAccess` that resides in the `MyApplication.BusinessLogic` namespace. We don’t have to declare the namespace explicitly first.

### 12.1.3 Visualising namespaces and modules

This diagram may be helpful to visualise the relationship between namespaces and modules in F#.

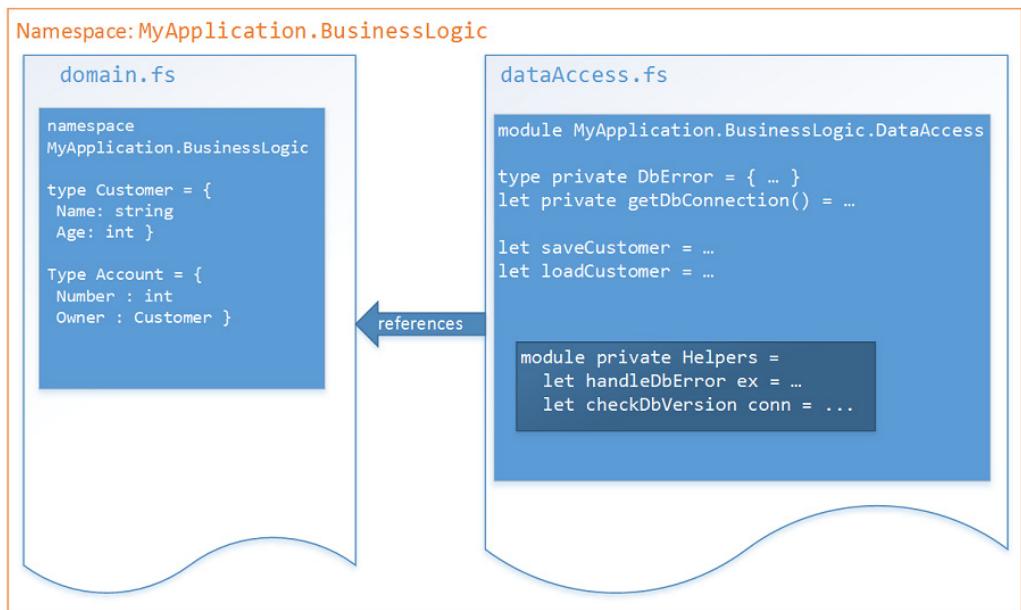


Figure 12.2 – Visualising the typical relationship between namespaces and modules

You can observe several things here. We have two **files** which will be compiled into a single **assembly**, both of which share the same *logical namespace* – `MyApplication.BusinessLogic`. Shared domain types are stored in a single file, `domain.fs`, whilst functionality operating on data access is stored in the `DataAccess` module in the same namespace (which as mentioned earlier, can be declared inline of the module declaration).

In `dataAccess.fs` we don't need to explicitly open `MyApplication.BusinessLogic` to get access to the `Customer` and `Account` types, since the module lives in that namespace anyway, just like you would see with two C# classes living in the same namespace. And, just like with C# classes, you *won't* automatically get access to all types in the *entire* namespace hierarchy – just to types that live in the *same namespace* as the module is declared in.

Also, note the nested module, `Helpers`, which lives inside the `DataAccess` module. It might help to think of this as an inner (nested) static class – you can use nested modules as a way of grouping together functions if you find your modules getting too large. If you're not sure of the full namespace of a module (or function), you can always mouse over it to

understand where it lives. Also, notice the `private` access modifier – see 12.3.1 for more on this.

```
module MyApplication.BusinessLogic.DataAccess
```

```
let loadCustomer =
```

```
()
```

```
val loadCustomer : unit
```

```
Full name: MyApplication.BusinessLogic.DataAccess.loadCustomer
```

Figure 12.3 – `loadCustomer`, a function living in a module that is declared in a namespace.

```
module MyApplication.BusinessLogic.DataAccess
```

```
| module Helpers =
```

```
| let checkDbVersion() = ()
```

```
val checkDbVersion : unit -> unit
```

```
Full name: MyApplication.BusinessLogic.DataAccess.Helpers.checkDbVersion
```

Figure 12.4 – `checkDbVersion`, declared in a nested module.

#### 12.1.4 Opening modules

When I first started using F#, I preferred treating modules as static classes – but nowadays, I find it more natural to treat them as namespaces that happen to also be able to store functions. One of the reasons for this is that modules can be *opened*, just like namespaces.

##### Using static classes

Something similar to this was added to the latest version of C#, whereby static classes can now be added to a `using` declaration. In this way, static classes gain a lot of extra flexibility for making more succinct code. Whilst F#4 can open modules, it currently cannot open static classes, although there is an accepted feature request on the F# language design website (<https://github.com/fsharp/fslang-suggestions>) to add support for this, so hopefully it'll be added in time for the next release.

This is useful when you don't want to continually refer to the module name in order to access types or functions – instead, you can just call the functions directly as though they were defined in the current module: -

**Listing 12.1 Opening modules**

```
open CustomerFunctions ①
let isaac = newCustomer "isaac"
isaac |> activate |> setCity "London" |> generateReport ②
```

- ① Opening the CustomerFunctions module  
 ② Unqualified access to functions from within the module

You'll find opening of modules to be a valuable tool in your arsenal when creating simple, easy to use DSLs – callers can simply open the module with your functions in it, and access the behaviour directly.

**A word on domain specific languages**

Writing DSLs are particularly common in F# due to the syntax of the language e.g. no brackets or braces, pipelining etc. which means that with a few simple functions you can quickly knock up a set of behaviours that are human readable; it's not uncommon to write code that at a top level a business analyst can read and understand. However, you should also be careful not to take DSLs too far with e.g. custom operators – it can be difficult to understand what they're doing, and if taken too far, they can sometimes be difficult to learn how to use (whilst admittedly being extremely powerful).

**12.1.5 Namespaces vs Modules**

Because modules can be `opened`, you might think that they're a complete replacement for namespaces, but they're not. Unlike namespaces, a module *cannot span multiple files*. Nor can you create a module that has the same *fully qualified name* as a *namespace in another file*. For this reason, you should still use namespaces as in C# - to logically group types and modules. Use modules primarily to store functions, and secondly to store types that are tightly related to those functions.

**Quick Check**

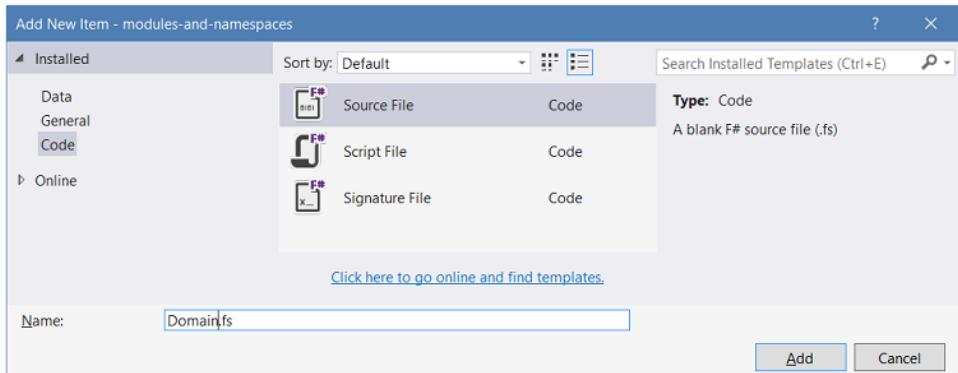
1. Can you store values in namespaces?
2. Can you store types in modules?

**12.2 Moving from scripts to applications****Now you try**

We discussed in lesson 3 the notion of moving from scripts to projects, and did this for a very simple application. Let's try it again now but now involving modules and namespaces, learning some features about them along the way.

1. Reopen the `MyFirstFSharpApp` that you created earlier, or simply create a brand-new F# Console application.

2. Create a new F# Source file called `Domain.fs` by selecting **Add New Item** when right-clicking the project, and then selecting **Code -> Source File**. This file will hold our types that make up our domain. Recall that .fs files act like .cs files – they live inside a project, and are compiled into a full-blown .NET assembly.



**Figure 12.5:** Creating a new .fs file in a project.

3. Create a second file, called `Operations.fs`. This will contain the functionality that acts on the domain.
4. Go into `domain.fs`. You'll see it contains simply a module declaration, which you can delete. Instead, add the declaration for a `Customer` record inside the `Domain` namespace. You declare the namespace that code is in by simply using the namespace `<my namespace>` declaration at the top of the file e.g. `namespace MyTypes`. Unlike C#, you *don't* need to add curly braces to live “within” this namespace (or even the F# equivalent i.e. indent your code).

### Listing 12.2 Declaring types within a namespace

```
namespace Domain ①

type Customer = ②
 { FirstName : string
 LastName : string
 Age : int }
```

- ① Namespace declaration
- ② Declaring a type to live within the namespace

Now the `Customer` type lives within the `Domain` namespace. Next, let's create our module to contain some functionality that can act on the `Customer`.

1. Open the `Operations.fs` file. You will see it already has the module declaration for us.
2. Underneath this, open the `Domain` namespace.

At this point, you need to watch out for file ordering in the solution explorer. `Domain.fs` must live *above* `Operations.fs` in order for the `Operations` module to access it. If it is placed below, you can highlight `Domain.fs` in solution explorer and use **ALT+UP** to move it up the dependency order, or right-click and use the Move Up context menu. If this isn't done, you'll receive an error message as per Figure 12.7.

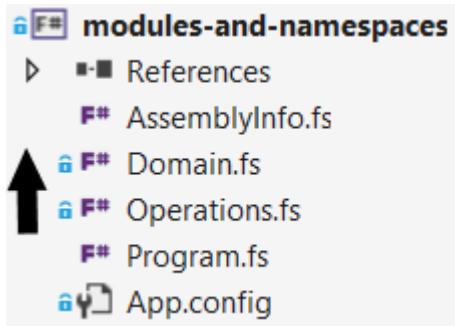


Figure 12.6 `Operations.fs` must live *below* `Domain.fs` to access `Domain.fs` from `Operations.fs`.

```
module Operations
```

```
open Domain
```

The namespace or module 'Domain' is not defined

Figure 12.7 Trying to access an inaccessible namespace

3. Create a couple of functions that act on a `Customer`, for example: -
  - o `getInitials` – gets the initials of the customer.
  - o `isOlderThan` – tests if a customer is older than a certain age

Your module should look something like this: -

#### Listing 12.3 Declaring a module that references a namespace

```
module Operations ①
open Domain ②

let getInitials customer = customer.FirstName.[0], customer.LastName.[0]
let isOlderThan age customer = customer.Age > age
```

- ① Declaring a module
- ② Opening the `Domain` namespace

Make sure all files are saved; we're now ready to hook them together in `Program.fs`.

4. Ensure that `Program.fs` is the last (lowest) file in the project so that it can access both Operations and Domain.
5. You'll notice that we don't have a module declaration in this file. The last file in an application can omit the module declaration and have it taken from the filename e.g. `Program`.
6. Add open statements for both Domain and Operation.
7. Have the main implementation create a customer, and print out whether the customer is an adult (older than 18) or a child.

#### **Listing 12.4 Declaring a module that references a namespace**

```
open Domain ①
open Operations

[<EntryPoint>]
let main argv =
 let joe = { FirstName = "joe"; LastName = "bloggs"; Age = 21 } ②
 if joe |> isOlderThan 18 then printfn "%s is an adult!" joe.FirstName
 else printfn "%s is a child." joe.FirstName ③

 0
```

- ① Opening up custom namespaces
- ② Creating a customer
- ③ Creating a simple pipeline for a function chain

#### **Quick Check**

3. In what order are files read for dependencies in F#?
4. When can you omit a module declaration in an F# file?

### **12.3 Tips for working with Modules and Namespaces**

Let's wrap up by learning a few miscellaneous features of modules and namespaces.

#### **12.3.1 Access Modifiers**

By default, types and functions are always public in F#. If you want to use a function within a module (or a nested module) but don't want to expose it publicly, simply mark it as `private`.

#### **12.3.2 The global namespace**

If you don't supply a *parent* namespace when declaring namespaces or modules, it'll appear in the `global` namespace, which is always open. Both `Domain` and `Operations` live in the global namespace.

### 12.3.3 Automatic opening of modules

You can also have a module *automatically* open, without the caller explicitly even having to use an open declaration, by adding the [`<AutoOpen>`] attribute on the module. With this attribute applied, just opening up the parent namespace in the module will automatically open up access to the module *as well*. You might use this if you have several modules that contain different functionality within the same namespace and would like to open them all up automatically. As long as your program file has access to the containing namespace, you can completely omit the `open` declarations. `AutoOpen` is very commonly used when defining DSLs, as you can simply open a namespace and suddenly get access to lots of functions and operators.

### 12.3.4 Scripts

Some of the rules above work slightly differently with scripts. For starters, you can create let bound functions directly in a script. This is possible because there's an *implicit* module that is created for you based on the *name* of the script (similar to automatic namespacing). You can of course explicitly specify the module in code if you want, but with scripts it's generally not needed.

#### **Quick Check**

5. What is the `AutoOpen` attribute for?
6. What is the default access modifier for values in modules?

## 12.4 Summary

That's it for namespaces and modules! In this lesson: -

- You saw how we typically separate out types and behaviour through namespaces and modules.
- You saw how we create and access namespaces in F#.
- You learned about the module system in F#.
- We built a sample application that uses namespaces and modules to separate out functionality, before calling both elements within a program file.

#### **Try This**

Create a sample module that contains functions that emulate a simple calculator in a module. Experiment with calling the functions from a separate script file. Then, experiment with the [`<AutoOpen>`] attribute; what impact does it have on the caller's code in terms of succinctness?

#### **Quick Check Answers**

1. No. Namespaces can only store types or modules.

2. Yes. Modules can hold types, values and nested modules.
3. Downwards – the first file in the project has no dependencies, the last has no dependants.
4. For the last file in the project.
5. Automatically opening access to the module when the parent namespace is opened.
6. Public.

# 13

## Achieving code reuse in F#

We're going to change tack a little in this lesson, and look at how we can use functions (and type inference) in F# to create lightweight code reuse, and to pass functionality (rather than data) through a system. If you've used the LINQ framework at all, much of this lesson will be familiar to you. We'll cover: -

- How we tend to achieve reuse in the OO world
- A quick review of the core parts of LINQ
- Implementing higher order functions in F#
- Dependencies as functions

We always look to reuse code in our applications, because copy and paste is evil - right? Unfortunately, you'll know just as well as I do that achieving reuse at a low cost is sometimes really, *really* hard to achieve! There are many different types of code reuse that we strive to achieve – in this lesson we'll focus on a common form of reuse, although the outcomes from this lesson can be applied across most forms of reuse.

Let's imagine a situation where you have a collection of customers, and need to filter out some of them based on logic that we don't yet know. In this case, it might be "is the customer female" – but at some point, you might need to write other types of filters - and you don't want to reimplement the logic of "filtering over the customers" every time you need a new filter. What you need is some way to separate out the two pieces of logic, but then "combine them" together as needed.

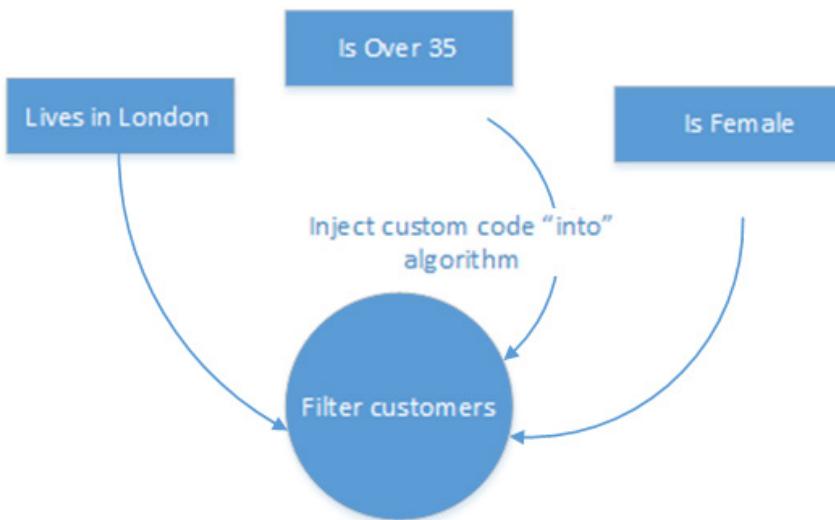


Figure 13.1 – Combining a fixed algorithm with varying custom logic.

Normally we'd look to either inheritance, or perhaps creating an interface to represent our "save" functionality and use something like the Template or Strategy pattern. I'm not going to get into a debate on the virtues of Template versus Strategy – you can read up on that in your own time – but here's a quick example of the Strategy pattern (in this case you might also think of it as variant of the Command pattern as well) in C#:

#### **Listing 13.1 Using interfaces as a way of passing code**

```
interface IFilter { bool IsValid(Customer customer); } ①
IEnumerable<Customer> Where(this IEnumerable<Customer> customers, IFilter filter) { ②
 foreach (var customer in customers)
 {
 if (filter.IsValid(customer))
 yield return customer;
 }
}
```

- ① Filter interface represents a contract used by Where()
- ② Where receives an instance of Filter to allow varying the algorithm

So we now have some "algorithm" that can be reused - the logic of "filtering over customers" - in our **Where** method, and a contract by which we can vary it – our **IFilter** interface – so we can write any arbitrary filter now over customers. Let's see how we can consume this design when we only want to retain customers older than 35:

**Listing 13.2 Consuming an interface-based design**

```
public class IsOver35Filter : IFilter { ①
 public Boolean IsValid(Customer customer) {
 return customer.Age > 35;
 }
}

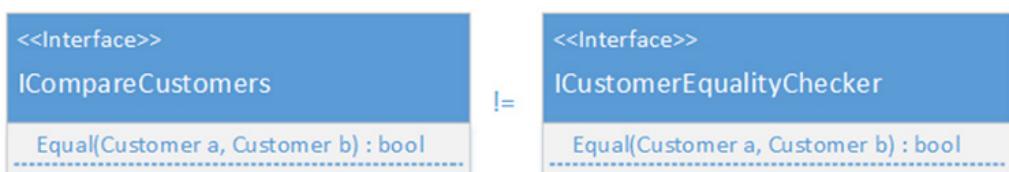
public void FilterOlderCustomers()
{
 var customers = new Customer[0];
 var filter = new IsOver35Filter(); ②
 var olderCustomers = customers.Where(filter); ③
}
```

- ① An instance of an IFilter
- ② Creating an instance of the IsOver35Filter class
- ③ Supplying the filter to the Where method

We've had to create a specific class to implement our IFilter interface, then create an instance of it later on before finally passing it to our Where method.

## 13.1 Reuse in the world of LINQ

Of course, there are many methods in the BCL that follow the above design, particularly from the early days of .NET such as IComparable, IComparer and IEquatable. All of them have a single method on them but require the overhead of an interface and class for you to implement them. Even worse, some of them have exactly the same signature, but you can't reuse them across both because in .NET interfaces are *nominal*, not *structural* – that is, even if two interfaces have the same *structure*, they are still treated as two incompatible types.



**Figure 13.2 – Nominal Types cannot be implicitly exchanged for one another even if they have the same structure**

LINQ, and C#3, introduced a whole raft of features which were inspired from the world of functional programming. One of the biggest takeaways from it was the pervasive use of *higher order functions* (HOF) throughout the LINQ framework. Despite the somewhat technical name, a HOF is simply a function which takes in *another* function as one of its arguments. Let's look at a design similar to LINQ's **Where** method when acting on customers:

### Listing 13.3 Using higher order functions to reuse code

```
IEnumerable<Customer> Where(this IEnumerable<Customer> customers, Func<Customer, bool>
 filter) { ①
 foreach (var customer in Customers)
 {
 if (filter(customer)) ②
 yield return customer;
 }
}
```

- ① Using `Func<Customer, bool>` as a means of a contract instead of an interface
- ② Calling the filter on the customer directly

This method looks suspiciously like that in Listing 13.1, except here we don't require a specific interface – instead, we simply pass in a *function* that adheres to a contract: it must take in a Customer and returns a Boolean – the same signature that the **IsValid** method has. This is much more flexible than working with interfaces, because *any* function that has this signature is compatible – it doesn't have to explicitly have been designed to be. We can now call it by passing the function directly, as can be seen in Listing 13.4.

In order to do this even more succinctly, C#3 also introduced the concept of *lambda expressions* – another technical-sounding term, but in reality, it's just a way of declaring a function *inline* of a method.

### Listing 13.4 Consuming a higher order function

```
public Boolean IsOver35(Customer customer) { ①
 return customer.Age > 35;
}

// ...code elided...
var olderCustomers = customers.Where(IsOver35); ②
var olderCustomersLambda = customers.Where(customer => customer.Age > 35); ③
```

- ① Creating a function of signature `Customer -> bool` to check a customer age
- ② Providing the `IsOver35` function to the `Where` higher order function
- ③ Reimplementing `IsOver35` as an inline lambda expression

With the lambda-based approach, we've achieved exactly the same logic as we started with, except now it's all achieved in a single line – which is generally a good thing: the code is more readable, there's less that can go wrong, it's easier to change etc. Let's take a moment to review these two methods for achieving reuse: -

**Table 12.1 – Comparing OO and FP mechanisms for reuse**

|                        | Object Oriented     | Functional            |
|------------------------|---------------------|-----------------------|
| Contract specification | Interface (nominal) | Function (structural) |
| Common Patterns        | Strategy / Command  | Higher Order Function |

|                       |                                |                  |
|-----------------------|--------------------------------|------------------|
| Verbosity             | Medium / Heavy                 | Lightweight      |
| Composability & Reuse | Medium                         | High             |
| Dimensionality        | Multiple methods per interface | Single functions |

### Delegates and anonymous methods

One interesting point is that .NET (and C#) has always supported the notion of typesafe function pointers through both delegates (C#1) and anonymous methods (C#2) – in effect, both are rendered obsolete by the introduction of Func<T> (and lambda expressions), which is a much more lightweight syntax than either of those.

### Java's approach to functions

Java introduced the concept of lambda expressions relatively late in the day (Java 8 in 2015). By this time, the “single method interface” was so common that the designers of Java opted to simply make single-method interfaces implicitly compatible with lambda function signatures – in effect, all single-method interfaces are treated as potential lambda expressions. In this way, all existing interfaces were “automatically” promoted into being usable as lambdas.

### Quick Check

1. Name one difference between nominal and structural types.
2. How do we pass logic between or across code in the OO world?
3. How do we pass logic between or across code in the FP world?

## 13.2 Implementing higher order functions in F#

C# and VB .Net (and the BCL) have a kind of mish-mash of both interface and high order function strategies – you’ll see newer features that were added to the BCL generally favour lambdas and higher order functions e.g. the Task Parallel Library, whilst older features usually favour interfaces and classes. Conversely, F#’s built-in libraries almost exclusively focus on higher order functions; as such, F# makes higher order functions extremely easy to work with and create.

### 13.2.1 Basics of Higher Order Functions

#### Now you try

Let’s start by trying to implement an equivalent of the behaviour above i.e. filter in F# to see the difference between both languages and approaches: -

#### Listing 13.5 Your first higher order function in F#

```
type Customer = { Age : int }
```

```

let where filter customers =
 seq {
 for customer in customers do
 if filter customer then ①
 yield customer }

let customers = [{ Age = 21 }; { Age = 35 }; { Age = 36 }]
let isOver35 customer = customer.Age > 35 ②

customers |> where isOver35 ③
customers |> where (fun customer -> customer.Age > 35) ④

```

- ① Calling the filter function with customer as an argument
- ② Explicitly creating a function to check the customer age
- ③ Supplying the isOver35 function into the where function
- ④ Passing a function inline using lambda syntax

Try executing this code. You'll see that both of the last lines returns the only `Customer` that is over 35. In the next samples, you'll see a couple of language features you've not seen yet: -

- The use of the `seq { }` block – this is a type of *computation expression* in F#, a more advanced topic we'll touch on later for asynchronous programming – here it's used to express that we're generating a sequence of customers using the `yield` keyword.
- An F# list, expressed using `[ ; ; ]` syntax – something we'll be covering in the next unit.

Just like C#, whilst we can throw let bound functions directly as a higher order function argument, we can also use F#'s lambda syntax. In fact, this code is essentially the same as the C# example, except I've swapped the order of the `filter` and `customer` arguments so that I can "pipe" `customers` into `where` (remember that `|>` works by flipping the *last* argument over to the left).

Our old friend type inference has come into play again, and it's worthwhile spending a little time looking at the type signature of the function: -

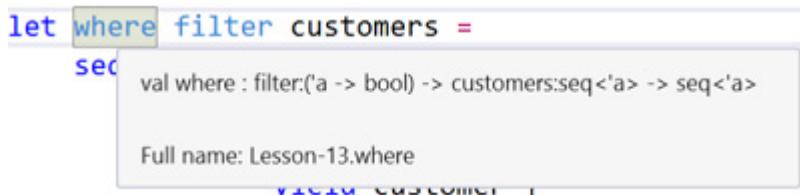


Figure 13.3 – F# inferring a higher order function automatically

As you can see from Figure 13.3, `filter` is identified as a function that takes in some `'a` and returns a `bool`.

- The compiler knows it's a function based on usage i.e. `filter customer`.

- It must return a boolean since the output is used in an if clause.
- It takes in any `a. This is interesting - in our original C# example, we explicitly bound our code to customers, but F# has realised that this function would work just as well over orders or numbers etc., so it's simply made it generic.
- A quick way to confirm that `filter` has been correctly identified as a function is to notice that it is coloured differently – assuming that F# Power Tools has been configured correctly.

Also, observe that the `customers` value has been identified as a `seq` (F# shorthand for `IEnumerable<T>`) since we use it within a `for` loop.

### 13.2.2 When to pass functions as arguments

Passing functions as arguments is something you'll do extremely often when working in F# as it's the primary way of achieving reuse, and when coupled with F#'s ability to infer, compose and pipeline functions is easy to achieve without having to write reams of hard-to-read `Func<string,int,bool>`-style type annotations. Although F# has support for interfaces (in fact, in some ways better support than C#), it's certainly not idiomatic to use them except when you're passing many dependencies as a logically grouped set of behaviours e.g. perhaps a set of logging functions or similar.

It's also fairly easy to create higher order functions by "reverse engineering" them – simply start by creating a normal function with the "varying" element hard coded into the algorithm. Then, identify all occurrences of that section, replace them with a simple named value which is added as an argument to the function. Here's a hard-coded version of the "filter customers over 35" function:

```
let whereCustomersAreOver35 customers =
 seq {
 for customer in customers do
 if customer.Age > 35 then
 yield customer }
```

Figure 13.4 – A hard-coded function that can be converted into a higher order function

Observe that this is the same as Listing 13.5, except the `filter customer` which was passed in as the first argument has been replaced with the highlighted element of code.

#### Quick Check

4. Can F# infer the types of higher order functions?
5. How can you easily identify higher order function arguments in VS?

### 13.3 Dependencies as functions

Whilst you can easily reference one piece of code from another in F#, using namespaces and modules to reference functions in other files etc., there are times when you'll want to decouple two sections of code from one another, often because you want to be able to "swap out" the implementation of one without affecting the other. A common use case for this is writing testable code – for example, you might "decouple" your code from a "real" database so that you can mock out data that it reads and writes to. This is often known as **dependency injection** – the class tells you what it requires in the constructor arguments, and you supply those requirements as dependencies. These dependencies often take the form of interfaces which contain the behaviour that can be plugged in.

However, we've also seen how many interfaces – particularly ones with single methods – can simply be replaced with functions. Indeed, it's often preferable to explicitly pass in dependencies as functions rather than one larger interface containing dozens of methods of which you only need one or two – it becomes much clearer to understand the relationship between a function and its dependencies. In F#, we can just as easily pass in dependencies, but instead of passing them into *constructors of classes*, we can simply pass them *into functions directly*.

#### Now you try

Let's try to write a function that will print out a specific message regarding the Customer's age to a variety of output streams, such as Console or the file system etc.

1. Create an empty script file and define a Customer record type (or continue below the existing script you've been working on).
2. Create a function, `printCustomerAge` that takes in a Customer and depending on the Customer's age prints out "Child", "Teenager" or "Adult", using `Console.WriteLine` to output text to FSI. The signature should read as: `let printCustomerAge customer =`
3. Try calling the function, and ensure that it behaves as expected.
4. Identify the "varying" element of code. For us, this is the call to `Console.WriteLine`.
5. Replace all occurrences with the value `writer`. Initially, your code will not compile, as there is no value called `writer`.
6. Now insert `writer` as the first argument to the function, so it now reads `let printCustomerAge writer customer =`.
7. You'll see that `writer` has been correctly identified as a function that takes in a `string` and returns an `'a`. Now, any function that takes in a `string` can be used in place of `Console.WriteLine`.

#### Listing 13.6 Injecting dependencies into functions

```
let printCustomerAge writer customer = ①
 if customer.Age < 13 then writer "Child!" ②
 elif customer.Age < 20 then writer "Teenager!"
```

```
else writer "Adult!"
```

- ① Specifying our dependency as the writer argument
- ② Calling writer with a string argument

You're now in a position to call this function. First you can confirm it works as before by passing `Console.WriteLine` as the first argument. You can also use the partial application trick to "build" a constrained version of `printCustomerAge` that prints to the Console.

### **Listing 13.7 Partially applying a function with dependencies**

```
printCustomerAge Console.WriteLine { Age = 21 } ①
let printToConsole = printCustomerAge Console.WriteLine ②
printToConsole { Age = 21 }
printToConsole { Age = 12 }
printToConsole { Age = 18 }
```

- ① Calling `printCustomerAge` with `Console.WriteLine` as a dependency
- ② Partially applying `printCustomerAge` to create a constrained version of it

8. Now let's try to create a function which can act as the dependency, in order to print out to the filesystem instead. We'll use `System.IO.File.WriteAllText` as the basis for our dependency (if the `temp` folder doesn't exist, create it first!):

### **Listing 13.8 Creating a dependency to write to a file**

```
open System.IO
let writeToFile text = File.WriteAllText(@"C:\temp\output.txt", text) ①
let printToFile = printCustomerAge writeToFile
printToFile { Age = 21 }
```

- ① Creating a File System writer that is compatible with `printCustomerAge`

9. Try to read back from the file using `System.IO.File.ReadAllText` to prove that the content was correctly written out.

You'll notice that I explicitly stated that you should supply dependencies as the first argument(s) in a function. This is so that you can *partially apply* the function – you inject the dependencies up front (e.g. `Console.WriteLine` in our case) which returns you a *new* function which requires the remaining argument(s) – in our case, just the customer object. This partially applied function might *itself* then be passed into other functions, who will have no coupling to e.g. `Console` or File Systems etc.

### **Quick Check**

6. What's the key difference between passing dependencies in F# and C#?

## 13.4 Summary

In this lesson, we learned about higher order functions (HOFs) – the primary way that we vary algorithms and pass code in F# - which prepares us for working with the collections modules that we'll be looking at in lesson 13.

- We looked at typical OO designs for extending behaviours through interfaces, and compared them to the functional approach of composing functions together through HOFs.
- We gained an understanding of how F#'s type inference engine makes it extremely easy to reverse engineer HOFs from existing code.
- We saw how we can use HOFs as a lightweight form of dependency injection.

### **Try This**

Create a set of functions that use another dependency in .NET – for example, working with HTTP data using WebClient. For example, write a function that “takes in” the HTTP Client to POST data to a URI. What is the dependency? The WebClient class, or a function *on* the WebClient?

### **Quick Check Answers**

1. Nominal types are defined by their fully-qualified type name. Structural types are defined by their signature.
2. We use interfaces to pass logic within a codebase in the OO world.
3. We use functions to pass logic within a codebase in the FP world.
4. Yes.
5. Visual F# Power Tools highlights functions in a different colour.
6. Dependencies in F# tend to be functions; in C#, they're interfaces.

# 14

## Capstone 2

**Before we move onto the next unit – collections – here’s another “end-of-level” bad guy for you to defeat. This time, we’ll shift our focus from the basics of coding in F# to what we covered in this unit. So, in this lesson you’ll be expected to: -**

- Develop a standalone F# application in Visual Studio
- Model a domain using Records, Tuples and Functions
- Create reusable higher order functions that can be altered through injected dependencies

That’s a lot to do, but if you take this step by step, you’ll do fine.

### 14.1 Defining the problem

In this exercise, you’re going to write a simple bank account system. It needs to have the following capabilities: -

1. The application should allow a customer to deposit and withdraw from an account that they own, and maintain a running total of the balance in the account.
2. If the customer tries to withdraw more money than they have in their account, the transaction should be declined i.e their balance should stay as is.
3. The system should write out all transactions to a data store when they are attempted. The data store should be pluggable e.g. file system, console etc.
4. The code should not be coupled to e.g. the file system or console input – it should be possible to access the code API directly without resorting to a console application.
5. Another developer will review your work, and they should be able to easily access all of these above components in isolation from one another.
6. The application should be an executable as a console application.
7. On startup, the system should ask for the customer’s name and opening balance. It then should create (in memory) an account for that customer with the specified balance.

8. The system should then repeatedly ask if the customer wants to deposit or withdraw money from the account.
9. The system should print out the updated balance to the user after every transaction.

What you *don't* have to worry about is: -

- Reading data back from the file system. The system should simply store the customers' current balance in memory. If the application is closed, there's no way to resume later on.
- Don't worry about opening multiple accounts.
- Don't worry about warning the user if they try to go overdrawn. Simply carry on with the same balance that they started with.

## 14.2 Some advice before you start...

This solution will be larger than the one in the previous capstone. So, before you dive in and start writing reams and reams of code, let me give you some simple advice that I always follow when approaching a sizeable chunk of work in F#: -

1. Start small. Resist what will probably be your natural urge to design a complex set of objects and relationships up front. Instead, just write simple functions that each do one thing, and do it well. Trust that you can compose them together later on.
2. Plug these functions together, either by composing them to one another through a third function that calls both, or calling one from another (perhaps via a higher order function).
3. Don't be afraid of copying and pasting code initially – you can refactor quickly in F#, especially when using higher order functions. See where reuse is through actual evidence of code, rather than prematurely guessing where it might be.

## 14.3 Getting started

As usual, we'll start by working with a simple script with some types and functions, experimenting and exploring our domain. Once we're happy with what we have, we'll migrate the code over to a full console executable that can be run as a standalone application. If you do get stuck, refer to the solution in the `code-listings/lesson-14` folder to help you. However, try to avoid simply copying and paste code from it – you'll get much more out of this by trying to do this yourself – only use the suggested solution as a last resort.

Start by creating a new F# Console Application named **Capstone2**, and add an empty .fsx file to the project.

## 14.4 Creating a domain

We'll begin by first trying to model the types in our domain – we'll use F# Records for this. You can identify two entities in our domain: -

- **Customer:** A named customer of our bank.
- **Account:** An account that is owned by a customer. An account should probably have a current balance, a unique ID, and a reference to the Customer that owns the account.

Create two record types that match the above definition, and then create an instance of an account in the script directly underneath – just to ensure that you’re happy with the shape of the account and the fields in it.

## 14.5 Creating behaviours

We now need a couple of functions to model withdrawals and deposits into the account. I’ll help you out by giving you a typical function signature for deposit and a hard-coded implementation that needs replacing.

### Listing 14.1 Sample function signature for deposit functionality

```
/// Deposits an amount into an account
let deposit (amount:decimal) (account:Account) : Account = ①
{ AccountId = Guid.Empty; Owner = { Name = "Sam" }; Balance = 10M }
```

① Pure function signature

Just to confirm: This is a *pure, curried* function that takes in two arguments (`amount` and `account`) and returns a new `Account` i.e. the updated version with the increased balance. Notice that I’ve explicitly type annotated this function; this isn’t necessary, and you can remove the annotations later – it’s just to help you along here if needed. Also, also notice that I’ve put the state – `account` – as the last argument to the function. This is so that we can pipe data through a chain e.g. `account |> deposit 50 |> withdraw 25 |> deposit 10`. Also, remember that you can use *copy-and-update* syntax in F# (using the `with` keyword) to create a new version of a record with updated data.

You’ll also need to create a `withdraw` function. It will have an identical signature, but the implementation will be slightly more complex – if the amount is greater than the balance, just return the account that was supplied. Otherwise, return an updated account with reduced balance. Make sure you test them out in the REPL / script as you go to ensure you’re happy with the code.

In the function signature above, we’ve not passed in a `Customer` record – that’s because in my model, `Customer` is a field on `Account`: -

### Listing 14.2 Suggested domain model

```
type Customer = { Name : string } ①
type Account = { AccountId : System.Guid; Owner : Customer; Balance : decimal } ②
```

① Customer record

② Account record with Customer as the Owner field

## 14.6 Abstraction and reuse through higher order functions

The next thing we need to think about is a logging / auditing mechanism. Let's write a couple of simple audit functions, one for the file system and one for the console. Both should have the same signature:

### Listing 14.3 Creating pluggable audit functions

```
let fileSystemAudit account message = ①
let console account message = ②
```

- ① Auditor that writes to file system
- ② Auditor that prints to console

In effect, these replace the need for the typical `ILogger` interface you might have used in the past that has a single `Log()` method on it.

For the file system auditer, it should append to the contents of `message` to a file whose path is `C:\temp\learnfs\capstone2\{customerName}\{accountId}.txt`. You will probably want to use `sprintf` as well as some methods within the `System.IO.File` namespace e.g. do you need to ensure that the directory exists first?

The console auditer should simply print to the console in the format "Account <accountId>: <message>". In this case, you'll probably want to use `printfn` e.g. "Account d89ac062-c777-4336-8192-6fba87920f3c: Performed operation 'withdraw' for £50. Balance is now £75".

Again, test these functions out in isolation in a script to prove you're happy with them – create a dummy account and customer and pass them in, ensuring that the correct outputs occur e.g.: -

### Listing 14.4 Testing out functions through scripts

```
let customer = { Name = "Isaac" }
let account = { AccountId = Guid.Empty; Owner = customer; Balance = 90M } ①

// Test out withdraw
let newAccount = account |> withdraw 10M ②
newAccount.Balance = 80M // should be true!

// Test out console auditer
console account "Testing console audit" ③
// "Account 00000000-0000-0000-0000-000000000000: Testing console audit"
```

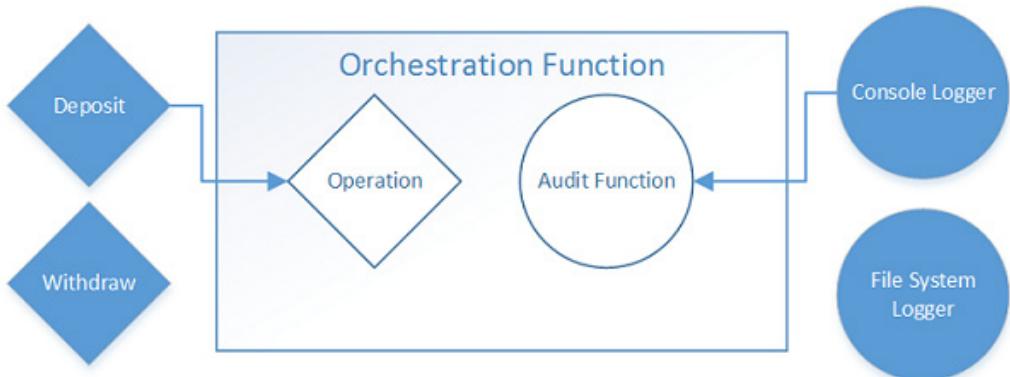
- ① Creating a dummy account for testing
- ② Testing the withdraw function
- ③ Testing out the console auditer

### 14.6.1 Adapting code with higher order functions

At this point notice that our behaviours have little in common – the `deposit` and `withdraw` functions have no ability to perform auditing; meanwhile our audit functions have no

knowledge of actual account behaviours, nor create the messages that need to be audited. We'll need something to wire them up together and actually create our audit messages! You'll need to create a new function that should: -

- Try to perform some arbitrary account operation (withdraw or deposit)
- Audit the details of the transaction e.g. "withdraw £50"
- If the account balance was modified, audit a message with the details of the transaction and the new balance
- If the account balance was not modified, audit a message that the transaction was rejected
- Return the updated account



**Figure 14.1 – Composing disparate behaviours into a single function**

This looks pretty nice in principle, but how can we write some code to actually achieve this? Observe that the description above is devoid of "implementations" – `operation` is either `withdraw` or `deposit`; similarly, we don't mention the type of auditor e.g. `console` or `file system`. Here's what our signature should look like: -

#### **Listing 14.5 Signature for an orchestration higher order function**

```
let auditAs (operationName:string) (audit:Account -> string -> unit) (operation:decimal ->
 Account -> Account) (amount:decimal) (account:Account) : Account = ①
```

##### **① Sample audit orchestration function**

This function should "wrap around" *both* an operation (e.g. `withdraw`) and an audit function (e.g. `console`), calling both of them appropriately. Let's review this function signature, one argument at a time: -

- `operationName`: The name of the operation as a string e.g. "withdraw" or "deposit"

- `audit`: The audit function we wish to call e.g. the console audit function
- `operation`: The operation function we wish to call e.g. the withdraw function
- `amount`: The amount to use on the operation
- `account`: The account to act upon

The function also returns the updated account. Let's compare this function to the signature of one of our operations, `deposit`:-

```
// Runs some account operation such as withdraw or deposit with auditing.
let auditAs operationName audit operation amount account =
 "Dependency" arguments "Remaining" arguments
```

**Figure 14.2 – Comparing the standalone deposit function with the wrapping auditAs function.**

It's import to remember that because `auditAs` is a curried function, we can just pass in the first three arguments, and get back a *new* function that requires the remaining arguments – which matches the signature of the original `deposit` function! Let's see how it works:-

#### **Listing 14.6 Partially applying a curried function**

```
let account = { AccountId = Guid.NewGuid(); Owner = { Name = "Isaac" } Balance = 100M } ①

account ②
|> deposit 100M
|> withdraw 50M

let withdrawWithConsoleAudit = auditAs "withdraw" consoleAudit withdraw ③
let depositWithConsoleAudit = auditAs "deposit" consoleAudit deposit

account ④
|> depositWithConsoleAudit 100M
|> withdrawWithConsoleAudit 50M
```

- ① Creating an account and customer
- ② Calling the “raw” deposit and withdraw functions
- ③ Creating new “decorated” versions of deposit and withdraw with console auditing through currying
- ④ Calling the “decorated” versions of deposit and withdraw

Create the implementation of the `auditAs` function and test that you can call it correctly. If you struggle to figure it out, first write a version that is tightly coupled to e.g. the `deposit` function and console logging (i.e. does not have the “dependency arguments” as per the previous figure); then pull those functions out as dependencies one at a time.

## 14.7 Writing a console application

Up until now, we only have a single F# script file – not much use as a standalone application. Let's pull the code we've written so far into dedicated modules and namespaces so that they can be built into a compiled application. Create some fs F# files as follows in the project: -

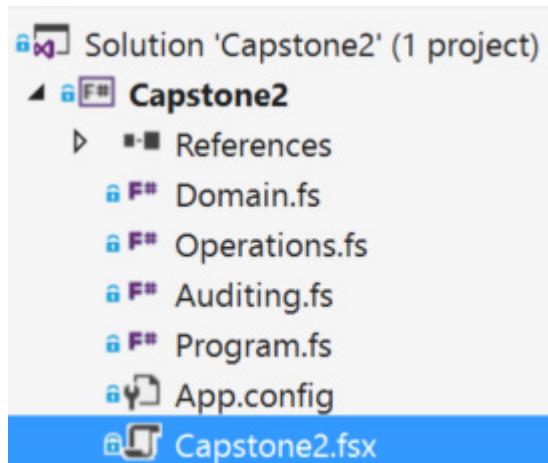


Figure 14.3 – Pulling script code out into a full application

- **Domain.fs:** contains the `Customer` and `Account` record types in the `Capstone2.Domain` namespace
- **Operations.fs:** contains the `deposit`, `withdraw` and `auditAs` functions in the `Capstone2.Operations` module
- **Auditing.fs:** contains the `console` and `filesystem` audit functions in the `Capstone2.Auditing` module
- **Program.fs:** contains the actual bootstrapper and runner

### 14.7.1 Writing the Program

The entry point program itself will be fairly simple: -

- Use a combination of `System.Console.ReadLine` and `Console.WriteLine` (or `printfn`) functions to get the user's name and opening balance, and create an `Account` and `Customer` record.
- Use `Decimal.Parse` to convert from a string to decimal. Don't worry about error handling – we can deal with that another day.
- Create decorated versions of both the `deposit` and `withdraw` functions that use the console auditor.
- Use a while loop to find out what action the user wishes to do (deposit / withdraw /

exit) – again, use Console functions to get user input. See 1.7.2 for more details on this.

- Depending on input, call the appropriate function i.e. the decorated deposit / withdraw functions to get an updated account.

### 14.7.2 Managing the account state

Unfortunately, at this stage we don't really know enough about state management to get away without using a mutable value to store the account state. At a later point in this book, we'll identify ways of writing imperative style code that are "normally" written as while loops with external state, but for now, here's a simple scaffold that you can "fill in the blanks": -

#### **Listing 14.7 Simplified main application**

```
let mutable account = ... ①

let withdrawWithAudit = withdraw |> auditAs "withdraw" Auditing.console ②
let depositWithAudit = deposit |> auditAs "deposit" Auditing.console

while true do
 let action = ... ③

 if action = "x" then Environment.Exit 0

 let amount = ... ④

 account <- ⑤
 if action = "d" then account |> depositWithAudit amount
 elif action = "w" then account |> withdrawWithAudit amount
 else account ⑥
```

- ① Build Account (and child Customer) record from console input
- ② Build decorated deposit and withdraw functions
- ③ Find out the user's action – "d", "w" or "x"
- ④ Find out the amount to use for the deposit or withdrawal
- ⑤ Call the appropriate operation
- ⑥ Default handler – do nothing

As in our earlier capstone, we'll use a mutable variable for controlling the overall loop here.

## 14.8 Referencing files from scripts

As you port code from scripts to full assemblies, you'll often find yourself wanting to access code that you've already ported from a script – perhaps you're testing out new code that interacts with some existing code etc. In this case, it's possible to load an `fs` file *into* an `fsx` script. This way, you can get the best of both worlds – you can write code that can be accessed from a script, but still run from within e.g. a console application or website etc. Try this code listing below from an empty `fsx` file.

**Listing 14.8 Accessing fs files from a script**

```
#load "Domain.fs" ①
#load "Operations.fs"
#load "Auditing.fs"

open Capstone2.Operations ②
open Capstone2.Domain
open Capstone2.Auditing
open System

let withdraw = withdraw |> auditAs "withdraw" consoleAudit
let deposit = deposit |> auditAs "deposit" consoleAudit

let customer = { Name = "Isaac" } ③
let account = { AccountId = Guid.NewGuid(); Owner = customer; Balance = 90M }

account ④
|> withdraw 50M
|> deposit 50M
|> deposit 100M
|> withdraw 50M
|> withdraw 350M
```

- ① Loading fs files into a script
- ② Opening namespaces of fs files
- ③ Creating some initial data
- ④ Testing code against a sample pipeline

The important part here is the `#load` directive. You can use `#load` to execute both `fsx` scripts and `fs` files directly into a script as if you had entered the code directly yourself. You need to think about the order of `#load` commands – you couldn't load `Operations.fs` before you had loaded `Domain.fs`, since the former depends on the latter.

If your code has been implemented correctly, you should see something as follows when executing the script: -

**Listing 14.9 Sample output of running the sample bank account script**

```
Account ...: Performing a withdraw operation for £50...
Account ...: Transaction accepted! Balance is now £40.
Account ...: Performing a deposit operation for £50...
Account ...: Transaction accepted! Balance is now £90.
Account ...: Performing a deposit operation for £100...
Account ...: Transaction accepted! Balance is now £190.
Account ...: Performing a withdraw operation for £50...
Account ...: Transaction accepted! Balance is now £140.
Account ...: Performing a withdraw operation for £350...
Account ...: Transaction rejected!
```

## 14.9 Summary

I hope that this exercise wasn't *too* difficult. Don't feel bad if you didn't think of everything up front and / or had to look at the suggested solution – it'll take time for you to develop an

"instinct" for when to use what tools in the language. Also, definitely don't feel frustrated if you thought "I could have done this with my eyes closed in C!"

Part of the difficulty nearly every developer goes through when learning F# (or any FP language) when coming from an OO language is to resist the temptation to fall back to the safety net of curly braces and mutation, or to use console runners as a way to iteratively develop an application. In fact, I suspect that some other experienced F# developers might have come up with a different solution to mine (notwithstanding the fact that we're not using all the features in F# yet!) – just like with OO, there are always different ways to solve challenges. But the core "features" of the application – immutability, expressions, pure functions, higher order functions etc. – would almost certainly all be present.

I would recommend that you think up some other coding challenges like this one yourself – something that requires you to do a little data modelling, and a little functional design, and try to implement it. The more you do exercises like this, the more your muscle memory will become attuned to all things F#, from basics like syntax for creating records to more advanced refactoring and designing higher order functions.

# Unit 4

## Collections in F#

One of the difficulties in designing a book such as this is that many of the language and library features overlap with one another, and it's often hard to focus on one specific aspect without introducing others into the mix as well. When I originally planned this book, I had envisaged discussing collections a little later – but after thinking about it, I decided to bring it forwards.

The reason for that is because collections in F# are *fantastic!* Combined with the succinct syntax you've already seen, and the possibility of a REPL-based environment, F# allows us to start working with data in all sorts of ways that you might not have considered before – something we'll revisit in Unit 7 (Working with Data).

Part of this unit will cover the basic ideas of functional collections – something that, if you've ever used LINQ before, will actually be quite familiar to you – and get you up to speed with the typical, most commonly used collection functions. The latter half of the unit will focus on some slightly more advanced tips and tricks to achieve features such as how we can use functional collections to work with immutable data to *simulate* imperative loops and mutation.

# 15

## *Working with collections in F#*

**Something we've only touched on very briefly so far is working with *collections* of data. Nearly everything we've done so far has involved dealing with a single record, or tuple etc. at a time. Yet F# has excellent capabilities when it comes to working with data sets – indeed, working with data is one of its strongest features, as we'll start to see more and more in the coming lessons. In fact, I've set aside this lesson and three more to discuss working with collections! This lesson will:** -

- Introduce you to some of the key collection types in F#
- Get you thinking about transformations in terms of pipelines
- Illustrate how to use immutable F# collections

Simultaneously one of C#'s greatest strengths and weaknesses is that it's become an extremely flexible language, allowing developers to pick any number of ways to approach a problem. This is great in the sense that it can appeal to many types of developers, but it also means – particularly for newcomers – that it can be difficult to get a steer on a consistent, idiomatic way to solve that problem. One great example of that is working with collections. I tend to see this as being divided into three camps: -

1. The "C#2" developer. When thinking about collection operations, the developer thinks in terms of imperative operations – for each loops, accumulators and mutations. They have never found working with LINQ particularly natural or enjoyable.
2. The "LINQ" developer. This developer has embraced C#3 features and uses lambda expressions when working with lists of data for simple operations such as filters, but will still use mutation and imperative code for non-obvious situations. In my experience, these developers often find it easier making the leap to functional programming.
3. The "Wannabe FP" developer. This developer has not only embraced LINQ over collections, but also has started to use those features for more general operations, be they accumulating data through aggregations, rules engines or parallelisable

computations etc. These developers, perhaps without even realising it, have already started to embrace functional programming.

Note that I'm not suggesting that any "one" type of developer in this (simplistic) generalisation are better or worse than the other, but what I would say is that depending on your current view point, you may find more or less of this chapter natural or alien to you! Unsurprisingly, with F# being an FP-first language, you'll find over the next few lessons that collections are used for all sorts of things, and not necessarily just for the typical "filter a list of customers" example.

## 15.1 F# Collections basics

We'll start with a simple challenge. Given a set of football (that's "soccer" for those of you that are not European!) results, we'll try to get an answer to the following question: show me which teams *won* the most *away* games in the season. Here's the structure of a single `FootballResult` record and some sample results: -

### **Listing 15.1 A sample dataset of football results**

```
type FootballResult = { HomeTeam : string; AwayTeam : string; HomeGoals : int; AwayGoals : int } ①
let create (ht, hg) (at, ag) = { HomeTeam = ht; AwayTeam = at; HomeGoals = hg; AwayGoals = ag } ②
let results = ③
 [create ("Messiville", 1) ("Ronaldo City", 2)
 create ("Messiville", 1) ("Bale Town", 3)
 create ("Bale Town", 3) ("Ronaldo City", 1)
 create ("Bale Town", 2) ("Messiville", 1)
 create ("Ronaldo City", 4) ("Messiville", 2)
 create ("Ronaldo City", 1) ("Bale Town", 2)]
```

- ① Record of our input data
- ② Simple helper function to quickly construct a record taking in two (string \* string) tuples
- ③ An F# list of records starts and ends with []

### **Now you try**

Before we go through some of the alternative solutions available, have a go at trying this out yourself in C# or VB .NET (or of course F# if you like!), using whichever style of programming you feel most comfortable in. The output that you want to end up with should look like this: -

1. Bale Town: 2 wins
2. Ronaldo City: 1 win

### **15.1.1 In-place collection modifications**

How did that go? Whichever solution you picked, let's review an imperative style for solving this sort of problem: -

1. Create an output collection to store the summary data – perhaps a mutable DTO called Team Summary that has the Team Name and Number of Away Wins.
2. For every result, if the away team scored more goals than the home team...
3. Check if the output collection already contains this team.
4. If it does, increase the count of Away Wins for that entry.
5. If it doesn't, create a new entry with Away Wins set to 1.
6. Implement a sort algorithm to ensure that the results are sorted based on the number of away wins.

### **Listing 15.2 An imperative solution to a calculation over data**

```
open System.Collections.Generic
type TeamSummary = { Name : string; mutable AwayWins : int } ①
let summary = ResizeArray() ②

for result in results do ③
 if result.AwayGoals > result.HomeGoals then
 let mutable found = false ④
 for entry in summary do
 if entry.Name = result.AwayTeam then
 found <- true
 entry.AwayWins <- entry.AwayWins + 1
 if not found then
 summary.Add { Name = result.AwayTeam; AwayWins = 1 }

let comparer = ⑤
 { new IComparer<TeamSummary> with
 member this.Compare(x,y) =
 if x.AwayWins > y.AwayWins then -1
 elif x.AwayWins < y.AwayWins then 1
 else 0 }

summary.Sort(comparer)
```

- ① Defining our output summary type
- ② Accumulator for output. ResizeArray is an alias for System.Collections.Generic.List
- ③ Core algorithm
- ④ Flag to check if this is a new entry in the accumulator
- ⑤ Custom IComparer for sorting based on away wins

After executing this code, if you “evaluate” `summary` (simply highlight the value and send it to FSI) you’ll see the output. There are a few things to observe here: -

1. The code follows a “flow chart” style design, with branching decisions based on intermediate state.
2. It’s very difficult to see intermediate stages of this code, and there’s nothing to suggest that any of it is really easily reusable. It’s more like we’ve taken the original objectives and mangled them together into a broth; as such, the final code doesn’t reflect the original intent to me insofar as you can’t read it at a glance to know what it does.
3. We modify in-place (mutate) the summary list when sorting.

4. As a side note, you can see one of F#'s nice features for working with interfaces, called *object initialisers*. We created an *instance* of `IComparer` without having to first define a concrete type!

Let's now compare this with a more declarative style of processing that fits much better with a functional style – expressions over immutable data with pure functions.

### 15.1.2 The collection modules

At this point, I want to introduce you to the collection modules – think of these as F#'s own version of the LINQ `Enumerable` library (although they are more than that in reality). There are three modules, each tied to an associated F# collection datatype – `List`, `Array` and `Seq` – containing functions designed for querying (and generating) collections (we'll see more about those three types in 15.2). The good thing is that whilst each module is optimised for the data type in question, they actually contain virtually identical surface areas – so once you learn one of them, you can reuse the same skills across the other two. Most of the query functions in these modules are *higher order functions*, and they follow a similar pattern: -

1. **Input #1:** A user-defined function to customise the higher order function
2. **Input #2:** An input list / array / seq to apply the function against in some way
3. **Output:** A new list / array / seq with the result of the operation

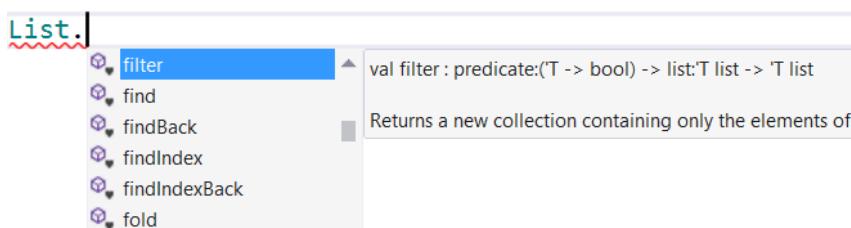


Figure 15.1 – Exploring the F# List module within an F# script.

This follows a similar pattern to LINQ, except whereas in LINQ the “input collection” is the *first* argument (in order to play nicely with extension methods), in F# the input collection is always the *last* argument to the higher order function (which is curried). This is, once again, in order to play nicely with the pipeline operator – the output (state) of one operation can be chained with the next one, much like LINQ does with extension methods. Let's look at some examples of these higher order functions for working with collections: -

#### **Listing 15.3 Standard pattern for F# collection module functions**

```
let usaCustomers = Seq.filter areFromUSA sequenceOfCustomers ①
let numbersDoubled = Array.map (fun number -> number * 2) arrayOfNumbers ②
let customersByCity = List.groupBy (fun c -> c.City) customerList

let ukCustomers = sequenceOfCustomers |> Seq.filter areFromUK ③
let tripledNumbers = arrayOfNumbers |> Array.map (fun number -> number * 3)
```

```
let customersByCountry = customerList |> List.groupBy (fun c -> c.Country)
```

- ➊ Passing a function into Seq.filter to get USA customers
- ➋ Using an inline lambda function with Array.map
- ➌ Getting UK customers with Seq.filter and pipeline operator

The second set of function calls are essentially the same as the first except we've "flipped" the final argument over to the left of the pipe. It's also really important to be able to read and understand the function signatures in intellisense – so we'll spend a bit of time in the next lesson going through some common collection functions so that you gain the skills to figure out the other ones yourself.

### LINQ and F#

Note that you *can* use the standard LINQ functions in F# – simply open the `System.Linq` namespace and all the extension methods will magically appear on any collection. But I'd strongly urge you to favour F#'s collection libraries – they're designed specifically with F#'s type system in mind and usually lead to more succinct and idiomatic solutions. F# also has a `query { }` construct which allows use of IQueryables – have a read of them yourself on MSDN; they're extremely powerful.

### 15.1.3 Transformation pipelines

With the collection modules in mind, let's return to our challenge. Approaching this problem with a functional style needs a slightly different approach – we'll first try to identify some simple, isolated functions that we can quickly create, and *only then* look to compose them together using some reusable *higher order functions*. Let's start by thinking about *what* it is we want to do, rather than the "how" that we focused on before.

1. Find all results that had an away win.
2. Group all the away wins by the away team.
3. Sort the results in descending order by the number of away wins per team.

In order to build that, we first need to answer the question, "What is an *away win*?" That's easy – whenever the Away Team scores more goals than the Home Team. Let's can create a simple function for that, and then build up a pipeline using the `List` module: –

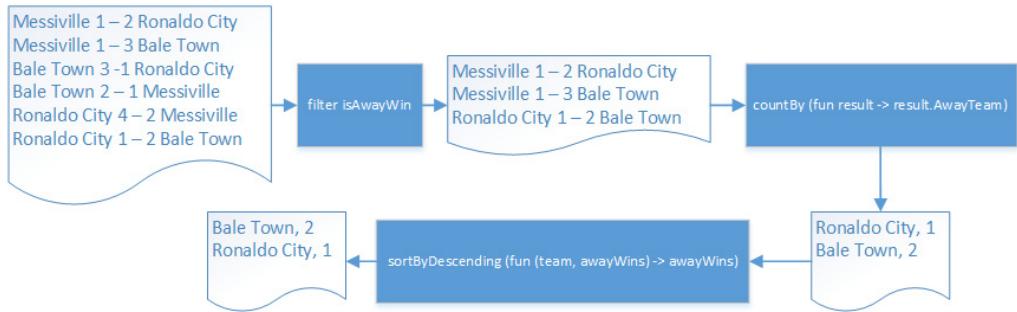
### **Listing 15.4 A declarative solution to a calculation over data**

```
let isAwayWin result = result.AwayGoals > result.HomeGoals ➊

results
|> List.filter isAwayWin ➋
|> List.countBy(fun result -> result.AwayTeam) ➌
|> List.sortByDescending(fun (_, awayWins) -> awayWins)
```

- ➊ A standalone function to calculate if a result is an away win
- ➋ Using `isAwayWin` within the `List.filter` HOF
- ➌ Using `countBy` with an inline lambda expression to return the number of rows for each away team

I find it helps to think of a transformation pipeline like that of above (and this also works for when working with composed functions operating over a single object) is to view it as a set of “dumb machines” that simply take in some set of data and give back a new one.



**Figure 15.2 – Visualising our transformation pipeline in terms of distinct stages**

There are some interesting properties of this pipeline: -

1. All stages are composed together with simple functions and pipelines. We could easily add in a new stage if we wanted to in the middle.
2. Each operation is a *pure* function that is completely decoupled from the overall pipeline. We can easily test out e.g. `isAwayWin` in isolation on a single, dummy result to ensure it works properly. We could also reuse it in any number of other pipelines or sections of code – it’s not “baked in” to the overall query we’re carrying out.
3. Each stage does not affect the input collection. We can repeat any stage a million times and it will always give the same result.

Not only is the code much, much smaller, and much more readable – it’s also much less likely to have any bugs, because we’re deferring probably 90% of the code that we wrote earlier to a set of general purpose higher order functions – filter, grouping, ordering etc. – and simply varying them by passing in an appropriate bit of code; the trick for us is to learn the most common higher order functions so that they become second nature.

Also – observe that the three functions we’re using – `filter`, `countBy` and `sortByDescending` all follow the same function signature as identified earlier, taking in a varying function and an input collection, and returning a *new collection*. In fact, you’ll often find that pipelines follow three stages: -

1. Create a collection of some sort.
2. Once you’re “inside” the collections world, you can perform one or many transformations on them. You never have to check if the collection is null or empty as the collections functions do that for you.

3. You end up with a final collection, or perform some aggregation to "leave" the collections world e.g. sum, average, first, last etc.



Figure 15.3 – Typical stages for a collections pipeline

#### 15.1.4 Debugging pipelines

##### **Now you try**

A side effect of these properties, and F#'s REPL, is that we can opt to simply execute *part* of the pipeline and check the output of the pipeline at that stage. This is particularly useful if you have a complex pipeline and aren't getting the correct results at the end – you can execute the pipeline repeatedly, each time going a little further, until you find the error.

1. To make life easier, before executing each of the next steps, clear the FSI output by right clicking over FSI and choosing **Clear All (not Reset!)**
2. In the REPL, with the code from Listing 15.4 at the ready, simply execute the first line of the pipeline (`results`) using **ALT + ENTER**. You'll see all six results sent to FSI.
3. Repeat the process, but this time highlight two lines, so that you execute both `results` and the `filter`.
4. Do the same again to include the `countBy`.

```

results
|> List.filter isAwayWin
|> List.countBy(fun result -> result.AwayTeam)
|> List.sortByDescending(fun (_, awayWins) -> awayWins)

F# Interactive >
> val it : (string * int) list = [("Ronaldo City", 1); ("Bale Town", 2)]
>

```

Figure 15.4 – Executing a subset of a pipeline for debugging and exploratory purposes.

5. As you execute each “subset” of the pipeline, building up to the end, compare the results with that of Figure 15.2.

### 15.1.5 Compose, compose, compose

I believe that one of the most common mistakes people make when looking at operations like this is when it comes to implementing a solution, people take a top-down approach i.e. trying to implement loops, manual filters and sorts etc. over an entire dataset.

Very often you'll end up with a far, far more effective solution by taking a *bottom-up* approach. Solve the *simple* parts of the problem first by writing small, easy-to-reason about functions, and then see how you can plug them together and reuse them as higher order functions. Look at the above example – we answered the question of *what* our filter is, but not *how* to perform the filter itself - that was delegated to the `Seq.filter` function.

If you ever find yourself writing a function which itself takes in a collection and manually iterates over it, you're probably doing extra work yourself – as you'll see over the remainder of these lessons.

#### Quick Check

1. What are the three main collection modules in F#?
2. Why is the input collection the last argument to collection functions?
3. What are some of the problems with processing collections imperatively?

## 15.2 Collection Types in F#

Let's now take a look in a little more detail on the three collection types – Sequences, Arrays and Lists.

### 15.2.1 Working with sequences

F# has a number of different collection types, the most common of which is `seq` (short for sequence). Sequences are effectively an alias for the `IEnumerable<T>` type in the BCL, and for the purposes of this lesson you can consider them interchangeable with LINQ-generated sequences, in that they are lazily-evaluated and (by default) do not cache evaluations. Also,

as Arrays and F# Lists implement `IEnumerable<T>`, you can use functions in the `Seq` module over both of them as well.

You can create sequences using the `seq { }` syntax, but in my experience this isn't needed that often, so I'm going to skip over it. Instead, focus on thinking about the `Seq` module to consume existing `IEnumerable` values etc.

### 15.2.2 Using .NET arrays

We actually looked at .NET arrays in one of the very first lessons. Like C#, F# has language syntax for arrays. However, F# syntax is much more lightweight, and F# also has a nice "slicing" syntax to allow you to extract a subset of an array.

#### **Listing 15.5 Working with .NET Arrays in F#**

```
let numbersArray = [| 1; 2; 3; 4; 6 |] ①
let firstNumber = numbersArray.[0] ②
let firstThreeNumbers = numbersArray.[0 .. 2] ③
numbersArray.[0] <- 99 ④
```

- ① Creating an array using `[| |]` syntax
- ② Accessing an item by index
- ③ Array slicing syntax
- ④ Mutating the value of an item in an array

You can also iterate over arrays using `for ... do` syntax as per sequences. Remember that Arrays are just standard BCL Arrays. They are high performance, but ultimately mutable (although you can safely rely on the `Array` module functions to create new arrays on each operation).

#### **Collection separators in C# and F#**

Watch out! In C#, we separate items in an array with the comma e.g. `new [] { 1, 2, 3 }`, but as in F#, the comma is used to create tuples - we use the *semi-colon* to separate items in an array, sequence or list e.g. `[ 1; 2; 3 ]`. If you use commas, you won't get a compile time error, as this is valid F#. Instead, you'll simply end up with a single tupled item! You can alternatively create a collection by placing each element on a new line, in which case you can omit the semi-colon separator entirely.

### 15.2.3 Immutable Lists

F# Lists (not to be confused with the `System.Collections.Generic.List<T>` aka `ResizeArray`) are native to F#. They work in a similar manner to Arrays in that they are eagerly evaluated and you can index into them directly, but have one key difference – F# Lists are immutable. This means that once you create a list, you cannot add or remove items from it (and if the data inside the list is immutable, it's entirely fixed) – instead we create new lists based on existing lists using *F# language syntax* for lists.

Internally, F# lists are linked lists – so it's very quick to create a new list with e.g. a single new item at the front of the list. Let's have a quick look at some F# list syntax - try working through this sample one line at a time so that you can see the result of each operator.

#### **Listing 15.6 Working with F# Lists**

```
let numbers = [1; 2; 3; 4; 5; 6] ①
let numbersQuick = [1 .. 6] ②
let head :: tail = numbers ③
let moreNumbers = 0 :: numbers ④
let evenMoreNumbers = moreNumbers @ [7 .. 9] ⑤
```

- ① Creating a list of six numbers
- ② Shorthand form of list creation (also valid on arrays and sequences)
- ③ Decomposing a list into head (1) and a tail (2 .. 6)
- ④ Creating a new list by placing 0 at the front of numbers
- ⑤ Appending moreNumbers and [ 7 .. 9 ] together to create a new list

You'll see that F# has some more "special" operators for working with lists: -

1. Create new lists using the [ a; b; c ] syntax
2. Deconstruct a list into a single item (head) and remainder (tail) with the :: operator
3. Place a single item at the front of a list using the :: operator
4. Merge two lists together using the @ operator.

You'll see a warning for the third expression, mentioning something about *pattern matching*. Don't worry about this for now – we'll come back to this in the next unit.

In addition to these language features for working with lists, you have the entire `List` module at your disposal to perform all manner of useful functions on it, such as sorting, filtering, etc. etc. To be honest, most of this functionality is achievable using the `List` module but it's sometimes useful to use these operators, particularly as it can keep code succinct. Don't be surprised if you recoil at this initially – once you learn the operators (and it's really only :: and @) you'll be fine.

#### **15.2.4 Comparing and Contrasting Collections**

As you can use all three collections almost interchangeably, it's sometimes difficult to know when to use what one. Here's a handy table that quickly distinguishes the features of them: -

**Table 15.1 – Comparing F# Sequences, Lists, and Arrays**

|              | <u>Seq</u> | <u>List</u> | <u>Array</u> |
|--------------|------------|-------------|--------------|
| Eager / Lazy | Lazy       | Eager       | Eager        |
| Forward-only | Sometimes  | Never       | Never        |
| Immutable    | Yes        | Yes         | No           |

|                          |        |               |        |
|--------------------------|--------|---------------|--------|
| Performance              | Medium | Medium / High | High   |
| Pattern Matching Support | None   | Good          | Medium |
| Interop with C#          | Good   | Medium        | Good   |

Note that performance is a more complex area – as always, your mileage may vary depending on the context e.g. you can add to the front of a List quickly, but not necessarily to the tail; sequences have a `Seq.cache` function that can be used to avoid repeated evaluation etc. Also, we haven't looked at pattern matching yet – so just bear that row in mind!

### Quick check

4. How does `seq` relate to `IEnumerable<T>?`
5. How do higher order functions relate to collection pipelines?
6. What are the main differences between an imperative and functional approach to working with collections?

## 15.3 Summary

In this lesson, we learned about processing collections in a functional style, and the benefits that we can gain from doing this.

- We explored the three core F# collections
- We learned about functional collection pipelines
- We saw a few operations that we might often perform on collections
- We saw what immutable lists are

In the next lessons, we'll build on this knowledge and gain confidence in working with collections by working through some common operations and functions.

### Try This

Find an existing LINQ query that you've written over an in-memory dataset; try to convert it to an equivalent Seq pipeline. Or, find an existing query you've written in an imperative style; try to rewrite it to a query pipeline using a set of chained Seq functions.

### Quick Check Answers

1. Seq, Array and List.
2. This allows easy pipelining of multiple operations through currying.
3. Difficult to compose behaviours; hard to reason about.
4. `seq` is effectively an F# alias for `IEnumerable`, and all functions in the `Seq` module can operate over `IEnumerables`.
5. We use higher order functions to “vary” collection operations which are then chained together to form more complex functionality.

6. Imperative routines favour modifying collections "in place", whereas a functional approach creates new collections for each stage of a pipeline. Imperative routines tend to merge all logic together whereas a functional approach tends to view operations as distinct stages which feed into one another.

# 16

## *Useful collection functions*

**Now that you have a reasonable high-level understanding of collections in F#, this lesson will focus on getting your “muscle memory” trained to using collections in practical situations.**

- We'll cover the most common collection functions across the three types that we've learned about (Seq, List and Array) with some visualisations and hands-on examples
- We'll try to compare with similar LINQ operations
- We'll further illustrate the differences of imperative and declarative solutions
- We'll also talk about moving between collection types

Each operation we cover will have a simple example associated with it, alongside some typical use cases and equivalents in both imperative coding and LINQ (if it exists). Go through every example in a script yourself rather than simply reading them – you'll only learn these effectively through practical experience. After that, I'll also point out some other, related functions in the collections libraries (denoted by *see also*) that you should look at in your own time.

A quick note – just like LINQ, most of the methods in F# collections operate on empty collections without a problem – you'll simply get back an empty collection again.

### **16.1 Mapping Functions**

Mapping functions take a collection of items, and return *another* collection of items. Usually the mapping can be controlled in some way by the caller, but there are some specialised forms of mapping here as well.

#### **16.1.1 Map**

The most common collection function you'll ever use is `map`. `Map` converts all the items in a collection from one *shape* to another shape, and always returns the same number of items in

the output collection as were passed in. At the risk of repeating myself, it's crucial to learn how to understand the signatures of collection functions. Here's the signature for `List.map`:

```
mapping:('T -> 'U) -> list:'T list -> 'U list
```

- **mapping** is a function that maps a *single item* from '`T`' to '`U`'
- **list** is the input list of '`T`' that you wish to convert
- The **output** is a list of '`U`' which has been mapped



Figure 16.1 – Mapping from a Person list to a String list

The direct equivalent to this in terms of LINQ is `Select()`. A common approach to performing the same operation with a loop is to first manually create an empty output collection, write a `for` loop to iterate over the collection, and manually populate the output collection with the output of every mapped item.

#### **Listing 16.1 Map**

```
let numbers = [1 .. 10] ①
let timesTwo n = n * 2 ②

let outputImperative = ResizeArray() ③
for number in numbers do
 outputImperative.Add (number |> timesTwo)

let outputFunctional = numbers |> List.map timesTwo ④
```

- ① Input data
- ② Mapping function
- ③ Manually constructing an output collection, iterating and adding to output
- ④ Using the `List.map` higher-order function to achieve the same output

You can use `map` for most use cases where the number of input and output elements are the same, for example loading a set of customers from a list of customer ids, or parsing a set of strings to decimals. Variants of `map` include `map2`, which works by combining two lists of the same type into a new merged list, and `mapi`, which includes an index item along with the item itself – useful for when you need to know the index of the item.

*See also: map2, map3, mapi, mapi2, indexed*

### Tuples in higher order functions

F# collection functions make extensive use of tuples as a lightweight way to pass pairs or triples of data items around. F# allows you to “unpack” tuples within lambda expressions directly in within a higher order function, so the following code is perfectly valid:-

```
["Isaac", 30; "John", 25; "Sarah", 18; "Faye", 27]
|> List.map(fun (name, age) -> ...)
```

The key part here is the lambda expression in the `map` call, where the function takes in `name` and `age`. This is a form of *pattern matching*, which automatically deconstructs the object passed in into its constituent parts.

### 16.1.2 Iter

`Iter` is essentially the same as `map`, except the function that you pass in *must return unit*. This is useful as an “end function” of a pipeline, such as saving records to a database or printing records to the screen – in effect, any function that is side-effectful.

```
action:('T -> 'unit) -> list:'T list -> unit
```

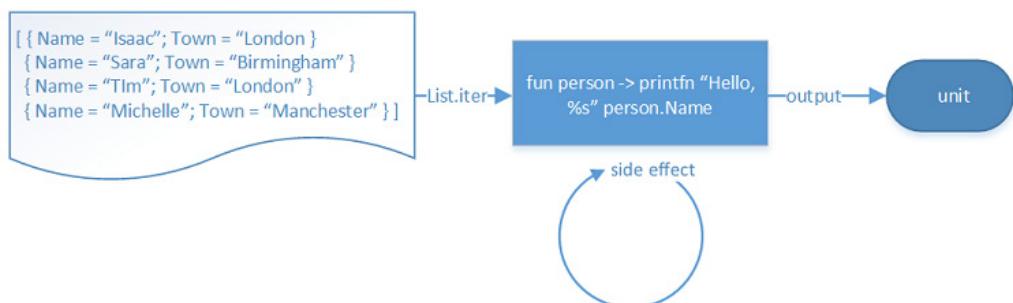


Figure 16.2 – Printing the name of a collection of customers.

In this example, we’re operating over each record and printing out a string to the console. This is a side effect, and there’s no tangible output for each output – just `unit`. There is no like-for-like equivalent to `iter` in LINQ, but you can achieve the same functionality using a basic `for-each` loop (or `for ... in` loop in F#). Compare the signature of this function with `map` and see where the difference is.

See also: `iter2`, `iter3`, `iteri`, `iteri2`

### 16.1.3 Collect

`Collect` is a useful form of `map` (in fact, it’s the other way around, as you can implement `map` through `collect` but not the other way around!) which has many other names including `SelectMany`, `FlatMap`, `Flatten` and even `Bind`. It takes in a list of items, and a function that

returns a *new collection* from each *item in that collection* – and then merges them all back into a *single list*. Sounds confusing, right? Let’s take a look at the `collect` signature first:

```
mapping:('T -> 'U list) -> list:'T list -> 'U list
```

Now compare it to `map`:

```
mapping:('T -> 'U) -> list:'T list -> 'U list
```

See the subtle difference? `Collect` says that the mapping function must return a *list*, rather a *single value*. Here’s an example – let’s say that we have a list of customers, and each customer has a list of orders. Let’s also assume that we already have a function called `loadOrders`, which takes in a `Customer` and returns the orders for that customer (`Customer -> Order list`). You want to retrieve all of orders for customers 1, 2, and 5 as a *single list*.

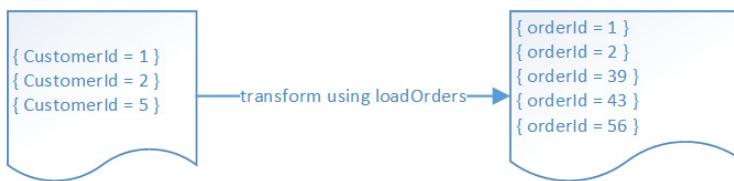


Figure 16.3 - A prime candidate for a collect operation

We have a function to load all the orders for a single customer, but how can we use that function to build up a *merged* set of orders? Unfortunately, if you try to load orders for each customer using `map`, you’ll end up with the following dataset: -



Figure 16.4 – Calling map against a function which returns a list

The signature of the result of this operation would be `Order list list` (or in C# terms, `List<List<Order>>`) – each function call itself returns a list, so you end up with a list of lists! That’s not what we want. This is where `collect` comes in – it expects the higher-order function to return a *collection* which it calls on each item, just like `map`, but the difference is that it merges *all* the items into a *single list*, as per Figure 16.3.

**Listing 16.2 Collect**

```
type Order = { OrderId : int }
type Customer = { CustomerId : int; Orders : Order list; Town string }
let customers : Customer list = []
let orders : Order list = customers |> List.collect(fun c -> c.Orders) ①
```

- ① Collecting all orders for all customers into a single list

Use `collect` to resolve “many-to-many” relationships, so that you can treat all “sibling” children as a single concatenated list.

**16.1.4 Pairwise**

`Pairwise` takes a list, and returns a new list of *tuple pairs* of the original adjacent items.

```
list:'T list -> ('T * 'T) list
```



Figure 16.5 – Pairwise operation on a list of numbers

This example shows a list of numbers, but can be equally applied to any list of objects that you wish to show as adjacent items. There are many times when pairwise operations are useful, such as calculating the “distance” between a list of ordered items such as dates – first, pairwise the elements, and then map the items.

**Listing 16.3 Using pairwise within the context of a larger pipeline**

```
open System
[DateTime(2010,5,1); DateTime(2010,6,1); DateTime(2010,6,12); DateTime(2010,7,3)] ①
|> List.pairwise ②
|> List.map(fun (a, b) -> b - a) ③
|> List.map(fun time -> time.TotalDays)
```

- ① A list of dates
- ② Pairwise for adjacent dates
- ③ Subtracting the dates from one another as a `TimeSpan`
- ④ Return the total days between the two dates

Several variations of this function exist; these all have slightly different signatures but perform similar functions – splitting a collection into collections of at least n elements, or n smaller collections etc. etc.

See also: `windowed`

### Quick Check

1. What is the F# equivalent of LINQ's `Select` method?
2. What is the imperative equivalent to the `iter` function?
3. What does the `pairwise` function do?

## 16.2 Grouping functions

As the name suggests, grouping functions perform some sort of logical grouping of data.

### 16.2.1 GroupBy

`GroupBy` works exactly as the LINQ version does, except the type signature is much simpler to read than the LINQ equivalent:

```
projection: ('T -> 'Key) -> list: 'T list -> ('Key * 'T list) list
```

The projection function simply returns a "key" that we group all the items in the list on. Note that the output is a collection of *simple tuples* – the first element of the tuple is the *key*, and the second element is the *collection of items in that group*. We don't need a custom type for the key/value pairing (such as the confusing `IEnumerable<IGrouping< TKey, TSource>>` in the LINQ implementation). Also note that each version of `groupBy` (`Seq`, `Array` and `List`) ensures that each grouping will be returned in the same type of collection – so groups in `Seq.groupBy` will be lazily evaluated, but `Array` and `List` will not be etc.

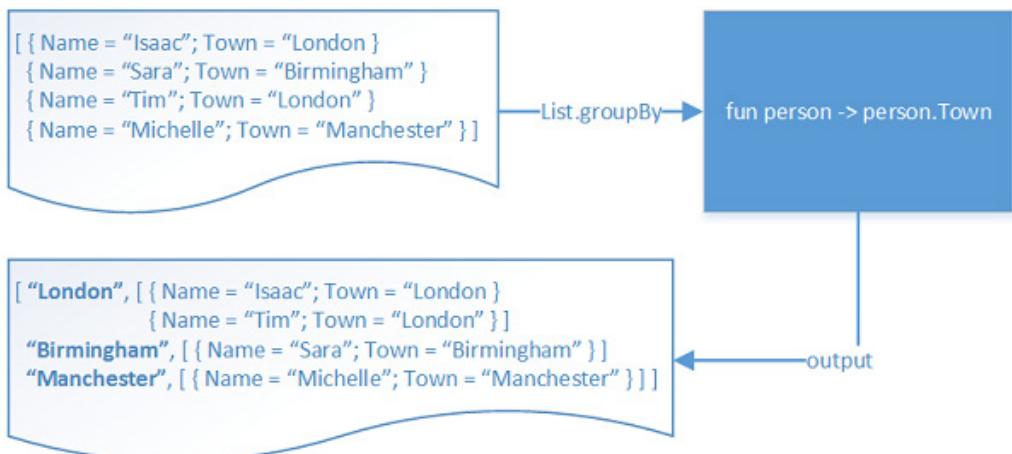


Figure 16.6 – Grouping a set of customers by Town

## 16.2.2 CountBy

A useful derivative of `groupBy` that exists, called `countBy`. This has a similar signature, but instead of returning the items in the group, it simply returns the *number of items* in each group.

```
projection: ('T -> 'Key) -> list: 'T list -> ('Key * int) list
```

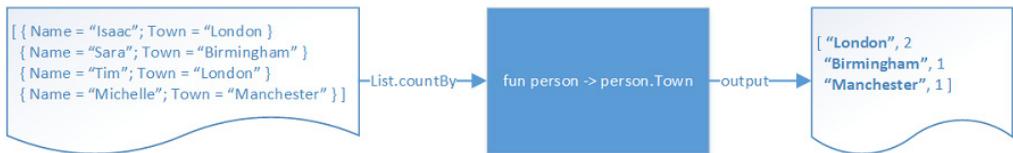


Figure 16.7 – Counting customers by town

## 16.2.3 Partition

`Partition` is a slightly simpler version of `groupBy`. You supply it a *predicate* (a function which returns true or false) and a collection; it returns two collections, partitioned based on the predicate.

```
predicate: ('T -> bool) -> list: 'T list -> ('T list * 'T list)
```

### Listing 16.4 Splitting a collection in two based on a predicate

```
let londonCustomers, otherCustomers = ①
 customers |> List.partition(fun c -> c.Town = "London") ②
```

- ① Decomposing the tupled result into the two lists
- ② Predicate function to split the list

Note that `partition` always splits into *two* collections – this restriction means that we can safely deconstruct the output directly into the two collections as per Listing 16.2. If there are no matches for either “half” of the split, an empty collection is returned for that half.

See also: `chunkBySize`, `splitInto`, `splitAt`

### Quick Check

4. When would you use `countBy` compared to `groupBy`?
5. Why would you use `groupBy` as opposed to `partition`?

## 16.3 More on collections

### 16.3.1 Aggregates

There are many aggregate functions in both the F# collections and LINQ libraries. They all operate on a similar principle: take a collection of items, and merge them into a smaller collection of items – often just one. Generally, you’ll find that aggregate functions are the last collection function in a pipeline. Here are some examples of aggregate functions in F# - you’ll probably be familiar with these functions already.

#### **Listing 16.5 Simple aggregation functions in F#**

```
let numbers = [1.0 .. 10.0] ①
let total = numbers |> List.sum ②
let average = numbers |> List.average
let max = numbers |> List.max
let min = numbers |> List.min
```

- ① Build a list of 10 floats
- ② Executing a set of aggregate functions

All of these functions are actually just specialized versions of a more generalised function called `fold`. In LINQ, it’s called **Aggregate()**. We’ll be looking at `fold` in more detail in the next lesson, as it has many applications aside from just summing numbers together.

### 16.3.2 Miscellaneous functions

This section covers a whole bunch of miscellaneous functions. Many of them have very similar LINQ equivalents, so you’ll probably already know them.

**Table 16.1 – Comparing miscellaneous functions**

| F#       | LINQ        | Comments                                                                                                                                                                                                                                                                             |
|----------|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| find     | Single()    | Equivalent to the Single() overload that takes a predicate; see also <code>findIndex</code> , <code>findBack</code> and <code>findIndexBack</code> .                                                                                                                                 |
| head     | First()     | Returns the first item in the collection; see also <code>last</code> .                                                                                                                                                                                                               |
| item     | ElementAt() | Gets the element at a given index.                                                                                                                                                                                                                                                   |
| take     | Take()      | The F# <code>take</code> implementation throws an exception if there are insufficient elements in the collection; use <code>truncate</code> for equivalent behavior to LINQ’s <code>Take()</code> . See also <code>takeWhile</code> , <code>skip</code> and <code>skipWhile</code> . |
| exists   | Any()       | See also <code>exists2</code> .                                                                                                                                                                                                                                                      |
| forall   | All()       | See also <code>forall2</code> .                                                                                                                                                                                                                                                      |
| contains | Contains()  |                                                                                                                                                                                                                                                                                      |

|          |            |                                                                                                                      |
|----------|------------|----------------------------------------------------------------------------------------------------------------------|
| filter   | Where()    | See also <code>where</code> .                                                                                        |
| length   | Count()    |                                                                                                                      |
| distinct | Distinct() | See also <code>distinctBy</code> .                                                                                   |
| sortBy   | OrderBy()  | See also <code>sort</code> , <code>sortByDescending</code> , <code>sortDescending</code> and <code>sortWith</code> . |

### Trying with collections

You might have noticed a whole bunch of functions that start with `try` e.g. `tryFind`, `tryHead` etc. etc. – these are equivalent to the “Default” methods in LINQ such as `FirstOrDefault()`. However, the F# equivalents all return `option` types. What are options? For now, think of these as `Nullable<T>`, although that’s not quite accurate. We’ll find out more about the `Option` type in the next unit.

### 16.3.3 Converting between collections

Occasionally, you’ll need to convert between lists, arrays and sequences – perhaps you have a function that returns an array and want to pipe the results into another function that expects a list, or you’re working with a collection that has specific performance characteristics best suited to an eager array than a lazy sequence. Therefore, each module has functions to easily convert to and from each collection type.

#### Listing 16.6 Converting between lists, arrays and sequences

```
let numberOne =
 [1 .. 5] 1
 |> List.toArray 2
 |> Seq.ofArray 3
 |> Seq.head
```

- 1 Construct an int list
- 2 Convert from an int list to an int array
- 3 Convert from an int array to an int sequence

As you can see, there are functions in all three modules that begin with `of` or `to` e.g. `ofList`, `toArray` etc. that perform the appropriate conversion.

#### Quick Check

6. What is the F# equivalent to LINQ’s `Aggregate` method?
7. What is the F# equivalent to LINQ’s `Take` method?
8. Give two reasons why you might need to convert between collection types in F#

## 16.4 Summary

That’s about it for this lesson! I hope you were able to see that the F# collection modules have a wide range of powerful functions – at the risk of repeating myself, go through all these

examples yourself, executing all the pipelines incrementally – execute just the first line, then the first two lines etc. and observe the outputs in FSI.

It's tempting to fall back to using LINQ for collection operations (indeed, there's nothing to stop you doing this) but you'll find (especially as we delve into more advanced features in the coming chapters) that the F# collection modules offer a much better fit for your code. In this lesson: -

- We saw a number of common collection functions that are commonly used in F#
- We saw how they relate to LINQ's set of collection functions
- We saw how F#'s native Tuple syntax can make the equivalent function signatures much simpler and more consistent than the LINQ equivalents

### **Try This**

Write a simple script that, given a folder path on the local file system, will return the name and size of each sub-folder within it. Make use of `groupBy` to group up files by folder, before using an aggregation function such as `sumBy` to total up the size of files in each folder. Then, sort the results by descending size. Then, enhance the script to return a proper F# Record which contains the Folder name, size, number of files, average file size and the distinct set of file extensions within the folder.

### **Quick Check Answers**

1. `map` is the equivalent to LINQ's `Select()` extension method.
2. For-each loops are the imperative equivalent to `iter`.
3. `pairwise` takes a collection of items and returns a new collection with the items windowed together in pairs.
4. `countBy` returns the number of elements per group; `groupBy` returns the actual elements themselves.
5. `groupBy` can partition a collection into an infinite number of groups; `partition` always splits a group into exactly two groups.
6. `fold`
7. `truncate (not take!)`
8. Performance reasons, or to match the type signature of a function that you are calling.

# 17

## Maps, Dictionaries and Sets

This lesson should be a fairly easy one as we round off the collection types in F#. We've so far looked at collections that model ordered elements of data in some way – Sequences, Lists and Arrays – which behave similar to the BCL List or `IEnumerable` types. We'll now spend a little bit of time looking at using other collection types in F#. We'll cover: -

- Working with the standard Generic Dictionary in F#
- Creating an immutable `IDictionary`
- The F#-specific Map type
- The F#-specific Set type

### 17.1 Dictionaries

#### 17.1.1 Mutable dictionaries in F#

You almost certainly already know the `System.Collections.Generic.Dictionary` type from C# or VB .NET – it acts as a standard “lookup” collection, allowing fast retrieval of values based on a unique key. You’ll be happy to know that, just like the majority of the BCL, you can use this class out-of-the-box in F#.

##### **Listing 17.1 Standard Dictionary functionality in F#**

```
open System.Collections.Generic

let inventory = Dictionary<string, float>() ①

inventory.Add("Apples", 0.33) ②
inventory.Add("Oranges", 0.23)
inventory.Add("Bananas", 0.45)

inventory.Remove "Oranges" ③

let bananas = inventory.["Bananas"] ④
```

```
let oranges = inventory.["Oranges"] ⑤
```

- ① Creating a dictionary
- ② Adding items to the dictionary
- ③ Removing an item from the dictionary
- ④ Retrieving an item
- ⑤ Trying to access an item that does not exist – exception is raised

This functionality should be pretty familiar to you – the only thing to remember is that in F#, indexer properties are preceded by a dot e.g. `inventory.[ "Bananas" ]`. But we can use F#'s syntax to make life a bit easier: just like the the .NET Generic `List` (or `ResizeArray`), F# can infer the generic types of a `Dictionary`, using one of two syntaxes: -

#### **Listing 17.2 Generic type inference with Dictionary**

```
let inventory = Dictionary<_,_>() ①
inventory.Add("Apples", 0.33)

let inventory = Dictionary() ②
inventory.Add("Apples", 0.33)
```

- ① Explicit placeholders for generic type arguments
- ② Omitting generic type arguments completely

### **17.1.2 Immutable Dictionaries**

There's one issue with the standard `Dictionary` – it's mutable, so additions and removals to the dictionary happen in-place. In the world of functional programming, we prefer immutable types where possible - so F# has a nice helper function to quickly create an *immutable* `IDictionary`, called `dict`. As the object `dict` returns is immutable, you can't add and remove items to it – instead, you supply it up-front with a *sequence of tuples* that represent the key / value pairs, which then become the fixed contents of the dictionary for its lifetime.

#### **Listing 17.3 Creating an immutable IDictionary**

```
let inventory : IDictionary<string, float> =
 ["Apples", 0.33; "Oranges", 0.23; "Bananas", 0.45] ①
 |> dict ②

let bananas = inventory.["Bananas"] ③

inventory.Add("Pineapples", 0.85) ④
inventory.Remove("Bananas")
```

- ① Creating a (string \* float) list of our inventory
- ② Creating an `IDictionary` from the list
- ③ Retrieving an item
- ④ Trying to add or remove items – `System.NotSupportedException` thrown

This syntax is pretty lightweight and easy to use, and is especially useful for those situations where you create a lookup and never modify it again. Unfortunately, as you can see from

Listing 17.3, because it implements `IDictionary` but is actually immutable, it has methods on it that you mustn't call (**Add**, **Clear** and **Remove**), because they'll throw exceptions. That's not so nice – if the type is immutable, it shouldn't be offering us those methods! The solution to this is to use a completely different type – the F# Map.

### Quickly creating full Dictionaries

The standard `Dictionary` doesn't allow you to easily create one with an initial set of data as `dict` does. However, what it does allow you to do is pass in an `IDictionary` as the input – which lookups generated by `dict` implements! So you can work around this restriction by doing the following:

```
["Apples", 10; "Bananas", 20; "Grapes", 15] |> dict |> Dictionary
```

Nice!

### Quick Check

1. What sort of situations would you use a Dictionary for?
2. How does F# syntax simplify creating Dictionaries?
3. Why might you use the `dict` function in F#?

## 17.2 The F# Map

The F# Map is an immutable key/value lookup. Just like `dict`, it offers the ability to quickly create a lookup based on a sequence of tuples, but *unlike* `dict`, it only allows us to safely add / remove items using a similar mechanism to “modifying” records or lists – by copying the entries from the existing `Map` to a new `Map`, and then adding or removing the item in question. You can't add items to an existing `Map`.



Figure 17.1 – Creating a new Map from an existing map plus a new item

Let's see how this looks: -

### Listing 17.4 Using the F# Map lookup

```
let inventory =
 ["Apples", 0.33; "Oranges", 0.23; "Bananas", 0.45] ❶
 |> Map.ofList ❷
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/get-programming-with-f-sharp>

Licensed to Gang Li <lg.careercross@gmail.com>

```

let apples = inventory.["Apples"] ③
let pineapples = inventory.["Pineapples"] ④

let newInventory =
 inventory
 |> Map.add "Pineapples" 0.87 ⑤
 |> Map.remove "Apples" ⑥

```

- ① Creating a (string \* float) list of our inventory
- ② Converting the list into a Map for quick lookups
- ③ Retrieving an item
- ④ Retrieving an item that does not exist – KeyNotFoundException thrown
- ⑤ Copying the map with a new item added
- ⑥ Copying the map with an existing item removed

The nice thing with this approach is that we get all the usual benefits of immutability – repeatability and safety etc. without the need to give up the ability to (in effect) easily add or remove items to it. Importantly, calling `Add` on a `Map` that already contains the key will *not* throw an exception. Instead, it'll simply replace the old value with the new one as it creates the new `Map` (the original `Map` will of course retain the original value).

We can also *safely* access a key in a `Map` using `TryFind`. This doesn't return the value, but a wrapped `option`. We'll learn about Options in the next unit – but just keep in the back of your mind that Maps play nicely with them, too.

### 17.2.1 Useful Map functions

In addition to just `add` and `remove`, the `Map` module has some other useful functions which are similar in nature to those in the `List`, `Array` and `Seq` modules and allow you to treat Maps as though they were enumerable collections, using the same chained pipelines that you're used to, such as: -

- `map`
- `filter`
- `iter`
- `partition`

The main difference between the signature of these methods and the equivalents in the other modules is that the `Map` higher order functions take in both the key *and* the value for each element in the map, whereas e.g a `List` for example only takes in the value.

```

Seq.map: mapping ('T -> 'U) -> source:Seq<'T> -> Seq<'U>
Map.map: mapping ('Key -> 'T -> 'U) -> table:Map<'Key, 'T> -> Map<'Key, 'U>

```

#### **Listing 17.5 Using the F# Map module functions**

```

let cheapFruit, expensiveFruit = ①
 inventory
 |> Map.partition(fun fruit cost -> cost > 0.3) ②

```

- ➊ Two maps, partitioned on cost
- ➋ Partition higher order function that receives both key (fruit) and value (cost) as arguments

Note the key and value are not passed as a tuple but as a curried function, hence why `fruit` and `cost` are separated by space, and not a comma.

### **Now you try**

Let's create a lookup for all the root folders on your hard disk and the times that they were created.

1. Open up a blank script.
2. Get a list of all directories within the C:\ drive on your computer (you can use `System.IO.Directory.EnumerateDirectories`). The result will be a sequence of strings.
3. Convert each string into a full `DirectoryInfo` object. Use `Seq.map` to perform the conversion.
4. Convert each `DirectoryInfo` into a tuple of the `Name` of the folder and its `CreationTimeUtc`, again using `Seq.map`.
5. Convert the sequence into a `Map` of `Map.ofSeq`.
6. Convert the values of the `Map` into their age in days using `Map.map`. You can subtract the creation time from the current time to achieve this.

### **Dictionaries, Dict or Maps?**

Given these three lookup types, when should you use which? My advice is as follows: -

1. Use the `Map` as your default lookup type. It's immutable, and has very good support for F# tuples and pipelining.
2. Use the `dict` function to quickly generate an `IDictionary` that is needed for e.g interop with other code e.g BCL code. The syntax is very lightweight, and is easier to create than a full `Dictionary`.
3. Use `Dictionary` if you need a mutable dictionary, or have some block of code that has specific performance requirements. Generally, the performance of a `Map` will be fine, but if you're in a tight loop performing thousands of additions / removals to a lookup, a `Dictionary` will perform better. As always – optimise as needed, rather than prematurely.

### **Quick Check**

4. What is the main difference between a `Dictionary` and a `Map`?
5. When should you use a `Dictionary` over a `Map`?

## **17.3 Sets**

Sets are a somewhat rarely used collection type, which is a pity as they allow us to very elegantly create solutions to certain problems that would otherwise be several lines of code. As the name suggests, a `Set` implements a standard mathematical set of data – in other words: -

In mathematics, a set is a collection of distinct objects

[https://en.wikipedia.org/wiki/Set\\_\(mathematics\)](https://en.wikipedia.org/wiki/Set_(mathematics))

In other words – unlike other collections, Sets cannot contain duplicates, and will automatically remove repeated items in the set for you. F# Sets are trivial to use, as they follow the same pattern as the other collection types. Let's see how we might create a Set: -

### **Listing 17.6 Creating a Set from a Sequence**

```
let myBasket = ["Apples"; "Apples"; "Apples"; "Bananas"; "Pinapples"] ①
let fruitsILike = fruits |> Set.ofList ②

// val fruitsILike : Set<string> = set ["Apples"; "Bananas"; "Pinapples"] ③
```

- ① Input data
- ② Converting to a set
- ③ Evaluated output shown in FSI

Observe how fruitsILike has only the unique fruits from myBasket, without us needing to explicitly call Distinct etc. Where sets in F# are *really* useful is when we need to perform *set-based operations* on two sets. Let's assume we had two baskets of fruits and wanted to combine them to find fruits we both like. Let's compare two different approaches – first using List, and then Set: -

### **Listing 17.7 Comparing List and Set based operations**

```
let yourBasket = ["Kiwi"; "Bananas"; "Grapes"] ①
let allFruitsList = (fruits @ otherFruits) |> List.distinct ②

let fruitsYouLike = yourBasket |> Set.ofList ③
let allFruits = fruitsILike + fruitsYouLike ④
```

- ① Creating a second basket of fruits
- ② Combining the two baskets using @, then distinct
- ③ Creating Sets of the Lists
- ④ “Summing” two Sets together performs a Union operation

The first two lines should be obvious. The interesting line is the fourth and final expression fruitsILike + fruitsYouLike where we seem to be adding two sets together. Can we do this? Yes, we can! This is because the Set module includes operator overloads for addition and subtraction, which internally it redirects to Set.union and Set.difference. As a one-off operation, using Lists with distinct etc. might suffice, but if you're trying explicitly to model a Set with set based behaviours – unions, difference, subset etc., Set is a much more elegant fit – here are some more examples.

**Listing 17.8 Some sample Set based operations**

```
let fruitsJustForMe = allFruits - fruitsYouLike ①
let fruitsWeCanShare = fruitsILike |> Set.intersect fruitsYouLike ②
let doILikeAllYourFruits = fruitsILike |> Set.isSubset fruitsYouLike ③
```

- ① Get fruits in A that are not in B
- ② Get fruits that exist in both A and B
- ③ Are all fruits in A also in B?

What's nice is that although I'm using strings here, Sets work with any type that supports comparison – which F# Records and Tuples do by default. So, you might use Sets for finding out which products exist in both warehouses, or whether all customers that live in New York are also high value customers. Of course, Sets also have a number of standard functions such as map, filter etc. as well as transformers from / to Lists, Seqs and Arrays.

**Quick Check**

6. What function might you use to simulate simple Set-style behaviour in a List?

**17.4 Summary**

That's a wrap! You've now seen all the main different types of collections that you'll commonly be dealing with in F#. We reviewed four different types of collections: -

- Dictionaries
- IDictionaries through `dict`
- Maps
- Set
- We're almost finished with collections now – all that's left in the next lesson is seeing how we can take advantage of some of the more powerful functions in the collections libraries to push things to the limit.

**Try This**

Continuing from the previous lesson, create a lookup for all files within a folder, so that you can find the details of any file that has been read. Experiment with sets by identify filetypes in folders. What file types are shared between two arbitrary folders?

**Quick Check Answers**

1. Typically for fast key / value lookups that can mutate over time.
2. You can omit generic type arguments, and use the `dict` function to help create them.
3. For immutable dictionaries i.e. lookups that can never change.
4. Dictionary is a mutable lookup; Map is immutable, creating new maps after each operation.

5. When you need to maximize performance, or are modelling an inherently mutable dataset.
6. `Distinct` or `DistinctBy`.

# 18

## *Folding our way to success*

In the last few lessons, we've covered the main collection types and how to use them. What we'll round off with is a few scenarios when collections can be used in interesting ways to achieve outputs and transformations that you might not think possible through *folding*. We'll look at:-

- Understanding aggregations and accumulation
- Avoiding mutation through fold
- Building rules engines and functional chains

### 18.1 Understanding aggregations and accumulators

You'll already be familiar with some of the *aggregation functions* in LINQ or F# Collections, such as Sum, Average, Min, Max and Count. All of these have a common signature: they take in a sequence of elements of type T, and return a single object of type U.

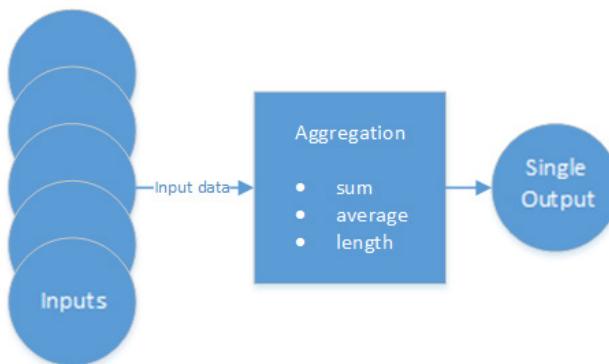


Figure 18.1 – High-level visualizing of aggregation

We can view this in terms of F# types as follows: -

### **Listing 18.1 Example aggregation signatures**

```
type Sum = int seq -> int ①
type Average = float seq -> float
type Count<'T> = 'T seq -> int
```

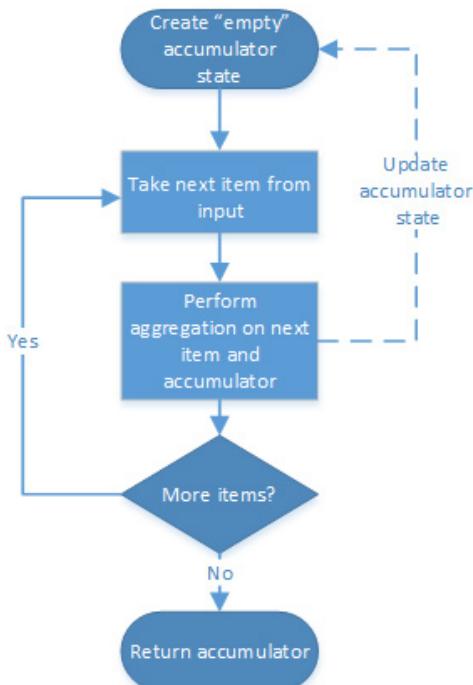
#### **① Some example types of Aggregation**

As you can see, the Sum, Average and Count functions all share a common theme – they take a *collection* of some “things” and returns some *single* other “thing”.

#### **18.1.1 Creating our first Aggregation function**

Let’s take a look at how we might implement the generic Sum aggregation – for example, calculating the sum of the numbers 1 to 5, or the total value of three customers, or total cost of 10 orders etc. Performing any aggregation, or *fold*, generally requires three things: –

- the input collection
- an accumulator to hold the “state” of the output result as it is built up
- an initial (“empty”) value for the accumulator to start with



**Figure 18.2 – How accumulators work imperatively**

So let's see how this might work for sum: -

### **Listing 18.2 Imperative implementation of Sum**

```
let sum inputs =
 let mutable accumulator = 0 ①
 for input in inputs do ②
 accumulator <- accumulator + input ③
 accumulator ④
```

- ① Empty accumulator
- ② Go through every item
- ③ Apply aggregation onto accumulator
- ④ Return accumulator

Interestingly, if you mouse over the `sum` function, you'll see that it has the exact signature described in Listing 18.1, i.e. `seq<int> -> int`. Once again, type inference helps out here by correctly determining that the `inputs` value is a collection (based on the `for` loop) and an `int` based on addition to `accumulator`.

### **Now you try**

Try to create aggregation functions using the above style for a couple of other aggregation functions.

1. Create a new .fsx script.
2. Copy the code from Listing 18.2
3. Try to create a function to calculate the *length* of a list (take any list from the previous lessons as a starting point!). The only thing that should change is the line that updates the `accumulator`.
4. Now do the same to calculate the *maximum* value of a list.

As you're by now well aware though, we're using a mutable variable here for our `accumulator`, as well as an imperative loop – not particularly composable. A standard answer from a functional programmer would be to rewrite this code using *recursion* – a style of programming where a function calls *itself* as a way of maintaining state. I'm not a massive fan of recursion personally – it can be quite difficult to follow, particularly if you come from an imperative background (see appendix 5 for a short example of it). As you'll see, the majority of the time you can get away without it; instead, we'll see an alternative, collection-based way to achieve the same code as above, without any mutation and accumulation.

### **Quick check**

1. What is the general signature of an aggregation?
2. What are the main components of any aggregation?

## 18.2 Saying hello to fold

Fold is a generalised way to achieve the exact sort of aggregations that we've just been looking at. It's a higher order function that allows us to supply an *input collection* we want to aggregate, a *start state* for the accumulator, and a function that says *how* to accumulate data. Let's look at the arguments signature for `Seq.fold`.

```
folder:('State -> 'T -> 'State) -> state:'State -> source:seq<'T> -> 'State
```

That's a relatively scary-looking signature, so let's break it down step by step: -

- **folder**: A *function* that is passed into fold that handles the accumulation e.g. summing, averaging, getting the length etc.
- **state**: The initial start state
- **source**: The input collection

Let's see what it looks like to implement sum using the `fold` function: -

### Listing 18.3 Implementing sum through Fold

```
let sum inputs =
 Seq.fold
 (fun state input -> state + input) ①
 0 ②
 inputs ③
```

- ① Folder function to sum the accumulator and input
- ② Initial state
- ③ Input collection

I've put the arguments on different lines here to make it somewhat clearer what each argument is. The key part is the **folder** function: it takes in the *current state* (accumulator) value and the *next item* in the collection; your responsibility is to calculate the *new state* from those two items. If we compare it to Listing 18.2, you'll see the same three key elements as there; the difference is that we don't have to explicitly store an accumulator value, or iterate over the collection. All of that is taken care of us by `fold` itself. Let's now add some logging so that we can see exactly what's going on.

### Listing 18.4 Looking at fold with logging

```
let sum inputs =
 Seq.fold
 (fun state input ->
 let newState = state + input
 printfn "Current state is %d, input is %d, new state value is %d" state input
 newState
 newState)
 0
 Inputs

sum [1 .. 5]
```

```

Current state is 0, input is 1, new state value is 1
Current state is 1, input is 2, new state value is 3
Current state is 3, input is 3, new state value is 6
Current state is 6, input is 4, new state value is 10
Current state is 10, input is 5, new state value is 15

```

- 1 Creating the new state
- 2 Debug message
- 3 Returning the new state

It might help to see this “threading” of state as a visual diagram: -

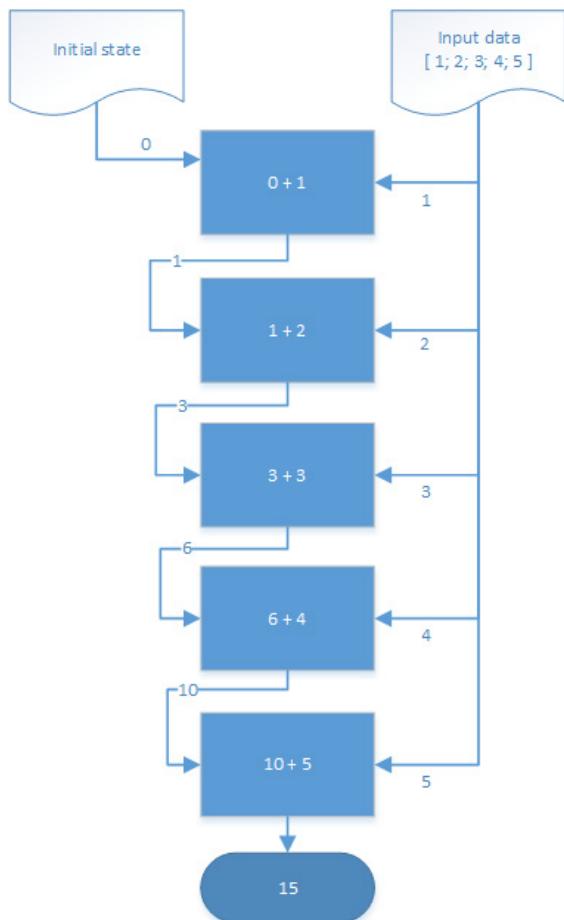


Figure 18.3 – Visualising how state is threaded through fold

## Now you try

Let's create a few aggregations of our own to improve our familiarity with `fold`.

1. Open up your script from earlier.
2. Try to implement a length function using `fold`.
3. Try to implement a max function using `fold`.

### Some examples of real-world use of aggregations

You probably use aggregations all the time without realising it – the trick is to spot the signature of “given a collection of items, we get back a single item”. Examples include: -

1. Retrieving the total price of a set of orders.
2. Merging together a collection of financial transactions in order to determine if a customer is high risk or not.
3. Aggregating a set of events in an event-driven system over some initial data.
4. Showing a single red / amber / green status on the dashboard of an internal website to show whether all back-end systems are functioning correctly.

### 18.2.1 Making fold more readable

One thing that I don't find especially nice is the way that the arguments to `fold` are laid out. They can be made more readable using one of two tricks: the pipeline operator, which you already know, and the rarely-used *double pipeline* operator – which acts the same as the normal pipeline, but takes in the last *two* arguments and moves them to the front as a *tuple*. Here are three ways of calling the same function: -

#### **Listing 18.5 Making fold read in a more logical way**

```
Seq.fold (fun state input -> state + input) 0 inputs
inputs |> Seq.fold (fun state input -> state + input) 0 ①
(0, inputs) ||> Seq.fold (fun state input -> state + input) ②
```

- ① Using pipe to move “inputs” to the left-hand side
- ② Using the double pipe to move both the initial state and “inputs” to the left-hand side

Using the double pipe helps me visualise folds a lot – the code now reads to me as: “Here's some initial state of 0 and a collection of input numbers. Fold them both through this function, and give me the answer”.

### 18.2.2 Related fold functions

In addition to the more specific aggregations such as `sum` and `average`, the F# collection library contains some variants on `fold`:

- **foldBack**: Same as fold, but goes backwards from the last element in the collection.
- **mapFold**: Combines map and fold – emits a sequence of mapped results and a final state.
- **reduce**: A simplified version of fold, using the first element in the collection as the initial state, so you don't have to explicitly supply one. Perfect for simple folds such as sum (although it will throw an exception on an empty input - beware!)
- **scan**: Similar to fold, but generates the intermediate results as well as the final state. Great for calculating running totals.
- **unfold**: Generates a sequence from a single starting state. Similar to the yield keyword.

### 18.2.3 Folding instead of while loops

What about times where we don't have an up-front collection of data? Perhaps we're waiting on some user input, or streaming data from a remote data source. Look at the following example, where I'm streaming data from a file and counting the number of characters in the file: -

#### **Listing 18.6 Accumulating through a while loop**

```
open System.IO
let mutable totalChars = 0 ①
let sr = new StreamReader(File.OpenRead "book.txt") ②

while (not sr.EndOfStream) do ③
 let line = sr.ReadLine()
 totalChars <- totalChars + line.ToCharArray().Length ④
```

- ① Initial state
- ② Opening a stream to a file
- ③ Stopping condition
- ④ Accumulation function

Obviously, there are easier ways to count characters in a file – the point is that we have some unknown “end” to this stream of data, rather than a fixed, up-front collection. How can we use `fold` here, which takes in a sequence of items as input? The answer is to *simulate* a collection using the **yield** keyword. Let's take a look: -

#### **Listing 18.7 Simulating a collection through sequence expressions**

```
open System.IO
let lines : string seq =
 seq { ①
 use sr = new StreamReader(File.OpenRead @"book.txt")
 while (not sr.EndOfStream) do
 yield sr.ReadLine() ②
 }

(0, lines) ||> Seq.fold(fun total line -> total + line.Length) ③
```

- 1 Sequence expression
- 2 Yielding a row from the StreamReader
- 3 A standard fold

The `seq { }` block is a form of *computation expression*. We won't talk too much about them in this book, but essentially a computation expression is a special block in which certain keywords, such as `yield`, can be used (there are others, as we'll see in Unit 8). Here, `yield` has the same functionality as in C# - it "yields" items to *lazily generate* a sequence. Once we've done that, we can fold over the sequence of strings, just like we did earlier.

### Quick check

3. What is the difference between `reduce` and `fold`?
4. Which two F# keywords are important in order to lazily generate sequences of data?

## 18.3 Composing functions with fold

The last element of `fold` that's worth at least looking at briefly is as a way to *dynamically compose* functions together – in other words, given a *list of functions* that have *the same signature*, give me a *single* function that runs all of them together. Let's take the example of a simple rules engine: we're writing a simple parser and want to validate that a supplied piece of text is valid: -

- Every string should contain three words
- The string must be no longer than 30 characters
- All characters in the string must be upper case

What we *don't* want to do is hard code the code that does the parsing and validation. Instead, we want to be able to supply a *collection* of rules and build them together to form a single rule – this way we can add new rules without affecting the main code base. Trying to do this with mutation and imperative loops can be a real pain, so in the interest of space I'm going to skip that out entirely (but feel free to try it out yourself!). What we'll look at first is how we'll model such a rules engine, and then how we can perform the "composition" element of it.

Let's start with some simple functions to represent our rules. We're not going to bother with interfaces or anything like this – instead, we'll define a simple function signature for a rule, which we'll also *alias* to a specific type name called `Rule` to make code a bit easier to read later on: -

```
type Rule = string -> bool * string
```

This signature says: give me some text as a `string`, and I'll give you back both a `Boolean` (passed or failed) and a `string` (the error message in case of failure). We can now use that signature to make a list of rules.

**Listing 18.8 Creating a list of rules**

```
open System
type Rule = string -> bool * string

let rules : Rule list = ①
 [fun text -> (text.Split ' ').Length = 3, "Must be three words" ②
 fun text -> text.Length <= 30, "Max length is 30 characters"
 fun text -> text
 |> Seq.filter Char.IsLetter
 |> Seq.forall Char.IsUpper, "All letters must be caps"]
```

- ① List definition
- ② All rules provided inline

**Type Aliases**

Notice the use of a **type alias** in 18.8 - Rule. Type aliases let us define a type signature that we can use *in-place* of another one. An alias isn't a new type – the definition it aliases is interchangeable with it, and the alias will be erased at runtime – it's just a way to improve documentation and readability. Note that the compiler won't know which signature to use, so intellisense can sometimes show the "full" type rather than the aliased one.

For a larger system, with more complex rules, you might want to put the rules into a module as proper let bound functions, and then simply create a list based on those functions. In this case though, I've simply defined the rules inline. Notice how we're able to simply create a tuple of a **function** with a specific signature and an **error message** – F# infers that `text` is of type `string` because we've said above that `rules` is a list of type `Rule`.

**18.3.1 Composing rules manually**

Given those three rules above, here's how you might manually "compose" all three rules into a single "super rule":

**Listing 18.9 Manually building a "super rule"**

```
let validateManual (rules: Rule list) word =
 let passed, error = rules.[0] word ①
 if not passed then false, error ②
 else
 let passed, error = rules.[1] word ③
 if not passed then false, error
 else
 let passed, error = rules.[2] word
 if not passed then false, error
 else true, ""
```

- ① Testing the first rule
- ② Checking if the rule failed
- ③ Rinse and repeat for all remaining rules

### 18.3.2 Folding functions together

The approach we've just seen obviously doesn't scale particularly well. An alternative approach is to create a function that when given a list of rules gives back a *new* rule that runs all the individual rules, using the `reduce` form of `fold`. We've not looked at `reduce` in detail yet, so refer back to 18.2.2 for an explanation of it if needed.

#### Listing 18.10 Composing a list of rules using reduce

```
let buildValidator (rules : Rule list) =
 rules
 |> List.reduce(fun firstRule secondRule ->
 fun word -> ①
 let passed, error = firstRule word ②
 if passed then ③
 let passed, error = secondRule word
 if passed then true, "" else false, error
 else false, error) ④

let validate = buildValidator rules
let word = "HELLO FrOM F#"

validate word

// val it : bool * string = (false, "All letters must be caps")
```

- ① Higher order function
- ② Run first rule
- ③ Passed, move on to next rule
- ④ Failed, return error

Just like all our other aggregations, this one follows a similar pattern (albeit in this case, 'T and 'U are the same). We can also “explode” the aliased types at the same time: -

```
Rule seq -> Rule
(string -> bool * string) seq -> (string -> bool * string)
```

If you're an OO design pattern expert, you might recognise this as the *composite pattern*. What we're essentially doing is the same as in Figure 18.3, but rather than the state being an integer, it's an *actual function* which, on each iteration, covers *another rule*. Notice also that the signature maps to our original Aggregation type: `Rule list -> Rule`.

In effect, this code says “given two rules and a word, check the word against the first. If it passes, then check against the second one”. This is itself returned as a *function* to `reduce` as a “composed” rule, which is then used in the next iteration.

#### Now you try

You might have found that last bit of code a little hard to “get” at first. Let's explore it a tiny bit so we can better understand what happened, before we finish this lesson.

1. Put in some `printfn` statements inside the rules themselves e.g. `printfn "Running 3-word rule..."` so that you can see what's happening here. You'll have to make each rule a multiline lambda to do this.
2. Try to move the rules into a separate module as let-bound functions.
3. Add a new rule to the collection of rules which fails if there are any numbers in the text (the `System.Char` class has some helpful functions here!).

### **Quick Check**

5. What OO pattern is equivalent to “reducing” functions together?
6. What happens to a type alias after compilation? Is it available at runtime?

## **18.4 Summary**

That's the end of collections! We've now looked at a load of different aspects of collections in F#. In this lesson in particular, we saw: -

- Aggregations
- Folding over collections
- Folding over sequences
- Dynamically creating rules engines

We'll be using collections throughout the rest of the book, and you'll be introduced to some more features of the collections module later – but for now, take a breath and get ready for the next unit!

### **Try This**

Create a simple rules engine over the file system example from the previous lesson. The engine should filter out files that do not pass certain checks, such as over a specific file size, having a certain extension or before created before a specific date. Have you ever created any rules engines before? Try rewriting them in the style we've defined above.

### **Quick Check Answers**

1. `seq<'T> -> 'U`
2. A collection to fold over, an accumulator to hold aggregation state, and a start state.
3. Reduce does not require a seed value – it uses the first item in the collection.
4. `seq` (to create a sequence block) and `yield` (to yield back values)
5. The Composite Pattern.
6. Type Aliases are erased at runtime and revert back to the real type that they alias.

# 19

## Capstone 3

**To round off this unit, we'll dive back into the Bank Accounts problem we worked on in Capstone 2, but this time enhance it with a few new features which will test out some of your knowledge of collections, as well as further reinforcing the lessons you picked up earlier in this book. We'll look at: -**

- Creating and working with sequences
- Performing aggregations
- Composing functions together
- Organising code in modules

### 19.1 Defining the problem

In this exercise, we'll start from a variant of the code that we ended up with at the end of Capstone 2 and enhance it step-by-step. Essentially, we designed a basic application to allow you to create an in-memory "bank account", and perform withdrawals and deposits into the account. Now we're going to continue that good work with a few enhancements: -

- Update our main command handling routine to eliminate mutable variables
- Store a serialized transaction log to disk for each customer
- Rehydrate historical transactions and build up an up-to-date account using sequence operations

#### 19.1.1 Solution overview

You'll see in the `src/lesson-19` folder is a pre-built `Capstone3.sln` solution for you to open. There's also a sample solution in the `sample-solution` subfolder. As always – use it if you get stuck, but don't use it as a starting point – the whole idea is for you to try to solve this yourself!

## 19.2 Removing mutability

The first thing we can do with our new-found knowledge of sequences is to remove the dependency on mutable variables for our main driver program. If you remember from Capstone 2, we had to rely on an imperative while loop for keeping track of the state of the account as we “modified” the account with deposits or transactions.

### 19.2.1 Comparing Imperative and Declarative flows

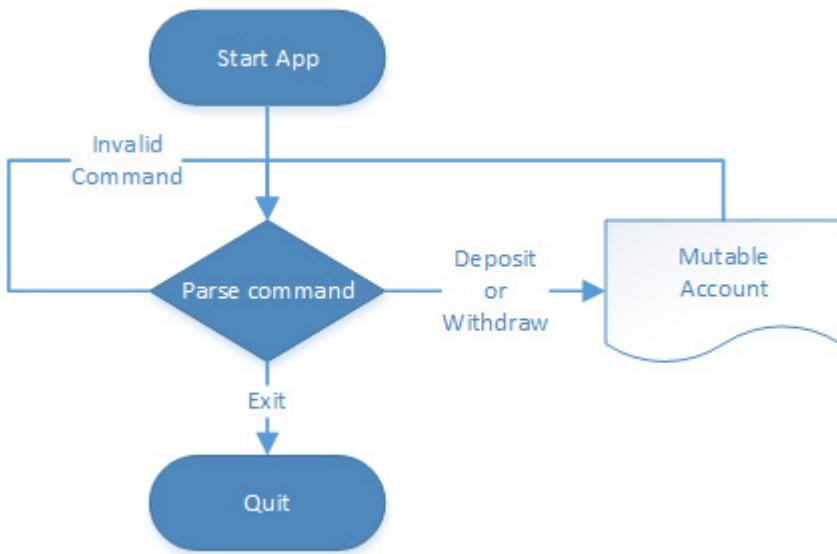


Figure 19.1 – Existing imperative main driver loop

Although having a mutable variable in a single, isolation place isn’t necessarily a problem, we can remove reliance on it without too much difficulty. Moving from an imperative to a declarative mode of thinking will also provide us with a more composable way of expressing this logic, and avoid branching logic. There are two ways of avoiding the imperative, mutable model we have: –

- Recursion, which I’ve deliberately avoided so far.
- The other is to treat the changes to the account as a *sequence* of operations which are applied against the *previous version* of the account; when the user decides to quit the application, the sequence stops and the final account version is the end state.

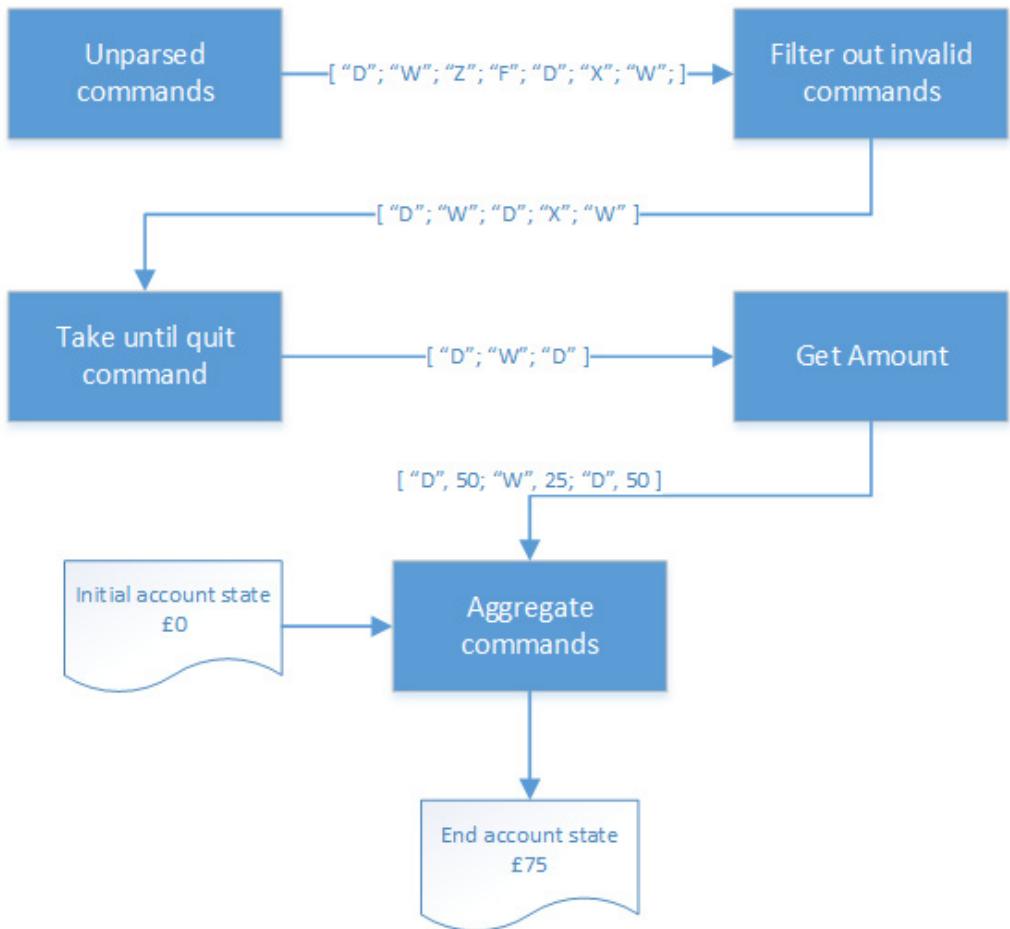


Figure 19.2 – Proposed declarative view of commands as a sequence

In other words, given a sequence of commands, we can filter out invalid ones, take until we receive a quit command, apply amounts onto the commands, and finally process them in sequence until we have our final account state.

The problem here, of course, is that our application is *interactive* – we’re asking the user for input here, rather than starting with a *predefined* set of commands. But that’s actually not too difficult to achieve as we’ll see shortly – so the easiest thing to do is to start with a pre-existing collection of commands in a script and try to build a pipeline that looks exactly as above, but in code.

**Listing 19.1 – Creating a functional pipeline for commands**

```
#load "Domain.fs"
let openingAccount =
 { Owner = { Name = "Isaac" }; Balance = 0M; AccountId = Guid.Empty } ①
let account =
 let commands = ['d'; 'w'; 'z'; 'f'; 'd'; 'x'; 'w'] ②

 commands
 |> Seq.filter isValidCommand
 |> Seq.takeWhile (not << isStopCommand)
 |> Seq.map getAmount
 |> Seq.fold processCommand openingAccount ④
```

- ① Initial opening account state
- ② Set of commands we wish to process
- ③ Aggregating the validated set of commands, using openingAccount as initial state

**Now you try**

Your task, should you choose to accept it, is to implement the functions used in the pipeline from Listing 19.1. The descriptions for each function is below, followed by stubs in Listing 19.2. Start by opening `Scratchpad.fsx` which has already been created for you and has appropriate `#load` statements to import the `fs` files which contain all required types.

1. `isValidCommand` – checks if the command is one of (`d`)eposit, (`w`)ithdraw, or e(`x`)it.
2. `isStopCommand` – checks if the command is the exit command
3. `getAmount` – takes in a command and converts it to a tuple of the command and also an amount. Your code should check: -
  - o If the command is deposit, then return ('`d`', 50M)
  - o If withdraw, return ('`w`', 25M)
  - o Otherwise, return the ('`x`', 0M)
4. `processCommand` – takes in an account and a (command, amount) tuple. It should then apply the appropriate action on the account and return the new account back out again.

**Listing 19.2 – Sample functions for command processing pipeline**

```
#load "Domain.fs"
#load "Operations.fs"

open Capstone3.Operations
open Capstone3.Domain
open System

let isValidCommand (command:char) = if command = 'w' then true else false
let isStopCommand (command:char) = false
let getAmount (command:char) = command, 0M
let processCommand (account:Account) (command:char, amount:decimal) = account
```

### Adapting functions to fit signatures

Notice that the arguments for `processCommand` uses a hybrid approach of curried and tupled form, so that it “plugs into” `Seq.fold` naturally. This is not uncommon to do, especially when working with higher order functions. You might ask when is the best time to “force” a function signature into a particular “shape” to fit with the caller – and there’s not always a great answer. In a way, it’s a similar question in the OO world of changing the signature of a class to fit into an existing interface – sometimes you’ll do it, other times you’ll use an adapter. In F#, an adapter over a function is simply a lambda (or at worst a let-bound function) that takes in arguments in one shape and maps to into another form, so it’s probably more common than in the OO world to use the adapter approach.

Now that you have a pipeline, you can test it out in the REPL, executing progressively more of the pipeline, one stage at a time; you should see outputs as per Figure 19.2 for each stage, and end up with an account with a balance of £75.

#### 19.2.2 Moving to user-driven input

Now that we’ve tested our pipeline out in a hard-coded fashion, let’s change to a user-driven input. As you’ll see, our pipeline stays unchanged – the only difference is that we’ll replace the hard-coded set of commands with user-generated input, and change the `getAmount` function to again return user-generated input. First, let’s migrate your code into the program.

#### Now you try

1. Open up the Capstone3 solution and navigate to `Program.fs`.
2. Copy across the helper functions you created in your script above `main`.
3. Copy across the main pipeline you created based on Listing 19.1 in place of the entire “`// Fill in the main loop here...`” block.
4. Run the application. You should see the final account printed with a balance of £75.

Now that we’ve ported the code, let’s start to replace the hard-coded input by using a *lazy* sequence via the `yield` keyword.

### Listing 19.3 – Creating a sequence of user-generated inputs

```
let consoleCommands = seq { ①
 while true do
 Console.Write "(d)eposit, (w)ithdraw or e(x)it: "
 yield Console.ReadKey().KeyChar } ②
```

- ① A sequence block
- ② Yielding out keys sourced from the console

This sequence will execute *forever*; every time the pipeline pulls another item from the sequence of commands, it’ll loop through, print to the console and read the key entered. Importantly, we resist the temptation to filter out invalid commands etc. here – we want this as a simple stream of keys that we can plug in to our existing pipeline.

5. Next, replace the `getAmount` function with a new `getAmountConsole` function. It will also print to the console to ask the user to enter the amount, before reading a line from the console and parsing it as a `Decimal`. As this function has the same signature (`char -> char * decimal`) we can simply replace the existing call to `getAmount` in the pipeline with this one.
6. Run the application. With a little bit of care, the output will like something (although probably not *exactly*) like this: -

```
C:\WINDOWS\system32\cmd.exe
Please enter your name: Sam
Current balance is £0
(d)eposit, (w)ithdraw or e(x)it: d
Enter Amount: 50

Account Sam: deposit of 50 (approved: true)
Current balance is £50

(d)edeposit, (w)ithdraw or e(x)it: d
Enter Amount: 25

Account Sam: deposit of 25 (approved: true)
Current balance is £75
```

**Figure 19.3 – Sample output of our application using a function pipeline over a lazy sequence**

What's nice about this approach is that we were able to treat the *inputs* of data independently of the *processing logic* over that data. We tested some filtering logic and hooked into our existing code base, before simply replacing a couple of small functions in our pipeline to move from some hard-coded data suitable for testing in a script, to a user-driven console application.

### 19.3 Writing Transactions to disk

Let's have a little more fun now and add the ability to persist individual *account transactions* to disk, which will then allow us to re-load an account from disk when we restart the application. Recall that so far, we already support the ability to write to both the console and disk. However, the format that we emit to disk is not really suitable for what we need – it's just the user-friendly console outputs; we need a structure that's easily *machine-readable*.

What we'll do is to amend our logging functions so that instead of taking in a raw string message to log, it'll take in a `Transaction` record, which (unsurprisingly) contains the details of the transaction being attempted.

#### Now you try

1. In `Domain.fs`, create a new record type, `Transaction`. It should contain enough detail with which to store what has occurred, such as the amount, whether a deposit or

withdrawal (just use a `string` or `char` field), a timestamp and perhaps whether or not the attempt was successful.

2. Open up the `FileRepository` module and change the `writeTransaction` function so that it takes in a `transaction` rather than a `message`.
3. Next, create a serialized string that represents the transaction record using basic `sprintf` functionality; use a custom delimiter so that you can easily “split apart” the string again later on. Of course, we might want to use a third-party serialization framework such as JSON.Net, but in the interests of keeping things simple for now, use something like this one: -

#### **Listing 19.4 – Sample serializer for a transaction record**

```
let serialized transaction =
 sprintf "%0***%s***%M***%b"
 transaction.Timestamp
 transaction.Operation
 transaction.Amount
 transaction.Accepted
```

In my suggested solution, I’ve created a module called `Transactions` in `Domain.fs` which contains the serialization function, but feel free to put it wherever you want.

4. Your code now won’t compile, because the console logger (`printfTransaction` in the `Auditing` module) no longer matches the function signature of the file logger. Fix it up so that it takes in a `transaction` instead of a message string, and then prints a console message based on the contents of the transaction record.
5. You’ll also need to fix the `auditAs` function, which carries out some operation (deposit or withdraw), and logs out what happened. You’ll need to change it to create a `transaction` and then pass that to the `audit` function that’s supplied.
6. Once that’s compiling, make sure in `Program.fs` to replace the raw calls to `deposit` and `withdraw` with the ready-made `depositWithAudit` and `withdrawWithAudit` functions.
7. Test the application; you should see that it correctly creates a file in an `accounts` directory for each transaction made with the serialized contents of each transaction in each file.

## **19.4 Rehydrating an account from disk**

OK, great – we’ve now managed to get our tool serializing all the transactions to disk. Now, let’s go the other way. What the tool should now do is on startup, instead of creating a blank account with a balance of £0, is to search for a folder for that user (by name), and then rehydrate the current status of their account based on their transaction history. This is actually pretty simple, again, if you break it down it small, composable functions.

### Now you try

1. Create a function, `loadAccount` in `Operations`. This function should take in an `owner`, `accountId` and a list of `transactions`, and return back an `account`. You'll want to sort the transactions by oldest date first, and then `fold` them together into an `account` (pretty much as you've done already in the main program). Depending on whether the transaction was a withdrawal or a deposit, call the appropriate function in `Operations`.
2. Test this function out (you developed the function in the script, right?) so that given a set of transactions that you create in the script, you end up with an account that has the correct balance. When you're happy with it, port it into the application.
3. Now let's deal with pulling back the transactions from disk, which we can then plug into `loadAccount`. Create a `deserialize` function that, given a single `string`, will recreate a `Transaction` record from it. If you created a `Transactions` module, put it in there.
4. Create a function `findTransactionsOnDisk` in `FileRepository` that, given an `owner`, can retrieve the account id and all the deserialized transactions from the folder. There are a number of private helper functions already written for you in the module to help you along. As there's a possibility that this is a new user, and this function is an *expression*, you'll have to cater for that possibility. As we haven't learned the "proper" way to do that in F# yet, just return a new `Guid` for the account id and an empty sequence of transactions using `Seq.empty`.
5. In `Program.fs`, instead of starting with a hard-coded empty account, after capturing the account owner's name, you'll want to call `FileRepository.findTransactionsOnDisk` before passing the results to `Operations.loadAccount`. Depending on the signatures of the two functions, you might be able to simply compose them together into one function – but this will depend on whether the output of the first matches the input of the second!
6. Test the application out by first creating a new customer and performing several transactions before quitting the application. Then, restarting the application should rehydrate the same account based on the transactions that were already saved.

## 19.5 Summary

And we're done! As always, there's a suggested solution in the repository for you to look at to give you some ideas on what you might have done. We'll come back to this solution later on in the book, so that we can incorporate other F# language features and libraries in as appropriate.

# Unit 5

## The Pit of Success with the F# Type System

We're making progress now! We've covered F# syntax, FP principles, functions and collections – almost there! This next unit will be the last one to really focus on core "language" features in F# - remember that this book covers a *core subset* of the F# language – after which we'll have some more fun with actually writing some meaningful applications that use a variety of frameworks and libraries.

A common phrase you'll hear in F# circles is the ability to "make illegal states unrepresentable". In C# and VB .NET, we're used to the notion of "proving" that an application is correct through, for example, unit tests and console applications. In F#, we still use unit tests, but to a far smaller degree. Part of that, as you know, is because of the REPL. But another side of it is that the F# type system allows us to represent business logic as code so that if your application compiles, it probably *just works*. Sounds crazy, right?

In a more tangible sense, this unit is all about modelling program flow, domains and business logic in F#, covering a set of language features that work together to provide a much more powerful way to reason about what your program does than simple if / then statements and inheritance hierarchies do.

# 20

## *Program Flow in F#*

In C# and VB.NET, we have a variety of ways of performing what I consider to be “program flow” i.e. branching mechanisms and, to an extent, loops. In this lesson, we’ll compare and contrast those features with equivalents in F#, looking at: -

- For and While Loops
- If / Then expressions
- Switch / Case
- Pattern Matching
- When we’re done, you’ll have a good idea in how to perform all sorts of complex conditional logic much more succinctly than you might be used to.

### 20.1 A tour around loops in F#

I’m going to cover loops very briefly in this lesson as the F# side of things is essentially just a slightly different syntax compared to C# / VB .NET with similar behaviour – this leaves us more room to focus on branching logic in F#, which is much more interesting. You’ve already seen examples of these constructs in this book already, so you can consider this reference material more than anything.

The main thing to know is that, Comprehensions aside, these looping constructs, while officially expressions, are inherently imperative, designed to work with side-effectful code – code that doesn’t have any tangible output e.g. printing to the console, or saving to the database. In that respect, these loops should be very familiar to you (and therefore not used as often in F# as C# / VB .NET).

### 20.1.1 For loops

Foreach and for loops can be modelled using the `for .. in` construct in F#. `for in` loops also have a handy syntax for creating *ranges* of data quickly, so although there's a separate construct in F# for "simple" for loops (known as `for .. to`), I've ignored it here.

#### **Listing 20.1 for .. in loops in F#**

```
for number in 1 .. 10 do ①
 printfn "%d Hello!" number

for number in 10 .. -1 .. 1 do ②
 printfn "%d Hello!" number

let customerIds = [45 .. 99]
for customerId in customerIds do ③
 printfn "%d bought something!" customerId

for even in 2 .. 2 .. 10 do ④
 printfn "%d is an even number!" even
```

- ① Upwards-counting for loop
- ② Downwards-counting for loop
- ③ Typical for each-style loop
- ④ Range with custom stepping

### 20.1.2 While loops

while loops in F# behave just like those that you're used to.

#### **Listing 20.2 while loops in F#**

```
open System.IO
let reader = new StreamReader(File.OpenRead @"File.txt") ①
while (not reader.EndOfStream) do ②
 printfn "%s" (reader.ReadLine())
```

- ① Opening a handle to a text file
- ② while loop which runs while the reader is not at the end of the stream

#### **Breaking the loop**

The main restriction of loops in F# is that there is no concept of the `break` command, so you can't exit out of a loop prematurely – sorry. If you want to simulate premature exit of a loop, you should consider replacing the loop with a sequence of values that you `filter` on (or `takeWhile`), and loop over *that* sequence instead. In fact, in case you've not noticed yet, no code we've returned in a function uses "early return" – that's because it's not supported in F#. Again, because everything is an expression, each branch must have an equivalent result.

### 20.1.3 Comprehensions

Comprehensions are a powerful way of generating lists, arrays and sequences of data based on for loop-style syntax. The closest equivalent in C# would be the use of the `System.Linq.Enumerable.Range()` method, except rather than *library* support, F# has native *language* support for this. Here's how they work: -

#### Listing 20.3 Comprehensions in F#

```
open System

let arrayOfChars = [| for c in 'a' .. 'z' -> Char.ToUpper c |] ①
let listOfSquares = [for i in 1 .. 10 -> i * i] ②
let seqOfStrings = seq { for i in 2 .. 4 .. 20 -> sprintf "Number %d" i } ③
```

- ① Generating an array of the letters of the alphabet in uppercase.
- ② Generating the squares of the numbers one to ten.
- ③ Generating some arbitrary strings based on every fourth number between 2 and 20.

You'll find comprehensions very useful for quickly generating collections of data based on some set of numbers – for example, calling a SQL stored procedure to load all customers between two date ranges etc.

#### Quick Check

1. What restriction does F# place on you with returning out of loops?
2. What is the syntax to perform “for each” loops in F#?
3. Can you use while loops in F#?

## 20.2 Branching logic in F#

We're used to using one of two branching mechanisms in C# - **if / then** and **switch / case**. The former is used for most general purpose branching logic; in my experience, the latter tends to be used more rarely. That's unfortunate – switch / case is a more constrained model than if / then which operates against a *single value*. This means that it's often a little bit easier to reason about branching decisions than if / then, but overall it's fairly limited, working only against objects from a few types (integers, strings and enums). If / then is more powerful, but is completely unconstrained, which means it's relatively easy to write code that is hard to reason about, or branches that accidentally operate over different data.

F# has an entirely different construct for handling branching logic called Pattern Matching – let's work through an example to illustrate how we can improve upon if / else expressions.

### 20.2.1 Priming Exercise – Customer Credit Limits

Let's say we wanted to write some code which calculated a customer's credit limit, based on the customer's third-party credit score, and the number of years that they've been a customer

of ours. For example, if the customer has a credit score of "medium" and had been with us for one year, we would give them a credit limit of \$500.

It sometimes helps me to understand "compound" conditions such as this if I visualize it as a kind of truth table. Let's do that now for the different cases that we want to model: -

**Table 20.1 Modelling logic as a truth table**

| Credit Score    | Years as a Customer | Credit Limit |
|-----------------|---------------------|--------------|
| medium          | 1                   | 500          |
| good            | 0                   | 750          |
| good            | 1                   | 750          |
| good            | 2                   | 1000         |
| good            | <anything else>     | 2000         |
| <anything else> | <anything else>     | 250          |

Even with just two "features" and a couple of "or" conditions, modelling this with if / then expressions can be a little awkward: -

#### **Listing 20.4 if / then expressions for complex logic**

```
let limit =
 if score = "medium" && years = 1 then 500 ①
 elif score = "good" && (years = 0 || years = 1) then 750 ②
 elif score = "good" && years = 2 then 1000
 elif score = "good" then 2000 ③
 else 250 ④
```

- ① A simple clause
- ② Complex clause – AND and OR combined.
- ③ "Catch all" for "good" customers
- ④ "Catch all" for other customers

You might think that this is pretty standard code - but the truth is it's somewhat difficult to reason about the relationship of all the clauses as one unified piece of business logic – each clause is *completely unrelated*. There's nothing to stop you accidentally comparing against something else instead of `score` in just one branch, for example. Also, notice how the "catch all" parts of the code are somewhat "implicit" - this can lead to all sorts of weird and wonderful bugs, particularly when you have more than just a couple of clauses.

#### **20.2.2 Say hello to pattern matching**

F#'s solution to modelling branching logic is an entirely different construct called *pattern matching*. Pattern matching is an expression-based branching mechanism that also allows *inline binding* for a wide variety of F# constructs – in other words, the ability to deconstruct a

tuple or record *whilst* pattern matching. Perhaps the most apt way that I've heard it described before is as "switch / case on steroids" – essentially the principals are somewhat similar, but pattern matching takes things a whole lot further. Let's take a look!

#### **Listing 20.5 Our first pattern matching example**

```
let limit =
 match customer with
 | "medium", 1 -> 500 1
 | "good", 0 | "good", 1 -> 750 2
 | "good", 2 -> 1000 3
 | "good", _ -> 2000 4
 | _ -> 250 5
```

- 1 Implicitly matching on a tuple of rating and years
- 2 If medium score with 1-year history, limit is \$500
- 3 Two match conditions leading to \$750 limit
- 4 "Catch all" for other customers with "good" score
- 5 "Catch all" for all other customers

The syntax is a little different to what you're probably used to (but then again, that's par for the course with F#!) – but it's also pretty powerful. Some things to note: -

- We always test against a single source object, much like switch / case. Unlike if / then, it's impossible for us to compare against different values across branches.
- We match against *patterns* that represents specific cases. In our example these patterns are tuples of credit score and years.
- The compiler can automatically infer the type of `customer as (string * int)` based on the usage within the different patterns.
- We're matching here against some *constant values*, just like switch / case. However, we can also *deconstruct* a tuple and match against the individual components of it (as we'll see shortly).
- We can model multiple patterns to a single, shared output.
- We can model "catch all" style scenarios as well with the wildcard (`_`) pattern - even on just a *subset* of the overall pattern.
- Pattern matching is, of course, an expression – so the match returns a value (in our case, the credit limit for the customer). Just like if / then expression, all the different branches *must* return the same type – in our case, that's an integer (the credit limit).

#### **20.2.3 Exhaustive checking**

One thing that switch / case gives us over if / then in C# is that the compiler can give us a little bit more support e.g. if we switch on the same case twice etc. Pattern matching has even more better compiler support - it gives us *exhaustive* checking. Because pattern matches are expressions, a pattern match must *always* return a result; if an input value can't be matched by the code at runtime, F# will actually throw an exception! So, to help us out F# will actually

warn us if we don't cater for *all* potential possibilities, as well as telling us about rules that can never be matched.

### Now you try

Let's work through a simple example that illustrates how exhausting pattern matching works.

1. Open a new script file.
2. Create a function `getCreditLimit` that takes in a "customer" value. Don't specify the type of the customer – let the compiler infer it for us.
3. Copy across the pattern match code from this above sample that calculates the limit and return back the limit from the function. Ensure this compiles and you can call it with a sample tuple e.g. ("medium", 1)
4. Remove the final (catch all) pattern (`| _ -> 250`).
5. Check the warning highlighted at the top of the match clause.

```
let getCreditLimit customer =
 match customer with
 | "medium" | "good", _
```

Incomplete pattern matches on this expression. For example,  
the value '(\_0)' may indicate a case not covered by the pattern(s).

Figure 20.1 – Exhaustive pattern matching

In other words, we haven't catered for the case where the customer has some arbitrary credit score and 0 years' history.

1. Call the function with a value that won't be matched e.g. ("bad", 0) and see in FSI that a `MatchFailureException` is raised.
2. Fill in a new pattern at the bottom of `(_, 0)` and set the output for this case as 250.
3. Notice that the warning has now changed to say we need to fill in a case for `(_, 2)` etc. etc.

Exhaustive pattern matching is useful here as a reminder to add a "catch all", but really comes into its own when working with *discriminated unions*, which we'll see later in this unit. Let's now demonstrate how F# also warns us about *unreachable* patterns.

1. Change the new pattern that you just created to match on `(_, 1)`.
2. Move that clause to be the first pattern.
3. Observe that a warning is now shown against the `(medium, 1)` case.

```

match customer with
| _, 1 -> 250
| "medium", 1 -> 500
| "good" This rule will never be matched
| _ match

```

The code shows a pattern match for 'customer'. It includes three cases: one for a tuple where the first element is \_, and the second is 1, resulting in 250; another for a tuple where the first element is "medium" and the second is 1, resulting in 500; and a third case for "good". A tooltip 'This rule will never be matched' is shown over the fourth case, which is a wildcard pattern '\_'. Below this, there is a partially visible 'match' keyword.

Figure 20.2 – F# warns us about patterns that can never be matched.

It's important to understand that pattern matching works *top down* – so you should always put the most *specific* patterns first, and the most *general* ones last.

#### 20.2.4 Guards

F# also gives us a nice escape hatch for pattern matching so that we can do *any* form of check within a pattern rather than just matching against values – this is known as the `when` guard clause. For example, we could merge two of our above patterns into one with the following: -

##### **Listing 20.6 Using the when guard clause**

```

let getCreditLimit customer =
 match customer with
 | "medium", 1 -> 500
 | "good", years when years < 2 -> 750 ①
 | ... // etc.

```

① Using the when guard to specify a custom pattern

Notice that we've also bound the number of years that the customer has been with us to the `years` symbol – just like when decomposing a tuple normally, you can choose any symbol that you want. You can also use bound symbols on the *right-hand-side* of the arrow – useful if the action for that pattern needs to use input values.

##### **When not to when?**

You obviously have a lot of control using the `when` clause in pattern matching, but there's a cost associated with this: The compiler won't try to figure out anything that happens *inside* the guard - so as soon as you use one the compiler won't be able to perform exhaustive pattern matching for you (although it will still exhaust all possibilities that it can prove).

#### 20.2.5 Nested matches

Just like `if / then` (or `switch / case`), you can *nest* pattern matches. I'd only recommend doing so when you have an extreme number of repeated elements – otherwise the benefit of

removing the repeated element is offset by the cost of the extra code complexity. Here's how the same example would look with nested matches: -

#### **Listing 20.7 Nesting matches inside one another**

```
let getCreditLimit customer =
 match customer with
 | "medium", 1 -> 500
 | "good", years -> ①
 match years with ②
 | 0 | 1 -> 750 ③
 | 2 -> 1000
 | _ -> 2000
 | _ -> 250 ④
```

- ① Matching on "good" and binding years to a symbol
- ② A nested match on the value of years
- ③ Single value match
- ④ "Global" catch all

#### **Quick Check**

4. What are the limitations of switch / case?
5. Why can unconstrained clauses such as if / then expressions lead to bugs?
6. What sort of support does pattern matching give us to ensure correctness?

### **20.3 Flexible pattern matching**

So far we've seen above a couple of types of pattern matching – constant matching (as per Listing 20.6) and *tuple* pattern matching i.e. the ability to deconstruct and match against a tuple of values using standard F# tuple syntax. However, pattern matching can do much more than that – we can also match many against other "types" of data. Here are a few common types of matches – you can find a complete list on the F# documentation site which is at this url: <https://msdn.microsoft.com/en-gb/visualfsharpdocs/conceptual/fsharp-language-reference>.

#### **20.3.1 Collections**

F# lets you *safely* pattern match against a list or array. This means that instead of having code that first checks the length of a list before indexing into it, you can get the compiler to safely extract values out of the list in one operation. For example, let's say you have code that should operate on a list of customers.

- If **no** customers (i.e. the list is empty) are supplied, you throw an error
- If there's **one** customer, print out their name.
- If there are **two** customers, you'd like to print the sum of their balances.
- **Otherwise**, print out the total number of customers supplied.

Pretty arbitrary logic, right?

## Now you try

Let's work through the above logic to see pattern matching over lists in action.

1. Create a `Customer` record type which has fields `Balance : int` and `Name : string`.
2. Create a function called `handleCustomers` that takes in a list of `Customer` records.
3. Implement the above logic using standard if / then logic. You can use `List.length` to calculate the length of customers, or explicitly type annotate the `Customer` argument as `customer list` and get the `Length` property on the list.
4. Use `failwith` to raise an exception e.g. `failwith "No customers supplied!"`.
5. Now enter the following pattern match version for comparison: -

### **Listing 20.8 Matching against lists**

```
let handleCustomer customers =
 match customers with
 | [] -> failwith "No customers supplied!" ①
 | [customer] -> printfn "Single customer, name is %s" customer.Name ②
 | [first; second] -> printfn "Two customers, balance = %d" (first.Balance +
 second.Balance) ③
 | customers -> printfn "Customers supplied: %d" customers.Length ④

handleCustomer [] // throws exception
handleCustomer [{ Balance = 10; Name = "Joe" }] // prints name
```

- ① Matching against an empty list
- ② Matching against a list of one customer
- ③ Matching against a list of two customers
- ④ Matching against all other lists

One big difference with pattern matching versus manually checking the length of lists first is that here it's *impossible* to accidentally try to access a value in a list that doesn't exist, as the compiler is doing both the length check and "expanding" the values of the list for us. In other words, we're replacing runtime logic for compile-time safety.

Of course, an example like this is only useful for small lists – you wouldn't do this for lists of hundreds of items – but you'd be surprised how often you check against lists that have just a few items in them.

### **20.3.2 Records**

We can also pattern match on *records*. What does this mean? Well, here's an example of pattern matching against our fictional `Customer` type to return a description of them.

### **Listing 20.9 Pattern Matching with Records**

```
let getStatus customer =
 match customer with
 | { Balance = 0 } -> "Customer has empty balance!" ①
 | { Name = "Isaac" } -> "This is a great customer!" ②
 | { Name = name; Balance = 50 } -> sprintf "%s has a large balance!" name
```

```
| { Name = name } -> sprintf "%s is a normal customer" name ③
{ Balance = 50; Name = "Joe" } |> getStatus
```

- ① Match against Balance field
- ② Match against Name field
- ③ Catch all, binding Name to name symbol

Notice that we don't have to fill in all the fields – just the ones that we want to match against. However, if we want to, we can bind specific fields to symbols so that we can use them on the right-hand-side – pretty neat!

You can even mix and match patterns – how about checking that a list of customers has three elements, the first customer is called "Tanya" and the second customer has a balance of 25? No problem!

#### **Listing 20.10 Combing multiple patterns together**

```
match customers with
| [{ Name = "Tanya" }; { Balance = 25 }; _] -> "It's a match!" ①
| _ -> "No match!"
```

- ① Pattern matching against a list of three items with specific fields

#### **Quick Check**

7. What collection types can you *not* pattern match against?

## **20.4 To match or not to match**

With two branching mechanisms at your disposal, which should you use – if/then or pattern matching? My advice is: Use pattern matching by default – it's more powerful, easier to reason about and much more flexible. The *only* time it's really simpler using if / then is when you're working with side-effectful code that returns unit, and you're "implicitly" missing the default branch: -

#### **Listing 20.11 When to use if / then over match**

```
if customer.Name = "Isaac" then printfn "Hello!" ①
match customer.Name with ②
| "Isaac" -> printfn "Hello!"
| _ -> ()
```

- ① If / then with implicit "default" else branch
- ② Match with explicit "default" case

The F# compiler is smart enough with if / then to automatically put in a default handler for us for the "else" branch, but the match construct always expects an explicit default handler.

## 20.5 Summary

That's the end of the Program Flow unit. Let's recap what we looked at: -

- We briefly reviewed for and while loops
- You saw how to use *comprehensions* to easily generate collections
- We compared if / then expressions with switch / case statements.
- But we spent most of the lesson looking at pattern matching, a powerful branching mechanism in F#.

We've really only scratched the surface of what's possible with Pattern Matching. Its real benefit is that it's incredibly expressive, powerful and yet simple to use - once you know what can be done with it. Don't be surprised if you found it a little unusual – the idea of pattern matching is unlike both branching mechanisms you already know. We'll be using it more and more in the coming lessons in this unit as it's pervasive within F#, so you'll have more opportunities to get your hands dirty with it.

### **Try This**

Experiment with pattern matching over lists, tuples and records. Start by creating a random list of numbers of variable length and writing pattern matches to test if the list: -

- Is a specific length
- Is empty
- Has the first item equal to 5 (*hint: use head / tail syntax here with ::*)

Then experiment with pattern matching over a record. Continue with the file system Try This exercise from the previous lessons; pattern match over data to check whether a folder is large or not, based on average file size or count of files.

### **Quick Check Answers**

1. You cannot prematurely exit from a for loop in F#
2. `for <binding> in <collection> do <code>`
3. Yes; while loops work in F# pretty much as in C#.
4. Limited set of types it can work over; no binding support.
5. It's easy to write branches that are difficult to reason about and inconsistent.
6. Exhaustive pattern matching; binding and construction of tuples and collections.
7. Sequences cannot be pattern matched against; only arrays and lists are supported.

# 20

## *Program Flow in F#*

In C# and VB.NET, we have a variety of ways of performing what I consider to be “program flow” i.e. branching mechanisms and, to an extent, loops. In this lesson, we’ll compare and contrast those features with equivalents in F#, looking at:-

- For and While Loops
- If / Then expressions
- Switch / Case
- Pattern Matching
- When we’re done, you’ll have a good idea in how to perform all sorts of complex conditional logic much more succinctly than you might be used to.

### 20.1 A tour around loops in F#

I’m going to cover loops very briefly in this lesson as the F# side of things is essentially just a slightly different syntax compared to C# / VB .NET with similar behaviour – this leaves us more room to focus on branching logic in F#, which is much more interesting. You’ve already seen examples of these constructs in this book already, so you can consider this reference material more than anything.

The main thing to know is that, Comprehensions aside, these looping constructs, while officially expressions, are inherently imperative, designed to work with side-effectful code – code that doesn’t have any tangible output e.g. printing to the console, or saving to the database. In that respect, these loops should be very familiar to you (and therefore not used as often in F# as C# / VB .NET).

### 20.1.1 For loops

Foreach and for loops can be modelled using the `for .. in` construct in F#. `for in` loops also have a handy syntax for creating *ranges* of data quickly, so although there's a separate construct in F# for "simple" for loops (known as `for .. to`), I've ignored it here.

#### **Listing 20.1 for .. in loops in F#**

```
for number in 1 .. 10 do #A
 printfn "%d Hello!" number

for number in 10 .. -1 .. 1 do #B
 printfn "%d Hello!" number

let customerIds = [45 .. 99]
for customerId in customerIds do #C
 printfn "%d bought something!" customerId

for even in 2 .. 2 .. 10 do #D
 printfn "%d is an even number!" even

#A Upwards-counting for loop
#B Downwards-counting for loop
#C Typical for each-style loop
#D Range with custom stepping
```

### 20.1.2 While loops

while loops in F# behave just like those that you're used to.

#### **Listing 20.2 while loops in F#**

```
open System.IO
let reader = new StreamReader(File.OpenRead @"File.txt") #A
while (not reader.EndOfStream) do #B
 printfn "%s" (reader.ReadLine())
```

#A Opening a handle to a text file

#B while loop which runs while the reader is not at the end of the stream

#### **Breaking the loop**

The main restriction of loops in F# is that there is no concept of the *break* command, so you can't exit out of a loop prematurely – sorry. If you want to simulate premature exit of a loop, you should consider replacing the loop with a sequence of values that you filter on (or `takeWhile`), and loop over *that* sequence instead. In fact, in case you've not noticed yet, no code we've returned in a function uses "early return" – that's because it's not supported in F#. Again, because everything is an expression, each branch must have an equivalent result.

### 20.1.3 Comprehensions

Comprehensions are a powerful way of generating lists, arrays and sequences of data based on for loop-style syntax. The closest equivalent in C# would be the use of the `System.Linq.Enumerable.Range()` method, except rather than *library* support, F# has native *language* support for this. Here's how they work: -

#### **Listing 20.3 Comprehensions in F#**

```
open System

let arrayOfChars = [| for c in 'a' .. 'z' -> Char.ToUpper c |] #A
let listOfSquares = [for i in 1 .. 10 -> i * i] #B
let seqOfStrings = seq { for i in 2 .. 4 .. 20 -> sprintf "Number %d" i } #D

#A Generating an array of the letters of the alphabet in uppercase.
#B Generating the squares of the numbers one to ten.
#C Generating some arbitrary strings based on every fourth number between 2 and 20.
```

You'll find comprehensions very useful for quickly generating collections of data based on some set of numbers – for example, calling a SQL stored procedure to load all customers between two date ranges etc.

#### **Quick Check**

1. What restriction does F# place on you with returning out of loops?
2. What is the syntax to perform “for each” loops in F#?
3. Can you use while loops in F#?

## 20.2 Branching logic in F#

We're used to using one of two branching mechanisms in C# - if / then and switch / case. The former is used for most general purpose branching logic; in my experience, the latter tends to be used more rarely. That's unfortunate – switch / case is a more constrained model than if / then which operates against a *single value*. This means that it's often a little bit easier to reason about branching decisions than if / then, but overall it's fairly limited, working only against objects from a few types (integers, strings and enums). If / then is more powerful, but is completely unconstrained, which means it's relatively easy to write code that is hard to reason about, or branches that accidentally operate over different data.

F# has an entirely different construct for handling branching logic called Pattern Matching – let's work through an example to illustrate how we can improve upon if / else expressions.

### 20.2.1 Priming Exercise – Customer Credit Limits

Let's say we wanted to write some code which calculated a customer's credit limit, based on the customer's third-party credit score, and the number of years that they've been a customer

of ours. For example, if the customer has a credit score of "medium" and had been with us for one year, we would give them a credit limit of \$500.

It sometimes helps me to understand "compound" conditions such as this if I visualize it as a kind of truth table. Let's do that now for the different cases that we want to model: -

**Table 20.1 Modelling logic as a truth table**

| Credit Score    | Years as a Customer | Credit Limit |
|-----------------|---------------------|--------------|
| medium          | 1                   | 500          |
| good            | 0                   | 750          |
| good            | 1                   | 750          |
| good            | 2                   | 1000         |
| good            | <anything else>     | 2000         |
| <anything else> | <anything else>     | 250          |

Even with just two "features" and a couple of "or" conditions, modelling this with if / then expressions can be a little awkward: -

#### **Listing 20.4 if / then expressions for complex logic**

```
let limit =
 if score = "medium" && years = 1 then 500 #A
 elif score = "good" && (years = 0 || years = 1) then 750 #B
 elif score = "good" && years = 2 then 1000
 elif score = "good" then 2000 #C
 else 250 #D
```

#A A simple clause

#B Complex clause – AND and OR combined.

#C "Catch all" for "good" customers

#D "Catch all" for other customers

You might think that this is pretty standard code - but the truth is it's somewhat difficult to reason about the relationship of all the clauses as one unified piece of business logic – each clause is *completely unrelated*. There's nothing to stop you accidentally comparing against something else instead of `score` in just one branch, for example. Also, notice how the "catch all" parts of the code are somewhat "implicit" - this can lead to all sorts of weird and wonderful bugs, particularly when you have more than just a couple of clauses.

### **20.2.2 Say hello to pattern matching**

F#'s solution to modelling branching logic is an entirely different construct called *pattern matching*. Pattern matching is an expression-based branching mechanism that also allows *inline binding* for a wide variety of F# constructs. Perhaps the most apt way that I've heard it

described before is as “switch / case on steroids” – essentially the principals are somewhat similar, but pattern matching takes things a whole lot further. Let’s take a look!

#### **Listing 20.5 Our first pattern matching example**

```
let limit =
 match customer with #A
 | "medium", 1 -> 500 #B
 | "good", 0 | "good", 1 -> 750 #C
 | "good", 2 -> 1000
 | "good", _ -> 2000 #D
 | _ -> 250 #E
```

#A Implicitly matching on a tuple of rating and years  
#B If medium score with 1-year history, limit is \$500  
#C Two match conditions leading to \$750 limit  
#D “Catch all” for other customers with “good” score  
#E “Catch all” for all other customers

The syntax is a little different to what you’re probably used to (but then again, that’s par for the course with F#!) – but it’s also pretty powerful. Some things to note: -

- We always test against a single source object, much like switch / case. Unlike if / then, it’s impossible for us to compare against different values across branches.
- We match against *patterns* that represents specific cases. In our example these patterns are tuples of credit score and years.
- The compiler can automatically infer the type of `customer as (string * int)` based on the usage within the different patterns.
- We’re matching here against some *constant values*, just like switch / case. However, we can also *deconstruct* a tuple and match against the individual components of it.
- We can model multiple patterns to a single, shared output.
- We can model “catch all” style scenarios as well with the wildcard `(_)` pattern - even on just a *subset* of the overall pattern.
- Pattern matching is, of course, an expression – so the match returns a value (in our case, the credit limit for the customer). Just like if / then expression, all the different branches *must* return the same type – in our case, that’s an integer (the credit limit).

#### **20.2.3 Exhaustive checking**

One thing that switch / case gives us over if / then in C# is that the compiler can give us a little bit more support e.g. if we switch on the same case twice etc. Pattern matching has even more better compiler support - it gives us *exhaustive* checking. Because pattern matches are expressions, a pattern match must *always* return a result; if an input value can’t be matched by the code at runtime, F# will actually throw an exception! So, to help us out F# will actually *warn us* if we don’t cater for *all* potential possibilities, as well as telling us about rules that can *never* be matched.

## Now you try

Let's work through a simple example that illustrates how exhausting pattern matching works.

1. Open a new script file.
2. Create a function `getCreditLimit` that takes in a "customer" value. Don't specify the type of the customer – let the compiler infer it for us.
3. Copy across the pattern match code from this above sample that calculates the limit and return back the limit from the function. Ensure this compiles and you can call it with a sample tuple e.g. ("medium", 1)
4. Remove the final (catch all) pattern (`| _ -> 250`).
5. Check the warning highlighted at the top of the match clause.

```
let getCreditLimit customer =
 match customer with
 | "medium" Incomplete pattern matches on this expression. For example,
 | "good", the value '(_,_)' may indicate a case not covered by the pattern(s).
 | _
```

Figure 20.1 – Exhaustive pattern matching

In other words, we haven't catered for the case where the customer has some arbitrary credit score and 0 years' history.

1. Call the function with a value that won't be matched e.g. ("bad", 0) and see in FSI that a `MatchFailureException` is raised.
2. Fill in a new pattern at the bottom of `(_, 0)` and set the output for this case as 250.
3. Notice that the warning has now changed to say we need to fill in a case for `(_, 2)` etc. etc.

Exhaustive pattern matching is useful here as a reminder to add a "catch all", but really comes into its own when working with *discriminated unions*, which we'll see later in this unit. Let's now demonstrate how F# also warns us about *unreachable patterns*.

1. Change the new pattern that you just created to match on `(_, 1)`.
2. Move that clause to be the first pattern.
3. Observe that a warning is now shown against the `(medium, 1)` case.

```
match customer with
 | _, 1 -> 250
 | "medium", 1 -> 500
 | "good" This rule will never be matched
 | _
```

Figure 20.2 – F# warns us about patterns that can never be matched.

It's important to understand that pattern matching works *top down* – so you should always put the most *specific* patterns first, and the most *general* ones last.

#### 20.2.4 Guards

F# also gives us a nice escape hatch for pattern matching so that we can do *any* form of check within a pattern rather than just matching against values – this is known as the `when` guard clause. For example, we could merge two of our above patterns into one with the following: –

##### **Listing 20.6 Using the when guard clause**

```
let getCreditLimit customer =
 match customer with
 | "medium", 1 -> 500
 | "good", years when years < 2 -> 750 #A
 | ... // etc.
```

#A Using the when guard to specify a custom pattern

Notice that we've also bound the number of years that the customer has been with us to the `years` symbol – just like when decomposing a tuple normally, you can choose any symbol that you want. You can also use bound symbols on the *right-hand-side* of the arrow – useful if the action for that pattern needs to use input values.

---

##### **When not to when?**

You obviously have a lot of control using the `when` clause in pattern matching, but there's a cost associated with this: The compiler won't try to figure out anything that happens *inside* the guard – so as soon as you use one the compiler won't be able to perform exhaustive pattern matching for you (although it will still exhaust all possibilities that it *can* prove).

---

#### 20.2.5 Nested matches

Just like `if / then` (or `switch / case`), you can *nest* pattern matches. I'd only recommend doing so when you have an extreme number of repeated elements – otherwise the benefit of removing the repeated element is offset by the cost of the extra code complexity. Here's how the same example would look with nested matches: –

##### **Listing 20.7 Nesting matches inside one another**

```
let getCreditLimit customer =
 match customer with
 | "medium", 1 -> 500
 | "good", years -> #A
 match years with #B
 | 0 | 1 -> 750 #C
 | 2 -> 1000
 | _ -> 2000
 | _ -> 250 #D
```

```
#A Matching on "good" and binding years to a symbol
#B A nested match on the value of years
#C Single value match
#D "Global" catch all
```

### Quick Check

4. What are the limitations of switch / case?
5. Why can unconstrained clauses such as if / then expressions lead to bugs?
6. What sort of support does pattern matching give us to ensure correctness?

## 20.3 Flexible pattern matching

So far we've seen above a couple of types of pattern matching – constant matching (as per Listing 20.6) and *tuple* pattern matching i.e. the ability to deconstruct and match against a tuple of values using standard F# tuple syntax. However, pattern matching can do much more than that – we can also match many against other “types” of data. Here are a few common types of matches – you can find a complete list on the F# documentation site which is at this url: <https://msdn.microsoft.com/en-gb/visualfsharpdocs/conceptual/fsharp-language-reference>.

### 20.3.1 Collections

F# lets you *safely* pattern match against a list or array. This means that instead of having code that first checks the length of a list before indexing into it, you can get the compiler to safely extract values out of the list in one operation. For example, let's say you have code that should operate on a list of customers.

- If **no** customers (i.e. the list is empty) are supplied, you throw an error
- If there's **one** customer, print out their name.
- If there are **two** customers, you'd like to print the sum of their balances.
- **Otherwise**, print out the total number of customers supplied.

Pretty arbitrary logic, right?

### Now you try

Let's work through the above logic to see pattern matching over lists in action.

1. Create a `Customer` record type which has fields `Balance : int` and `Name : string`.
2. Create a function called `handleCustomers` that takes in a list of `Customer` records.
3. Implement the above logic using standard if / then logic. You can use `List.length` to calculate the length of customers, or explicitly type annotate the `Customer` argument as `customer list` and get the `Length` property on the list.
4. Use `failwith` to raise an exception e.g. `failwith "No customers supplied!"`.
5. Now enter the following pattern match version for comparison: -

**Listing 20.8 Matching against lists**

```

let handleCustomer customers =
 match customers with
 | [] -> failwith "No customers supplied!" #A
 | [customer] -> printfn "Single customer, name is %s" customer.Name #B
 | [first; second] -> printfn "Two customers, balance = %d" (first.Balance +
 second.Balance) #C
 | customers -> printfn "Customers supplied: %d" customers.Length #D

handleCustomer [] // throws exception
handleCustomer [{ Balance = 10; Name = "Joe" }] // prints name

#A Matching against an empty list
#B Matching against a list of one customer
#C Matching against a list of two customers
#D Matching against all other lists

```

One big difference with pattern matching versus manually checking the length of lists first is that here it's *impossible* to accidentally try to access a value in a list that doesn't exist, as the compiler is doing both the length check and "expanding" the values of the list for us. In other words, we're replacing runtime logic for compile-time safety.

Of course, an example like this is only useful for small lists – you wouldn't do this for lists of hundreds of items – but you'd be surprised how often you check against lists that have just a few items in them.

**20.3.2 Records**

We can also pattern match on *records*. What does this mean? Well, here's an example of pattern matching against our fictional Customer type to return a description of them.

**Listing 20.9 Pattern Matching with Records**

```

let getStatus customer =
 match customer with
 | { Balance = 0 } -> "Customer has empty balance!" #A
 | { Name = "Isaac" } -> "This is a great customer!" #B
 | { Name = name; Balance = 50 } -> sprintf "%s has a large balance!" name #C
 | { Name = name } -> sprintf "%s is a normal customer" name #C

{ Balance = 50; Name = "Joe" } |> getStatus

#A Match against Balance field
#B Match against Name field
#C Catch all, binding Name to name symbol

```

Notice that we don't have to fill in all the fields – just the ones that we want to match against. However, if we want to, we can bind specific fields to symbols so that we can use them on the right-hand-side – pretty neat!

You can even mix and match patterns – how about checking that a list of customers has three elements, the first customer is called “Tanya” and the second customer has a balance of 25? No problem!

#### **Listing 20.10 Combing multiple patterns together**

```
match customers with
| [{ Name = "Tanya" }; { Balance = 25 }; _] -> "It's a match!" #A
| _ -> "No match!"
```

#A Pattern matching against a list of three items with specific fields

#### **Quick Check**

7. What collection types can you *not* pattern match against?

## **20.4 To match or not to match**

With two branching mechanisms at your disposal, which should you use – if/then or pattern matching? My advice is: Use pattern matching by default – it’s more powerful, easier to reason about and much more flexible. The *only* time it’s really simpler using if / then is when you’re working with side-effectful code that returns unit, and you’re “implicitly” missing the default branch: -

#### **Listing 20.11 When to use if / then over match**

```
if customer.Name = "Isaac" then printfn "Hello!" #A

match customer.Name with #B
| "Isaac" -> printfn "Hello!"
| _ -> ()
```

#A If / then with implicit “default” branch

#B Match with explicit “default” case

The F# compiler is smart enough with if / then to automatically put in the default unit handler for us, but the match construct always expects an explicit default handler.

## **20.5 Summary**

That’s the end of the Program Flow unit. Let’s recap what we looked at: -

- We briefly reviewed for and while loops
- You saw how to use *comprehensions* to easily generate collections
- We compared if / then expressions with switch / case statements.
- But we spent most of the lesson looking at pattern matching, a powerful branching mechanism in F#.

We've really only scratched the surface of what's possible with Pattern Matching. Its real benefit is that it's incredibly expressive, powerful and yet simple to use - once you know what can be done with it. Don't be surprised if you found it a little unusual – the idea of pattern matching is unlike both branching mechanisms you already know. We'll be using it more and more in the coming lessons in this unit as it's pervasive within F#, so you'll have more opportunities to get your hands dirty with it.

### **Try This**

Experiment with pattern matching over lists, tuples and records. Start by creating a random list of numbers of variable length and writing pattern matches to test if the list: -

- Is a specific length
- Is empty
- Has the first item equal to 5 (*hint: use head / tail syntax here with ::*)

Then experiment with pattern matching over a record. Continue with the file system Try This exercise from the previous lessons; pattern match over data to check whether a folder is large or not, based on average file size or count of files.

### **Quick Check Answers**

1. You cannot prematurely exit from a for loop in F#
2. `for <binding> in <collection> do <code>`
3. Yes; while loops work in F# pretty much as in C#.
4. Limited set of types it can work over; no binding support.
5. It's easy to write branches that are difficult to reason about and inconsistent.
6. Exhaustive pattern matching; binding and construction of tuples and collections.
7. Sequences cannot be pattern matched against; only arrays and lists are supported.

# 21

## *Modelling Relationships in F#*

Whilst we've looked at different ways of storing data in F# - tuples, records etc. – one thing we've not looked much at is how to model *relationships* of data together. So, in this lesson, we'll look at a way of doing just this in F#, using a feature called **Discriminated Unions** – a flexible and powerful modelling tool in F#. We'll: -

- Briefly review inheritance in the OO world
- Learn what Discriminated Unions are, and how to use them
- Compare and contrast inheritance and discriminated unions

We're probably all familiar with modelling relationships in C# already, through one of two mechanisms: Composition and Inheritance. The former establishes a "has-a" relationship, whereas the latter typically models the "is-a" relationship e.g.

- A computer "has a" set of disk drives
- A printer "is a" device

### 21.1.1 Composition in F#

We've already covered this in Lesson 9, where we created two Record types with one referencing the other, but let's quickly recap it with another example: -

#### **Listing 21.1 Composition with Records in F#**

```
type Disk = { SizeGb : int }
type Computer =
 { Manufacturer : string
 Disks: Disk list } #A

let myPc =
 { Manufacturer = "Computers Inc."
 Disks = #B
 [{ SizeGb = 100 }
 { SizeGb = 250 }] }
```

```
{ SizeGb = 500 }] }
```

#A Defining two Record types – Computer is dependent on Disk

#B Creating an instance of a Computer

### Units of Measure in F#

Just briefly – you'll notice we have a field here called `SizeGb` of type `int`. One of the nice features of F# is *units of measure*. These allow you to quickly create a specific type of *integer* - similar to generics - to prevent accidentally mixing incompatible integers together e.g. GB and MB, or Meters and Feet. In our example, we might have used something like `Size : int<gb>`. We won't be covering them, but I'd recommend looking into them in your own time.

In many ways, this is no different to having classes, with one having a property that is an instance of the second class - except that F# records are much more concise, and we don't fall into the dogmatic "one-file-per-class" approach.

#### 21.1.2 Modelling a type hierarchy

The problem is that so far, we don't have any way of modelling an "is-a" relationship in F#. For example, if we wanted to model different types of hard disks in the OO world, it might look something like this: -

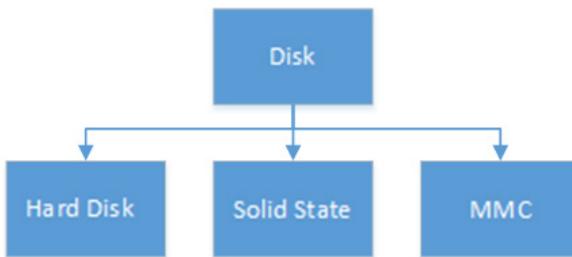


Figure 21.1 – An “is-a” model which might map to an inheritance hierarchy

In the OO world, we'll use inheritance here: -

- A Hard Disk inherits from Disk
- We store “shared” fields in the Disk class
- We store fields unique to the Hard Disk on the Hard Disk class
- Common behaviour is stored in the Disk class
- We allow overriding of common behaviour through polymorphism.

So, we might have our common data represented on the Disk type (such as Manufacturer and Size), but have the RPM speed on Hard Disk and number of pins on MMC. Similarly, we might have a `Seek()` method on the disk, which might work significantly differently across all three

disks. However, the ability to seek to a file is a piece of functionality common to all disks, and so might be implemented using *polymorphism* – having an abstract method on the base class, and then overriding that method in the derived class. Callers then only couple themselves to the base class, without having to worry about which implementation they’re really dealing with. We’ll next look at how we model this same sort of relationship without classes.

## 21.2 Discriminated Unions in F#

The standard functional programming answer to this problem is by using a *discriminated union*. There are other names for it such as *sum types*, *case classes* in Scala or *algebraic data types* for people that want to sound really smart. The best way to think about them is one of two ways: -

1. Like a normal type hierarchy, but one that is *closed*. By this I mean that you define all the different “sub-types” up-front. You cannot declare new sub-types later on.
2. As a form of C#-style Enums, but with the ability to add *metadata* to each enum case.

Let’s take a look at a discriminated union for our fictional three-case disk drive hierarchy: -

### **Listing 21.2 Discriminated Unions in F#**

```
type Disk = #A
| HardDisk of RPM:int * Platters:int #B
| SolidState #C
| MMC of NumberOfPins:int #D

#A "Base type"
#B Hard Disk sub-type, containing two custom fields as metadata
#C SolidState – no custom fields
#D MMC – single custom field as metadata
```

Each case is separated by the pipe symbol (just like pattern matching). If we wish to attach some specific metadata to the case, we separate each value with an asterisk. At this point, it’s worth pointing out that we’ve modelled the equivalent of an *entire* type hierarchy in four lines of code. Compare this with what we’d normally do in C# with a conventional class hierarchy: -

- Create a separate class for the base type and for each subclass
- “Best practice” (allegedly) dictates that we put each subclass into its own file
- Create a constructor for each, with appropriate fields and public properties etc.

### 21.2.1 Creating instances of DUs

#### **Now you try**

Let’s see how we’d now create, and then use, such a discriminated union in F#. Firstly, let’s start by creating a new script `DiscriminatedUnions.fsx` in F#, and then copying across the

discriminated union definition from above. Next, let's create some different types of disks. Creating an instance of a DU case is simple.

```
let instance = DUCase(arg1, arg2, argn)
```

So, let's create an instance a Hard Disk with 250 RPM and 7 platters, followed by an MMC disk with 5 pins. Finally, let's create an SSD disk. As this disk contains no custom parameters, you can do away with the "constructor call" completely. Let's see how this might look: -

### **Listing 21.3 Creating Discriminated Unions in F#**

```
let myHardDisk = HardDisk(RPM = 250, Platters = 7) #A
let myHardDiskShort = HardDisk(250, 7) #B
let args = 250, 7
let myHardDiskTupled = HardDisk args #C
let myMMC = MMC 5
let mySsd = SolidState #D

#A Explicit named arguments
#B Lightweight syntax
#C Passing all values as a single argument, can omit brackets
#D Creating a DU case without metadata
```

How did you get on? Note that in our DU, we've assigned specific names to each metadata field. This is actually optional – you can simply specify the types e.g. `int * string` – but putting in names gives us some documentation, as well as more helpful intellisense.

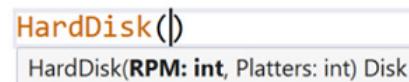


Figure 21.2 – Generated DU “constructors” in F#

Notice also that the left hand side values are all types as the “base type” of `Disk`, rather than the specific subtypes.

```
let myHardDisk = HardDisk(RPM = 250,
 val myHardDisk : Disk
 Full name: Lesson-21.myHardDisk
```

Figure 21.3 – Values are assigned as type `Disk`, not as `HardDisk`.

## 21.2.2 Accessing an instance of a DU

Now that we've created our DU, how do we actually use it? You might try and "dot into" `myHardDisk` and access all the field. If you try this, you'll be left disappointed – you won't get any properties. That's because `myHardDisk` is of type `Disk`, and *not* `HardDisk`. What you need to do is somehow safely "unwrap" this into one of the three subtypes: `HardDisk`, `SolidState` or `MMC` (it's irrelevant that we can "see" in the code that this is really a `HardDisk`. As far as the type system is concerned, it could be any one of the three). How do we safely do this? We use our new-found friend from the previous lesson – pattern matching.

Let's assume we wanted to make some function that handled our hypothetical `Seek()` method from earlier. Recall that in an OO hierarchy, we'd make an abstract method on the base class, and provide implementations on each case. In other words, *all implementations would live with their associated type* - there is no one place you would see *all* of the implementations. In F#, we take a completely different approach: -

### **Listing 21.4 Writing functions for a discriminated union**

```
let seek disk =
 match disk with
 | HardDisk _ -> "Seeking loudly at a reasonable speed!" #A
 | MMC _ -> "Seeking quietly but slowly" #B
 | SolidState -> "Already found it!" #C

seek mySsd #C

#A Match on any type of hard disk
#B Matching on any type of MMC
#C Returns "Already found it!"
```

Of course, as you know with pattern matching, we can also match on specific values within a match case. So we can enhance the above code to match on Hard Disks with an RPM of 5400 and 5 spindles: -

### **Listing 21.5 Pattern Matching on values**

```
| HardDisk(5400, 5) -> "Seeking very slowly!" #A
| HardDisk(rpm, 7) -> sprintf "I have 7 spindles and RPM %d!" rpm #B
| MMC 3 -> "Seeking. I have 3 pins!" #C

#A Matching a hard disk with 5400RPM and 5 spindles
#B Matching on a hard disk with 7 spindles and binding RPM for usage on the RHS
#C Matching an MMC disk with 3 pins
```

This at first glance might appear completely bizarre – we're putting all the different implementations in a single place! Every time we add a new type, we'll have to amend this function! Actually, this is *by design*, and is actually incredibly powerful. Firstly, we get F# to safely check the type of the subclass for us – we can't "accidentally" access the `RPM` field when dealing with a `SolidState` disk; the type system won't let us until we've "matched"

against the appropriate subtype. Next, not only can we use pattern matching to unbind specific values to variables (as in the second case above with RPM) but remember that pattern matching enforces *exhaustive matching*. So, we can use F#'s compiler support to warn us if we've missed out a *specific case*. For example, if we replace the "catch all" hard disk and MMC match cases we had in 18.4 with those from 18.5, we'll see a warning as follows: -

```
warning FS0025: Incomplete pattern matches on this expression. For example, the value
'HardDisk (_, 0)' may indicate a case not covered by the pattern(s).
```

Similarly, if we decided to add a new disk type – say, `UsbStick` – the compiler would instantly warn us that we're not handling that case here. So, we can safely add new types without fear of "forgetting" to handle it. This is all possible because DUs represent a *fixed* type hierarchy – we can't create new sub-types anywhere except where the DU is defined.

### **Now you try**

Let's now write a function which performs some pattern matching over a discriminated union.

1. Create a function, `describe`, which takes in a hard disk.
2. The function should return texts as follows: -
  - o If an SSD, say "I'm a new-fangled SSD."
  - o If an MMC with 1 pin, say "I've only got 1 pin."
  - o If an MMC with less than 5 pins, say "I'm an MMC with a few pins."
  - o Otherwise, if an MMC, say "I'm an MMC with <pin> pins."
  - o If a hard disk with 5400 RPM, say "I'm a slow hard disk"
  - o If the hard disk has 7 spindles, say "I have 7 spindles!"
  - o Otherwise, simply state "I'm a hard disk"
3. Remember to use the wildcard `_` character to help make partial matches e.g. (5400 RPM + "any" number of spindles), and *guard clauses* with the `when` keyword.

---

### **Using wildcards with Discriminated Union matches**

It's tempting to use a plain wildcard for the final case in the last exercise. However, you should always try to be as *explicit as possible* with match cases over discriminated unions. So in the previous example, you would prefer `HardDisk _` rather than simply `_`. This way, if you add a new type to the discriminated union e.g. `UsbStick`, you'll always get warnings from the compiler to remind you to "handle" the new case.

---

### **Quick Check**

1. What is the OO equivalent of discriminated unions?
2. Which language feature in F# do we use to "test" which case of a DU a value is?
3. Can you add new cases to a DU "later on" in your code?

## 21.3 Tips for working with Discriminated Unions

Let's now look at a few best practices for working with DUs now.

### 21.3.1 Nested DUs

You can easily create *nested* discriminated unions - in other words, a type of a type. Let's assume we wanted to create different *types* of MMC drives and make a nested match on that - how would we do it? Firstly, we create our "nested" DU case, and then add that to the original type hierarchy as metadata on the "parent" case (in our situation, that's the MMC case of Disk).

#### **Listing 21.6 Nested Discriminated unions**

```
type MMCDisk = #A
| RsMmc
| MmcPlus
| SecureMMC

type Disk =
| MMC of MMCDisk * NumberOfPins:int #B

match disk with
| MMC(MmcPlus, 3) -> "Seeking quietly but slowly" #C
| MMC(SecureMMC, 6) -> "Seeking quietly with 6 pins."
```

#A Nested DU with associated cases  
#B Adding the nested DU to our parent case in the Disk DU  
#C Matching on both top-level and nested DUs simultaneously

### 21.3.2 Shared fields

We've not yet look at how to share *common* fields across a DU - for example, the manufacturer name of a hard disk, or size. This isn't actually supported with DUs - you can't put "common" fields on the "base" of the DU e.g. the Disk type. The best way to achieve this is by a combination of a Record *and* a Discriminated Union, by create a *wrapper* record to hold both common fields, plus one more field that contains the discriminated union - the "varying" data.

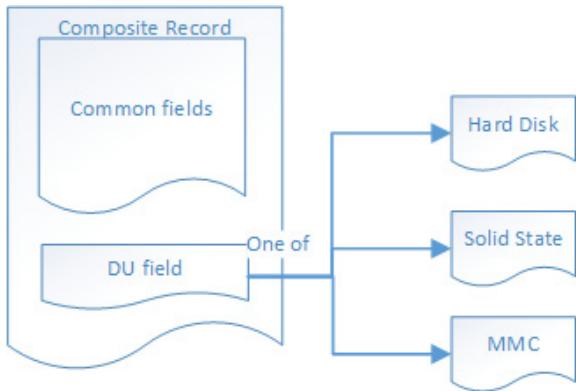


Figure 21.4 – Composing shared fields with varying custom data through a Record and DU

Here's a code sample that's evolved from the start of this lesson, which models both shared and varying data: -

#### **Listing 21.7 Shared fields using a combination of records and discriminated unions**

```

type DiskInfo =
{ Manufacturer : string #A
 SizeGb : int
 DiskData : Disk } #B
type Computer = { Manufacturer : string; Disks : DiskInfo list } #C
let myPc =
 { Manufacturer = "Computers Inc."
 Disks =
 [{ Manufacturer = "HardDisks Inc." #D
 SizeGb = 100
 DiskData = HardDisk(5400, 7) } #E
 { Manufacturer = "SuperDisks Corp."
 SizeGb = 250
 DiskData = SolidState }] }

```

#A Composite record, starting with common fields

#B Varying data with field as DU

#C Computer record – contains manufacturer and a list of disks

#D Creating a list of disks using [ ] syntax

#E Common fields + varying DU as a Hard Disk

You can actually visualise the above code easily. Consider how simple it is to map the code above directly to the following figure.

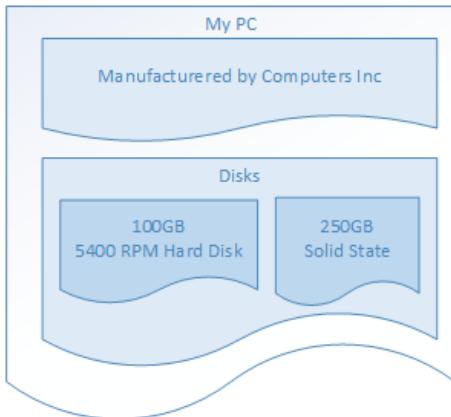


Figure 21.5 – Representing our data model visually

### Active Patterns

Very briefly, F# has an even more powerful - and lightweight - mechanism for classification of data called Active Patterns. They're a more advanced topic, but I would recommend you check them out in your own time – you can think of them as discriminated unions on steroids.

### 21.3.3 Printing out DUs

A quick tip – if you ever just want to print out the contents of a DU in a human readable form, instead of manually matching over all cases and generating a `sprintf` for each one, you can simply call `sprintf "%A"` on a DU – the compiler will pretty print the entire case out for you!

#### Quick Check

4. How do you model “shared” fields in a discriminated union?
5. Can you create one discriminated union with another one?

## 21.4 More about Discriminated Unions

### 21.4.1 Comparing OO hierarchies and Discriminated Unions

Let's finish off looking at DUs by comparing them with OO-style inheritance.

**Table 21.1 – Comparing Inheritance and DUs**

|       | Inheritance | Discriminated Unions |
|-------|-------------|----------------------|
| Usage | Heavyweight | Lightweight          |

|                                  |                      |                                 |
|----------------------------------|----------------------|---------------------------------|
| <b>Complexity</b>                | Hard to reason about | Easy to reason about            |
| <b>Extensibility</b>             | Open set of types    | Closed set specified together   |
| <b>Useful for plugin models?</b> | Yes                  | No                              |
| <b>Add new sub-types</b>         | Easy                 | Update all DU-related functions |
| <b>Add new methods</b>           | Breaking change      | Easy                            |

The hard and fast rule is - if you need to have an extensible set of open, pluggable subtypes that can be dynamically added, discriminated unions are not a great fit: Discriminated Unions are fixed at compile time, so you can't plug in new items easily.

For large number-of-cases (100s) DUs that change quickly, also think carefully - every time you add a new case, your pattern matches over the DU will need to be updated to handle the new subtype (although the compiler will at least tell you where you need to update your code!). In such a case, either a Record or raw functions might be a better fit, or falling back to a class-based inheritance model.

However, if you have a fixed (or slowly changing) set of cases – which in my experience is appropriate the vast majority of the time – then a DU is a *much* better fit. They're lightweight, easy to work with, and very flexible, as you can add new behaviours extremely quickly without affecting any the rest of your codebase and get the benefit of pattern matching. They're also generally *much easier* to reason about – having all implementations in a single place leads to much easier to understand code.

#### 21.4.2 Creating Enums

The last point I'll make here is on Enums. You *can* create standard .NET enums in F# easily enough – the syntax is somewhat similar to a DU: -

##### **Listing 21.8 Creating an Enum in F#**

```
type Printer = #A
| Injket = 0 #B
| Laserjet = 1
| DotMatrix = 2

#A Enum type
#B Enum case with explicit ordinal value
```

The only differences are that you must give each case an explicit ordinal, and you can't associate metadata with any case. And, whilst you can pattern match over an enum easily enough, enums cannot be *exhaustively* matched over, as you can cast any `int` to an enum (even if there's no associated enum case!). Therefore, you will *always* need to add a "catch all" wildcard handler to an enum pattern match in order to avoid a warning. This is something that can often catch us out – C# won't warn you about this, which can lead to many classes of bugs.

### Quick Check

6. When should you not use a discriminated union?
7. Why do you need to always place a wildcard handle for enums?

## 21.5 Summary

That's a wrap on this lesson! We covered: -

- Comparing DUs to class hierarchies
- When and when not to use DUs

Discriminated Unions are a really powerful tool in F#'s arsenal of features to help us model domains quickly and easily. We'll be using them for the next two lessons, so don't worry if it feels a little foreign to you - you're about to get much more comfortable with it!

### Try This

Take any example domain model you have recently written in OO and try to model it using a combination of discriminated unions, tuples and records. Alternatively, try to update the Rules Engine we looked at earlier in the book, so that instead of returning a tuple of the rule name and the error, it returns a Pass or Fail discriminated union, with the Failure case containing the error message.

### Quick Check Answers

1. Inheritance and Polymorphism.
2. Pattern matching.
3. No. DUs are closed, and fixed at compile-time.
4. Through a composite record that contains the common fields and the DU value.
5. Yes, DUs can be embedded (or *nested*) within one another.
6. For plug-in models or for unstable (or rapidly changing), extremely large hierarchies.
7. Any integer value can be cast to an enum without a runtime error.

# 22

## *Fixing the billion-dollar mistake*

Hopefully, in the last lesson you gained at least an initial appreciation of discriminated unions and how they allow us to quickly and easily model complex relationships. In this lesson we'll take a look at one specific of discriminated union that comes built into F# designed to solve a single problem: nothing! We'll learn about: -

- How we deal with “absence of value” situations in .NET today
- Working with optional data in F#
- Helper F# functions to deal with common optional scenarios

### 22.1 Working with missing values

Imagine we were reading a JSON document from a car insurer that contains information on a driver, including their “safety rating” that’s used to calculate their insurance premiums. Better drivers get a positive score (and lower premium), whilst poor drivers get a negative score. Unfortunately, the data is unreliable - so sometimes the JSON document won’t contain the safety score for a driver. In such a case, we’ll send a message to the data provider to request it at a later date again, and assign a temporary premium price of \$300: -



Figure 22.1 – Annual insurance premiums on a sliding scale

#### **Listing 22.1 Example JSON document with missing data**

```
{
 "Drivers" :
 [{ "Name" : "Fred Smith", "SafetyScore": 550, "YearPassed" : 1980 }, #A
 { "Name" : "Jane Dunn", "YearPassed" : 1980 }] } #B
```

```
#A Driver with a safety score of 550
#B Driver without a safety score
```

We deal with data like this all the time today - the problem is how we normally *reason about this* in .NET. Ideally, what we would like is the ability to encode into our C# class that fact that the `Name` and `YearPassed` fields are mandatory, and always populated - whilst `SafetyScore` might *sometimes* be missing and is therefore optional. Unfortunately, in C# and VB .NET today, we divide our types into two *types*, each of which behave differently in this sense: -

**Table 22.1 – Mandatory and Optional values in C# and VB .NET**

| Data Type | Example           | Support for “mandatory” | Support for “optional” |
|-----------|-------------------|-------------------------|------------------------|
| Classes   | String, WebClient | No                      | Yes                    |
| Structs   | Int, Float        | Yes                     | Partial                |

- **Classes** should always be considered optional, because we can set them to null at any point in time; it's impossible in the C# type system to indicate that a string can *never* be null.
- **Structs** can never be marked as null, but have a default value instead. For example, an integer will be set to 0 by default. There *are* some ways that we can handle optional data - however, these are achieved at the *library* level, and not within the *language and type system*.

If you think about it, this is bizarre. Why do we divide up the way we can reason about "missing data" based on the structs and classes? Surely we would like to unify this, so that all types of data can be marked as *either* mandatory or optional? Before we look at that some more, lets discuss both classes and structs in a little more detail.

### 22.1.1 The rise of the billion-dollar mistake

I'm pretty sure that at least 99% of you have seen this window many, many times in Visual Studio: -

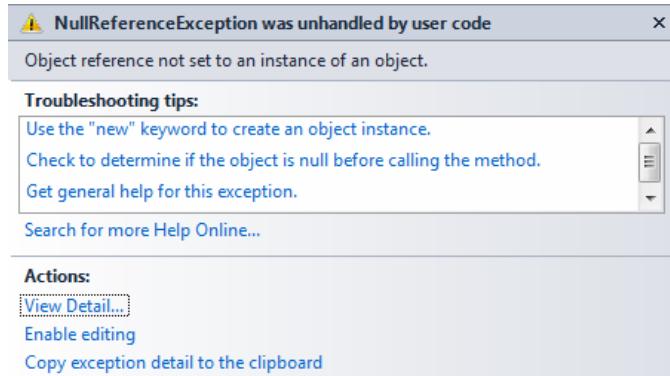


Figure 22.2 – A null reference exception dialog in Visual Studio

We've all seen null reference exceptions before. They crop up in places we never think should happen and cost us huge amount of time and effort – not only to fix them, but simply to diagnose why they happened in the first place. Yet it turns out that the whole concept of null was ultimately introduced as a sort of hack into the Algol programming language years ago.

I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

[https://en.wikipedia.org/wiki/Null\\_pointer#History](https://en.wikipedia.org/wiki/Null_pointer#History)

I would estimate that the cost of null reference exceptions and associated bugs at more than a billion dollars, many times over. And it's not just Sir Hoare - even members of the C# compiler team have said that they regret putting null into the language. The problem is that now it's in, it's almost impossible to remove. Here's an example of the sort of thing that the .NET type system *should* be able to prevent: -

#### **Listing 22.2 Breaking the type system in .NET**

```
string x = null; #A
var length = x.Length; #B
```

#A Creating a null reference to a string  
#B Accessing a property on a null object

We know that this code will fail! However, the type system can't help us out here, because in C# or VB .NET, *any* class object can be assigned null, at *any* time – so theoretically we should always test before accessing one. Because we can't mark a class object as mandatory, you'll probably have seen or written code like this many times: -

**Listing 22.3 Checking for nulls in C#**

```
public void Foo(string test) #A
{
 if (test == null) #B
 return

 // Main logic here...

#A Input argument - nullable class
#B Manual check if value is null
```

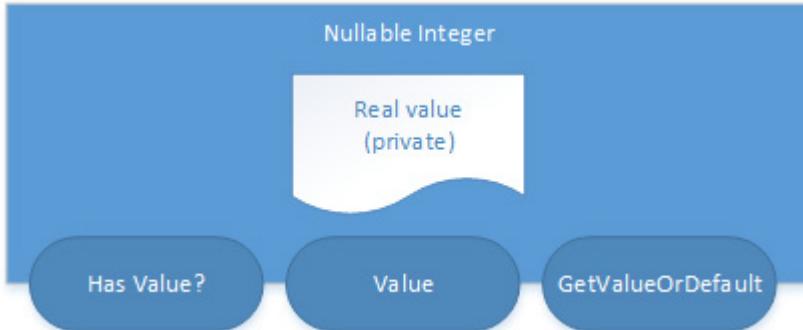
The problem here is – where do you draw the line at these sorts of null checks? In a perfect world, they should just happen at the *boundaries* of your application – essentially, only when taking in data from external systems – but we know that it’s common to end up with these sorts of null checks scattered throughout our codebase. It’s not particularly satisfactory, and worse still, it’s not safe – you probably won’t correctly figure out up-front *all* of the cases where there’s *really* a chance of getting a null reference exception.

**C# vNext and nulls**

There’s a large effort on the C# team to introduce some form of null checking support – essentially, it’ll be some kind of code analyser that performs branch analysis of your code to “prove” that you can safely access a class object. However, this is quite different from F#, which actually removes the whole notion of nullability from the type system, as you’ll see.

**22.1.2 Nullable Types in .NET**

So, we’ve shown where the standard class-based system is in .NET lets us down. Conversely, structs seem a step in the right direction: by default, they’re mandatory. But we obviously need to model “optional” values as well – how do we do this for integers, for example, when we don’t have null? There are few ways that we normally work around this, from picking a “magic” number that hopefully doesn’t clash with genuine values e.g. -1, to having an extra Boolean `IsValueSet` property on our class which we first need to check before going to the “real” value. Neither choices are satisfactory. In fact, this situation is so common that Microsoft developed the Nullable Type in .NET 2 to help. A nullable type is a *wrapper* that extends around a “real” struct value which *may* or *may not* exist.



**Figure 22.3 – Representation of a Nullable Type**

A `Nullable` holds an encapsulated “real” value (which might not be set), and provides a simple API on top of it: You can ask it whether there’s a value or not via the `HasValue` property, and if that returns true, you can then access the `Value` property. This is nicer than simple null checking with classes – at least here you can explicitly distinguish between times when a value can *never* be null (a standard integer) and those that *might* be missing (a nullable integer). Unfortunately, `Nullable` types can only wrap structs, not classes. Also, whilst `Nullable` objects add a convenience for us via an API, it doesn’t provide us with *type safety* around them – if you go straight to the `Value` property without first checking the `HasValue` property, you still run the risk of getting an exception.

### Quick Check

1. Why can’t C# prevent obvious null references?
2. How does the `Nullable` type improve matters when working with “might be missing” data?

## 22.2 Improving matters with the F# Type System

The simple truth is that the .NET type system itself isn’t geared up towards allowing us to easily and consistently reason about mandatory and optional data. Let’s now see how F# addresses these types of data.

### 22.2.1 Mandatory data in F#

First of all, unlike the inconsistency with classes and structs, in F# *all F# types* (that’s Tuples, Records and Discriminated Unions) behave in the same way in that they are all mandatory by default. In other words, *it is illegal to assign null to any F# type value*. Let’s see an example of this based on the computer model from the previous lesson: -

**Listing 22.4 Trying to set an F# type value to null**

```
let myMainDisk =
 { Manufacturer = "HardDisks Inc."
 SizeGb = 500
 DiskData = null } #A
```

#A Setting DiskData to null causes a compile-time error

**DiskData = null } ] }**

The type 'Disk' does not have 'null' as a proper value

Figure 22.4 – Trying to set the DiskData field to null in F#

Recall that records cannot be created with only “some” of the fields assigned, and you can’t “miss out” fields when creating discriminated unions. This means immediately that any code that only uses F# types cannot get a null reference exception – there’s simply no notion of null in the type system.

**Beating the F# type system**

There are corner cases in which you can get null reference exceptions with F# types, using various attributes, and some interoperability scenarios. However, for 99% of the time, you can forget about them – for day-to-day programming, it won’t be an issue. Also note that F# will not stop you assigning null to standard classes (although we’ll see later that there are some ways to improve these situations, too).

**F# Types at runtime**

All three F# types – Tuples, Records and Discriminated Unions – boil down to classes at runtime, and are therefore reference types. However, the next version of F# will introduce support to compile them down to value types instead. This has no impact of nullability at compile-time, but will be rather for specific performance and interop scenarios.

## 22.2.2 The Option Type

So, if we’ve just established that F# values can never be null and are “mandatory by default”, we’ll need some way to deal with optional data such as in 19.1 – and so F# has *option type* (also known as *Maybe* in some languages). You can think of it as a version of *Nullable*, except that it’s more flexible (in that it can work not only on F# types, but also classes and structs) and has *language support* to make it easier for us to more safely work with both “has a value” and “no value” cases. Here’s how we implement “optional” data in F#:

**Listing 22.5 Sample code to calculate a premium**

```

let aNumber : int = 10
let maybeANumber : int option = Some 10 #A

let calculateAnnualPremiumUsd score =
 match score with
 | Some 0 -> 250 #B
 | Some score when score < 0 -> 400
 | Some score when score > 0 -> 150
 | None -> #C
 printfn "No score supplied! Using temporary premium."
 300

calculateAnnualPremiumUsd (Some 250) #D
calculateAnnualPremiumUsd None

#A Creating an optional number
#B Handling a safety score of (Some 0)
#C Handling the case when no safety score is found
#D Calculating a premium with a wrapped score of (Some 250) and then None

```

Option is actually just a simple two-case discriminated union – `Some (value)` or `None`. Just like other discriminated unions, we pattern match over it in order to deal with all the cases, and unlike null reference checks with classes, here we must *explicitly* handle both cases (value and no value) up-front at compile time. We can't “skip out” the null check.

Also, notice that when we call this function we can no longer simply pass in 250 – we must first “wrap it” as `Some 250` (just like we “wrapped” the number of pins in an MMC disk in the previous lesson). This is slightly different to working with nullables in C#, where the compiler will silently wrap an integer in a nullable int for you – F# is a little stricter here, and you must explicitly wrap the number yourself.

**Now you try**

Let's try to model the data set from Listing 22.1.

1. Create a record type to match the structure of the customer.
2. For the “optional” field’s type, use an option (either `int option` or `Option<int>`)
3. Create a list which contains both customers using `[ a; b ]` syntax.
4. Change the function in Listing 22.5 to take in a full customer object and match on the `SafetyScore` field on it.

**Optional escape hatches**

If you “dot into” an option discriminated union, you'll see three properties on them: `IsSome`, `IsNone` and `Value`. The first two are *sometimes* useful, but generally you should favour pattern matching or helper functions (see 22.5). The latter field, `Value`, allows you to go straight to the value of the object without even checking if it exists. If it doesn't exist, you'll get a null reference exception – exactly what we're trying to avoid! Don't ever use this. Instead, use pattern matching to force you to deal with both `Some` and `None` cases in your code *up front*.

### Quick Check

3. Can you get null reference exceptions in F#?
4. How should you safely dereference a value that is wrapped in an option?

## 22.3 The Option module

There are many common scenarios related to working with options, and some associated helper functions that come with F# in the `Option` module. Most of them do the same thing – they take in an optional value, do some operation (mapping, filtering etc.) and then return back another optional value. It may help you to keep that in mind as we go through this section that many of these functions are similar to their equivalents in the collection modules.

### 22.3.1 Mapping

`Option.map` allows us to “map” an optional value from one *kind* of option to another by means of a mapping function: -

```
mapping:('T -> 'U) -> option:'T option -> 'U option
```

It actually performs a similar purpose as `List.map`, which you already know: -

```
mapping:('T -> 'U) -> list:'T list -> 'U list
```

In other words, it’s a *higher order* function that takes in some optional value and a mapping function to act on it - but only calls `mapping` if the value is `Some`. If the value is `None`, it does nothing. This is similar to how `List.map` only call the mapper if there’s at least one item in the list – if it’s an empty list, nothing happens. And just like `List.map`, the mapping function *doesn’t* have to have to know about (in this case) options – the act of checking is taken care for us.

Let’s go through an example. Imagine your colleague has written a function, `describe`, that describes the safety score of a driver e.g. “Safe” or “High risk”. It’s not designed to work with optional scores, but you want to run it against the optional safety scores from our JSON file. We could either use a pattern match on this, or we can use `Option.map`: -

#### Listing 22.6 Matching and Mapping

```
let description =
 match customer.SafetyScore with
 | Some score -> Some(describe score) #A
 | None -> None

let descriptionTwo =
 customer.SafetyScore
 |> Option.map(fun score -> describe score) #B

let shorthand = customer.SafetyScore |> Option.map describe #C
let optionalDescribe = Option.map describe #D
```

```
#A A standard match over an option
#B Using Option.map to act on the Some case
#C Shorthand to avoid having to explicitly supply arguments to describe in Option.map
#D Creating a new function that safely executes describe over optional values
```

All three expressions do the same thing – to only run `describe` if `SafetyScore` is `Some` value, and otherwise do nothing. `Option.map` is especially useful because you can write entire reams of code without having to worry about optional data – you can then simply chain them together and wrap them up in `Option.map` to get back a new function that does the “option check” for you for free. This is known as “lifting” a function.

Also, keep an eye out for `Option.map`’s sibling, `Option.iter` – as with `List.iter`, you can use this for “side-effectful” functions that return `unit`, such as printing out an optional customer’s name to the screen using `printfn`.

### 22.3.2 Binding

`Option.bind` is the same as `Option.map`, except it works with mapping functions that *themselves* return options!

```
binder:('T -> 'U option) -> option:'T option -> 'U option
```

`Bind` is more or less the equivalent to `List.collect` – it can “flatten” an `Option<Option<string>>` to just `Option<string>`, just like `collect` can “flatten” a `List<List<string>>` to `List<string>`. This is very useful if you chain multiple functions together each of which return an option: –

#### Listing 22.7 Chaining functions that return option with `Option.bind`

```
let tryFindCustomer cId = if cId = 10 then Some drivers.[0] else None
let getSafetyScore customer = customer.SafetyScore #A
let score = tryFindCustomer 10 |> Option.bind getSafetyScore #B
```

```
#A Two functions that each return an option value
#B Binding both functions together
```

Try this one out yourself – observe that if you replace the call to `Option.bind` with `Option.map`, you’ll get back an `Option<Option<int>>` – `bind` protects us here by doing the “double unwrap” for us.

### 22.3.3 Filtering

You can also *filter* an option using `Option.filter`. In other words, run a predicate over some optional value. If the value was `Some`, run the predicate – if it passes, keep the optional value. Otherwise, return `None`.

```
predicate:('T -> bool) -> option:'T option -> 'T option
```

### Listing 22.8 Filtering on options

```
let test1 = Some 5 |> Option.filter(fun x -> x > 5) #A
let test2 = Some 5 |> Option.filter(fun x -> x = 5) #B

#A test1 equals None
#B test2 is equal to (Some 5)
```

#### 22.3.4 Other Option functions

Here's a quick summary of some of the other optional functions. Have a look at them (and the others in the Option module) in more detail in your own time – some of them are very handy.

| Function      | Description                                                                           |
|---------------|---------------------------------------------------------------------------------------|
| Option.count  | If optional value is None, returns 0, otherwise returns 1                             |
| Option.exists | Runs a predicate over an optional value and returns the result. If None, return false |

#### Quick Check

5. When should you use `Option.map` rather than an explicit pattern match?
6. What is the difference between `Option.map` and `bind`?

## 22.4 Collections and Options

You might have already noticed that there's a kind of "symmetry" between the Collections and Options modules – they both have some similar functions such as `map`, `filter` and `count` etc. There are also a set of functions to *interoperate* between them.

#### 22.4.1 Option.toList

`Option.toList` (and its sibling, `Option.toArray`) take an optional value in, and if it's Some value, returns a list with that single value in it. Otherwise, it returns an empty list. This isn't always needed, but it's sometimes handy to be able to treat an optional value as a list (or array).

#### 22.4.2 List.choose

`List.choose` on the other hand is a very useful function – you can think of it as a specialised combination of `map` and `filter` in one. It allows you to apply a function that *might* return a value, and then automatically strips out any of the items that returned `None`.

```
chooser:('T -> 'U option) -> list:'T list -> 'U list
```

### Now you try

Let's imagine we had a database of customers with associated IDs, and a list of customer ids. We want to load the names of those customers from the database – but we're not sure if all of our customer ids are valid. How can we easily get back just those customers that exist?

1. Create a function `tryLoadCustomer` that takes in a customer id. If the id is between 2 and 7, return an *optional* string "Customer <id>" e.g. "Customer 4". Otherwise, return `None`.
2. Create a list of customer ids from 0 to 10.
3. Pipe those customer ids through `List.choose`, using the `tryLoadCustomer` as the higher order function.
4. Observe that you have a new list of strings, but only for those customers that existed.

### Options, Lists and Results

We're treading dangerously close to the M-word (that's Monad) here. If you found this idea of "symmetry" between lists and options (and of safely working with Options) interesting, it's definitely worth your while reading about "Railway Oriented Programming" on the F# for Fun and Profit website (<http://fsharpforfunandprofit.com/>) by Scott Wlaschin. Not only does it go into more depth on maps, binds and lifting than here – it gives a relatively easy-to-understand introduction on working with monads.

#### 22.4.3 “Try” functions

You'll see through the collections modules functions that start with `try`, such as `tryFind`, `tryHead` and `tryItem`. Think of these as equivalent to LINQ's `OrDefault` functions, except instead of returning `null` if the function doesn't have any output, these functions all return an `Option` value – `Some` value if something was found, and `None` otherwise.

#### Quick Check

7. Why are collection `try` functions safer to use than LINQ's `OrDefault` methods?

#### 22.5 Summary

We saw in this lesson how two features that we've learned earlier in this unit – pattern matching and discriminated unions – can be combined to provide a typesafe, reasonable way of dealing with absence of value, without resorting to nulls. We saw: -

- How we deal with absence-of-value in C#
- The `option` type in F#
- The `Option` helper module

### Try This

Write an application which displays information on a file on the local hard disk. If the file is not found, return `None`. Have the caller code handle both scenarios and print an appropriate response to the console. Or, update the rules engine code from previous lessons so that instead of returning a blank string for the error message when a rule passes, it returns `None`. You'll have to also update the "fail" case to some `Some` error message!

### Quick Check Answers

1. The type system does not support the notion of "non-nullability".
2. Members such as `HasValue` and `GetValueOrDefault` provide some control over nullable values.
3. Yes, when working with BCL types. F# types normally do not permit nulls.
4. As `Option` is just a discriminated union, use pattern matching to cater for both branches.
5. If the "None" case simply returns `None`, you can replace it with `Option.map`.
6. Bind should be used when the mapping function *itself* returns an `Option`.
7. `try` functions return option values, rather than nulls as with `defaultOf`.

# 23

## *Business Rules as Code*

**In the last lesson of this unit, we'll see how we can use F# language features such as Records, Options and Discriminated Unions to write code that can enforce business rules *within code*.**

**We'll:** -

- Discuss how we validate business rules today
- Look at domain modelling in F# more closely
- Explore single case discriminated unions
- Understand how to encode business rules through types
- Discuss exception handling

Our code always has some form of business rules within it. Generally, we validate that our code is correct by either running the application and manually seeing if it "does the right thing", or we write some form of automated test suite that sits "alongside" our code. This test suite often is as large as the "real" code base itself, and runs tests in code that check the results of the application.

There are, of course, limits to what these tests should and shouldn't do. As C# is a statically typed language, there are certain tests that we don't need to perform, because the compiler gives us certain *guarantees*. For example: -

- We don't ever need to check that the `Name` property on a `Person` class, which is a `string`, contains an `integer` – the type system provides that for us for free
- We don't ever need to check that an `int` is null

However, here are some things that we *might* want to write automated tests for: -

- Ensure that we don't mix up the `AgeInYears` and `HeightInCm` values, both of which are integers
- Check that an object can never get into an "illegal" state
- Check that if we call a method on a class with "invalid" data that we correctly reject it
- This lesson will show us some examples of these sorts of cases, and how to model

them using F#'s type system in such a way as to make *illegal states unrepresentable*. In other words, just like C# makes it "illegal" to store an integer in a string field, F# can take this a step further, allowing us to start to model business rules *in code*, so that it's impossible to represent an "illegal" state. This means our unit tests are a lot simpler, or ideally can be *completely omitted*.

We've already seen some examples of domain modelling in F# in the last few lessons, such as modelling the parts in a computer, or how to accurately model and deal with "absence of a value". We'll now look at taking this a step further within the context of a simple scenario. Let's assume that we want to model our customer's *contact details* in code that adheres to a number of simple rules. We'll take each rule one at a time, and see how our model evolves throughout the code.

## 23.1 Specific types in F#

We'll start by defining a simple customer record to fulfil our first requirement: -

Listing 23.1 A customer can be contacted by email, phone or post

**Listing 23.1 A sample F# record representing a sample Customer**

```
type Customer =
 { CustomerId : string
 Email : string #A
 Telephone : string
 Address : string }
```

#A Storing all possible contact detail values as three separate fields

We're using an F# record here, but you could imagine this to be a C# class – it'd look fairly similar. Yet it's not a great fit though for the above requirements. For example, any of the contact details could be filled in – or all of them – or none! You might say that this is completely reasonable. We might write some unit tests to "prove" to us that we never set more than one, and perhaps a get-only property that will tell us which of the three to use later on. But fundamentally we're relying on writing some code to test some other code in the same way a JavaScript developer might write some code to "prove" that a string property isn't assigned an integer. How can we more accurately model this?

### 23.1.1 Mixing values of the same type

One thing that strikes me from the above example is that all four fields use the *same type* – a string. Here's an example of a function that can create a customer: -

**Listing 23.2 Creating a Customer through a helper function**

```
let createCustomer customerId email telephone address =
 { CustomerId = telephone
 Email = customerId }
```

```

 Telephone = address
 Address = email }
let customer =
createCustomer "C-123" "nicki@myemail.com" "029-293-23" "1 The Street"

```

I hope you caught the errors in that sample: we've accidentally mixed up the assignments for a few of the fields! Because we're using the same simple type – `string` – for all the fields, we get no help here from the compiler. We'll probably find out that we've messed up here if we had written some unit tests, or perhaps some days later when we check the database and see we're storing data in the wrong columns. Doh!

### 23.1.2 Single Case Discriminated Unions

F# has a rather nice way of solving this called *single case* DUs. What's the point of a DU that only has a single possibility? Because of the simple syntax that they provide, we can use them as simple *wrapper classes* to prevent accidentally mixing up values. Here's the syntax for working with single case DUs: -

#### **Listing 23.3 Creating a wrapper type via a single case discriminated union**

```

type Address = Address of string #A
let myAddress = Address "1 The Street" #B
let isTheSameAddress = (myAddress = "1 The Street") #C
let (Address addressData) = myAddress #D

#A Creating a single case DU to store a string Address
#B Creating an instance of a wrapped Address
#C Comparing a wrapped Address and a raw string will not compile
#D Unwrapping an Address into its raw string as addressData

```

As you can see, compared to multi-case DUs, when defining a single case DU, we can omit the pipe to separate cases – and even put it all on a single line. As it's obvious what the contents of the discriminated union are, we can also omit the name of the value argument i.e. `string` instead of `address:string`. When we then want to create an instance of it, again because it's a single case, we get nice lightweight syntax of simply the type name and then the value.

Note that once we've "wrapped" up a value in a single case DU, we can't compare a "raw" value with it – we need to either "wrap" a raw value into the `Address` type, or "unwrap" the `Address`. We do this in the final line, where the raw string is put into the `addressData` value (notice that as this is a single case DU, we don't have to bother with pattern matching).

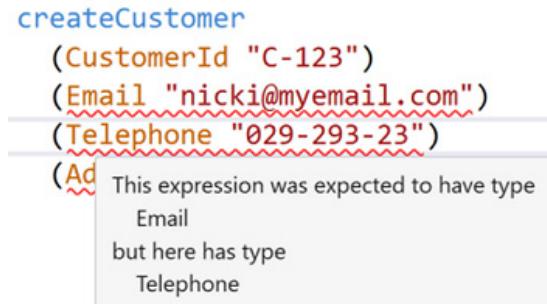
#### **Now you try**

Let's enhance our domain model so that we can't "accidentally" mix and match the values for the different fields. Start with an empty script, creating the `Customer` record and `createCustomer` function (*with* the incorrect assignments).

1. Create four single case discriminated unions, one for each `type` of string we want to store (`CustomerId`, `Email`, `Telephone` and `Address`)

2. Update the definition of the `Customer` type so that each field uses the correct wrapper type. Make sure you define the wrapper types before the `Customer` type!
3. Update the callsite to `createCustomer` so that you “wrap” each value into the correct DU. Note that you’ll need to surround each “wrapping” in parentheses (see Figure 23.1)

If you’ve done this correctly, you’ll notice that your code immediately stops working: -



**Figure 23.1 – A single case DUs can protect us from accidentally assigning values incorrectly**

Interestingly, we’ve received a compiler error on the `callsite` to `createCustomer` – this is actually a case of “following the breadcrumbs” with type inference – if you “mouse over” any of the arguments to the function itself, you’ll see that this is because we’ve mixed up the assignments to the wrong fields.

4. Fix the assignments in the `createCustomer` function and you’ll see that as if by magic all the errors disappear.

#### **Listing 23.4 Creating wrapper types for contact details**

```
type CustomerId = CustomerId of string #A
type Email = Email of string
type Telephone = Telephone of string
type Address = Address of string

type Customer =
 { CustomerId : CustomerId
 Email : Email #B
 Telephone : Telephone
 Address : Address }
```

```
#A Creating a number of single-case DUs
#B Using single case DUs in the Customer type
```

In addition to obviously trapping this error immediately, there’s also another benefit of using single case DUs: it’s now much easier to understand what a value *represents* rather than e.g. a raw string. We no longer need to rely on the *name* of a value e.g. `theAddress` but can now also use the *type itself* to indicate the use of the value – a `CustomerId` or `Address` etc.

### Give and take with the F# compiler

Using types to guide the compiler is a much *quicker* way of getting feedback than writing a unit test or similar – and it's a much *stronger* test. A unit test is written by a team of developers; they might be written inconsistently or have mistakes in them. A compiler is a program that gives us consistent behaviour - *quickly*. Relying on the compiler here by adding some “wrapper types” gives the compiler much more information about what we’re trying to do – and so in turn it can help us more by providing us with more guidance when we’ve done something we shouldn’t have.

Remember to always “wrap” a value in a DU at the earliest opportunity e.g. when loading in data from a text file or database, along with appropriate validation. Once it’s “inside” our domain model, we don’t have to revalidate it etc. ever again, and we only ever “unwrap” a DU to its raw contents when we need to perform some operation on the raw contents. Ideally, you’ll have all these related functions in a shared module. For example, you may have a function to “create” a `Telephone` from a `string`, which performs some regex validation on the raw string before safely returning a `Telephone`.

### Quick Check

1. What’s the benefit of single case DUs over raw values?
2. When working with single case DUs, when should you “unwrap” values?

### Answers

1. Type safety – impossible to accidentally mix and match values of different types. Also semantic meaning of values through types.
2. Only unwrap values when you need to access the raw contents – not before.

### Wrappers with C#

In truth, you *can* create similar sorts of functionality in C# or VB .NET by creating a wrapper class for each type e.g. `Telephone`, `Address` or `CustomerId`. The truth is that we rarely use them because of the overhead of doing this – creating a constructor and a public property on a class in a new file etc. Single-case discriminated unions are much simpler to both create and access, with a much lighter syntax. [Combining Discriminated Unions](#)

By moving to discriminated unions, we can be sure that we don’t accidentally mix up the wrong fields, and at the same time also know that none of our fields can ever be null (as DUs can never be assigned null). However, we still haven’t solved the task completely: we want to model that only *one* of the contact details should be allowed at any point in time.

### Now you try

Let’s merge all three of our our single case DUs into a *multi-case* DU and change our `Customer` type to only hold a field of `ContactDetails` instead of one field for each type of contact detail.

```
type ContactDetails =
| Address of string
```

```
| Telephone of string
| Email of string
```

1. Replace the three single-case DUs with the new `ContactDetails` type.
2. Update the `Customer` type by replacing the three optional fields with a single field of type `ContactDetails`.
3. Update the `createCustomer` function – it now only needs to take in two arguments, the `CustomerId` and the `ContactDetails`.
4. Update the callsite as appropriate e.g.

```
let customer =
 createCustomer (CustomerId "Nicki") (Email "nicki@myemail.com")
```

We can now guarantee that one and only one type of contact is supplied e.g. `Telephone`.

### 23.1.3 Using Optional values within a domain

This next requirement should be fairly simple: -

1. *Customers should have a mandatory primary contact detail and an optional secondary contact detail*

Add a new field to our `Customer` which contains an optional `ContactDetail`, and rename our original `ContactDetail` field to be `PrimaryContactDetails`.

#### **Listing 23.5 Adding an option field for optional secondary contact details**

```
type Customer =
 { CustomerId : CustomerId
 PrimaryContactDetails : ContactDetails
 SecondaryContactDetails : ContactDetails option } #A
```

#A Adding an optional field for secondary contact details

2. Update the `createCustomer` function and callsite as appropriate.

Now we're really starting to make some headway. We'll never have to null check this customer's primary contact details, and have modelled our data in such a way that we can only have one of the three types at once. We've also modelled an *optional* secondary contact details, and would use pattern matching to safely handle both value and absence-of-value cases.

#### **Quick Check**

- 3 What's the benefit of single case DUs over raw values?
- 4 When working with single case DUs, when should you "unwrap" values?

## 23.2 Encoding business rules with marker types

That was a pretty easy change. Let's now look at our final change, which is perhaps the most challenging – and interesting.

*Customers should be validated as genuine customers, based on whether their primary contact detail is an email address from a specific domain. Only when customers have gone through this validation process should they receive a welcome email. Note that we also will need to perform further functionality in future based on whether a customer is genuine or not.*

A simplistic solution to this might look something like this: -

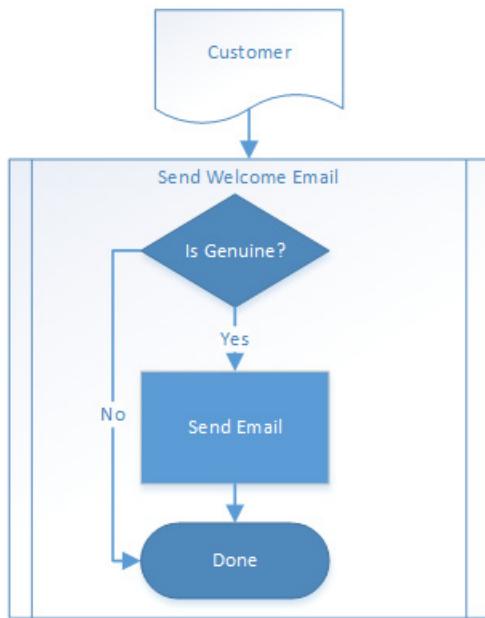


Figure 23.2 – A typical procedural piece of logic.

And that might work just fine. We'll probably write a couple of unit tests that takes in a Customer and ensure that we only send the welcome email for whatever we define as a genuine customer. Maybe we'll even separate the "is a genuine customer" into some helper function so that we can reuse it later. However, there's another way to do this: -

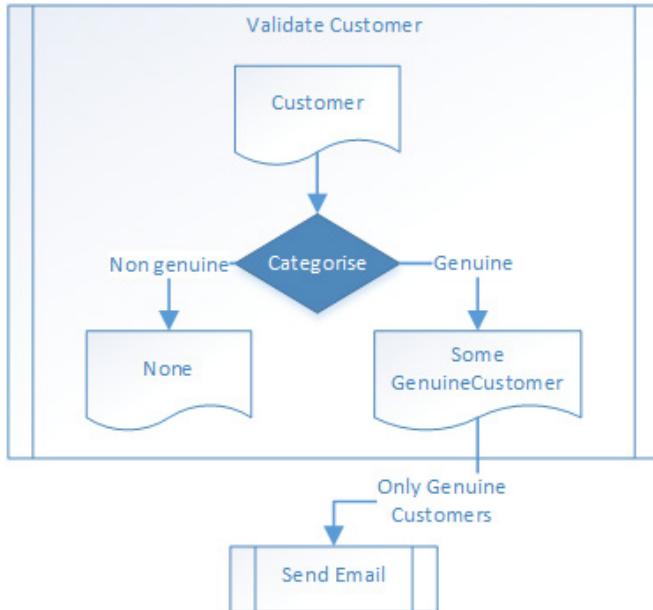


Figure 23.3 – Defining custom states with types to prevent illegal cases.

In this version, we create a function called `ValidateCustomer` that takes in a raw, “unknown” customer, and returns back out a new “genuine” customer – a *new type* which we can treat differently from the raw one. By doing this, we can now distinguish between an “unvalidated” customer and one which has been confirmed as genuine. As we’ll see shortly, this can be useful in a number of ways, but firstly let’s see what that looks like in code: -

#### **Listing 23.6 Creating custom types to represent business states**

```
type GenuineCustomer = GenuineCustomer of Customer #A
```

```
#A Single case DU to wrap around Customer
```

All we do here is create a single case DU which acts as a “marker” type – it wraps around a *standard* `Customer`, and allows us to treat it differently to them. Here’s the code that validates a customer: -

#### **Listing 23.7 Creating a function to rate a customer**

```
let validateCustomer customer =
 match customer.PrimaryContactDetails with
 | Email e when e.EndsWith "SuperCorp.com" -> Some(GenuineCustomer customer) #A
 | Address _ | Telephone _ -> Some(GenuineCustomer customer) #B
 | Email _ -> None

let sendWelcomeEmail (GenuineCustomer customer) = #C
```

```

printfn "Hello, %A, and welcome to our site!" customer.CustomerId

#A Custom logic to validate a customer
#B Wrapping our validated customer as Genuine
#C The sendWelcomeEmail only accepts a GenuineCustomer as input

```

This function takes in a normal (unvalidated) `Customer`, and creates an *optional* `GenuineCustomer` as output. Then, we create our `sendWelcomeEmail` function which *only allows* a `GenuineCustomer` as input. This is the key point - it's now *impossible* for us to call this function with an unvalidated customer: -

```

let customer = createCustomer (CustomerId "C-1")
sendWelcomeEmail customer

```

This expression was expected to have type  
`GenuineCustomer`  
but here has type  
`Customer`

Figure 23.4 – Making an illegal state unrepresentable

The only way to call it is to create a `GenuineCustomer` customer, and to do *that* we need to go pass the checks in the `validateCustomer` function. You can imagine a customer being created and potentially validated (or not!) early on in the application, and then used later to send the email - safe in the knowledge that we can't "accidentally" call it with the wrong "type" of customer. The email function has no knowledge of how the customer was rated - it simply makes it a requirement *through the type system*.

We also now no longer have to write unit tests for whether the email code does the correct check on the rating of the customer (or anywhere else we need to split between these types of customers), because the type system protects us; we can *only* send emails for customers of a certain type. In effect, we've made *illegal states unrepresentable* through types of data.

### Breaking the Rules?

You might ask what's to stop you simply taking a normal customer, and then manually wrapping as a `GenuineCustomer`, thereby breaking our "rules"? Well, you *could* theoretically do this - just like many other parts of F#, this is there to *guide you to the pit of success* - but it won't *prevent you going out of your way to break things!* If you want to really be safe, you *can* create "signature" files (think "header files") which can restrict constructors of discriminated unions to a single file, thereby all but guaranteeing reliable construction of a type, but this is usually overkill.

### 23.2.1 When and when not to use marker types

Creating marker types can be incredibly powerful – you can use it for all sorts of things. For example, imagine being able to define email address that have been Verified or unverified for your users. Or how about distinguishing between the different states of an order in the type system e.g. unpaid / paid / dispatched / fulfilled etc. You could have functions that only act on fulfilled orders and not have to worry about accidentally calling it with an unpaid order! You can also use them at the “boundary” of your application - perform validation on “unchecked” data and convert them into “checked” versions of data, which provide you with security that you can never run certain code on “invalid” data.

My advice is to start simple: *do* use single case DUs as wrapper cases to prevent simple errors like mixing up customer and order ids – it’s cheap and easy to do, and is a massive help in eliminating some awful bugs that can crop up. Taking it further with marker types to represent states is a step up, and definitely worth persevering with – you can eliminate entire classes of bugs as well as eliminate swathes of boiler plate unit tests – but be careful not to take it *too* far, as it can become difficult to wade through a sea of types if overdone.

#### Quick Check

5. Why don’t we create wrapper types such as single case DUs in C#?
6. What benefit do we get from using single case discriminated unions as marker types?
7. When should we “wrap up” raw values into single case discriminated unions?

## 23.3 Results vs Exceptions

In F#, you can use exceptions just as you would in C#, using `try .. with` syntax. In the spirit of this book, I’m not showing you it because there’s nothing really interesting to see (although an example is included within the appendices). What is interesting to note is that exceptions are not encoded *within the type system*. For example, let’s imagine inserting a customer into a database. The signature might look like this: -

```
insertContact : contactDetails>ContactDetails -> CustomerId
```

In other words – given some contact details, save them to the DB and return their generated Customer ID. However, this function doesn’t cater for the possibility that the database might be offline; someone with those contact details might already exist etc. In fact, someone looking at this code would only know this if there was a try / catch handler somewhere in code – which might be an entirely different area of the code base. This can be thought of an *unsafe* function.

An alternative to using exceptions is to use a *Result*. This is simply a two-case discriminated union that holds either a *Success* or *Failure*. Here, if the call passes, you return a *Success* with the `CustomerId` generated by the database. If it fails, you’ll return the error text from SQL as a *Failure* case.

**Listing 23.8 Creating a result type to encode success / failure**

```

type Result<'a> = #A
| Success of 'a
| Failure of string

insertCustomer : contactDetails>ContactDetails -> Result<CustomerId> #B

match insertContact (Email "nicki@myemail.com") with #C
| Success customerId -> printfn "Saved with %A" customerId
| Failure error -> printfn "Unable to save: %s" error

```

#A Defining a simple Result discriminated union  
#B Type signature of a function which might fail  
#C Handling both success and failure cases up front

Now, this function clearly states that it *might not work* – and callers would have to test both Success and Failure cases in order to safely get to the customer. In fact, this pattern is now so common that F#4.1 will contain a `Result` type built into the standard library, just like `Option`. Internally in `insertCustomer`, we'd execute our code in a `try / catch`; any caught errors would be returned as a Failure.

Note – there's a fine line between when to use Exceptions and when to use Results; I would suggest that your rule of thumb is: if an error occurs and is something that you don't want to reason about e.g. catastrophic error that leads to end of the application – stick to exceptions. However, if it's something that you do want to reason about e.g. depending on success or failure, you want to do some custom logic and then resume processing in the application – a Result type is a useful tool to have.

**Quick Check**

8. What benefit does a Result give us over an Exception?
9. How should we convert code that throws exceptions into one that returns a Result?

**23.4 Summary**

Believe it or not, that was the final “language” lesson in this book! We: -

- Saw how to model business states in code
- Explored some domain modelling step-by-step with F#
- Looked at single-case discriminated unions

There's another Capstone exercise coming up next to allow you to get some more confidence with working with types and collections, but after this we'll start looking at the *applications* of F# in various guises.

### **Try This**

Look at some existing domain you've written in C#, and try to see where you might benefit from using Options and Single Case DUs in your model. Try to port the domain over to F# and see what impact it has!

### **Quick Check Answers**

3. Type safety – impossible to accidentally mix and match values of different types. Also semantic meaning of values through types.
4. Only unwrap values when you need to access the raw contents – not before.
5. It's generally too much boilerplate to create single wrapper classes in C#.
6. Marker types allow us to encode simple business rules directly within the type system.
7. Wrap up raw values into discriminated unions as early as possible.
8. Results allow us to clearly state in the type system whether a function can "fail".
9. Using a try / with block, converting the exception to a Failure.

# 24

## Capstone 4

**It's back to Bank Accounts again! This time we're going to apply the lessons we learned on domain modelling into the bank account system. We'll see: -**

- How to use Options in practical situations
- Using Discriminated Unions to accurately model a closed set of cases
- Working with Collections with more complex data
- Enforcing business rules through the type system

### 24.1 Defining the problem

When we completed Lesson 19, we had a version of a working bank account application that could handle persistence to disk and back again, as well as removing mutation for our command handler. Now we're going to remove some of the "code smells" that have been left lying around by introducing some lovely F# domain modelling. You'll have to: -

- Replace the unbounded command handlers with a fixed discriminated union
- Embed options for several situations where we need to cater for "might not have any data" rather than the arbitrary default values we've used so far.
- Look at how we might enforce business rules via some F# types to stop overdrawn customers withdrawing funds.

#### 24.1.1 Solution overview

As per the previous capstone, `src/lesson-24` contains a pre-built solution for you to use as the basis for this lesson, plus a `sample-solution` folder with a fully-working version for you to learn from. Take a moment to familiarise yourself with the basic solution so that you understand what it's doing – hopefully it's not too far off where you ended up in the previous Capstone.

## 24.2 Stronger typing with discriminated unions

One of the issues with the current version of code for the main program routine is that we're using code to enforce all our rules. What's wrong with that, you may ask – after all, isn't that what we're supposed to do! On the one hand, yes. But at the same time, we want to make use of the F# type system as much as possible to save us from any boilerplate errors.

### 24.2.1 Reviewing the existing command handler

One problem we have is with the fact that we use a simple `char` value to represent our command. Of course, we need to work with a `char` to capture the console input – but once we have that value, we continue to chuck it around the codebase. This poses a few smells immediately: -

#### **Listing 24.1 – The existing command execution pipeline**

```
commands
|> Seq.filter isValidCommand #A
|> Seq.takeWhile (not << isStopCommand) #B
|> Seq.map getAmount
|> Seq.fold processCommand openingAccount
```

#A Filtering out invalid characters from the stream

#B Testing whether the character represents the exit command – in this case, 'x'

Something I'm not fond of here is firstly that we can comment out the `Seq.filter` line and our code still compiles – it shouldn't. We want some form of "validation layer" that can stop us from accidentally missing things like this out! Ideally, we want a *bounded* set of "commands" that represent actions our application can do, and then go from the weakly typed, essentially *unbounded* `char` type into the command type at the earliest possible opportunity.

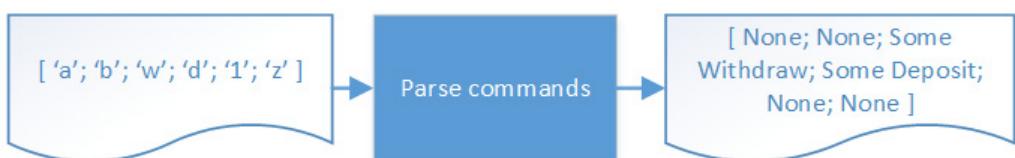


Figure 24.1 – Moving from a virtually unbounded type to a strictly bounded command type

Doing this means that within our code, we shouldn't ever be in the situation like we are in Listing 24.2: -

#### **Listing 24.2 – Working with unbounded values**

```
let processCommand account (command, amount) =
 if command = 'd' then account |> depositWithAudit amount
 else account |> withdrawWithAudit amount #A
```

## #A Using an “else” block as a “catch all” case

Spot the problem here? We’re “assuming” that if the `command` is not the character `d`, it must be `w` (for withdrawal). Even if we handled the withdrawal case explicitly, there’s still the “otherwise” case that needs to be handled. What do we do then? These sorts of smells can initially seem innocuous but can rapidly become the source of awful, awful bugs. Let’s stop this happening now, before it’s too late.

### 24.2.2 Adding a command handler with discriminated unions

Start within the `scratchpad.fsx` to test out these tasks, then move it back into `Program.fs` in the appropriate places.

1. The first thing we’ll want to do is create a simple discriminated union called `Command` to represent our three application commands: **Withdraw**, **Deposit** and **Exit**. These cases represent all of the activity our program can do.
2. Next, we need to write a simple function which can convert a `char` into a `Command`. I would normally use pattern matching here – you can match over the supplied character, and depending on the value, return the appropriate command.
3. Because we can’t be sure what value will be provided (i.e. it might not be `x`, `w`, or `d`), you’ll want to have this function return an `Option<Command>` to deal with the case that an invalid character is supplied; it’s common practice to prefix such functions with the word `try` e.g. `tryParseCommand` etc.
4. Make sure you’re happy with that function – test it out in the REPL – and then port it into the application.

Now we’re ready to “hook” our new `Command` domain into our code – essentially, in the pipeline around line 50.

5. Replace the call that filters out invalid commands to one which maps from simple characters into our `Command`. However, since our `tryParse` function will be returning option values, it’s better to use `choose` rather than `map` here (Lesson 22 covers `choose` – refer there if needed).
6. You’ll notice that the next line in the pipeline now immediately doesn’t work – that’s because it checks against the `char 'x'` – update it to compare against the `Exit` command instead (indeed, you might want to simply put the lambda inline now as the code is obvious to read).
7. Finally, you’ll need to update `processCommand`. This should now pattern match over the command that is supplied; depending on whether it’s **Deposit** or **Withdraw**, it should call the appropriate function. If it’s **Exit**, it should simply just return the account back out again.

As an extra exercise, you can also apply the same technique to the loading of data from the file system in `Operations.loadAccount`, so that instead of an `if / then` expression against the

text "withdraw", you can try to parse the text to a Bank Operation first and process that instead.

### 24.2.3 Tightening the model further

We now have a cleaner domain – it's easier to reason about, since we know that there are only three possible commands, and we don't have to guess what the different characters mean – the DU cases are self-explanatory. Unfortunately, one small smell remaining is that the match expression in `processCommand` has to cater for the Exit command as well, even though we know that it *should* never really be possible.

```
match command with
| Deposit -> Incomplete pattern matches on this expression.
| Withdraw -> For example, the value 'Exit' may indicate a case not covered
```

Figure 24.2 – Pattern Matching warns us that we have not handled all the cases for our DU.

This is a good example of F# “giving us a hint” that the current way that we've modelled our domain is not *quite* right. Let's fix this by enhancing our domain model so that we clearly show that there's a difference between the Exit command and the two bank operations.

### Listing 24.3 – Creating a two-level discriminated union

```
type BankOperation = Deposit | Withdraw #A
type Command = BankCommand of BankOperation | Exit #B
let tryGetBankOperation cmd = #C
 match cmd with
 | BankCommand op -> Some op
 | Exit -> None

#A Defining the types of bank operations
#B Defining the types of commands, one of which is a bank operation
#C Filtering out non-bank operation commands
```

What we've done now is create a “sub-group” DU – in other words, a Command is either: -

- Exit, or
- A Bank Operation (either Deposit or Withdraw)

If you follow this approach (and update the parsing code), you can use `tryGetBankOperation` in the pipeline to convert from a `Command` to a `BankOperation`; by the time you get to the `processCommand` function, you'll now be able to match against `BankOperation`, which only has the two cases – exactly what we wanted.

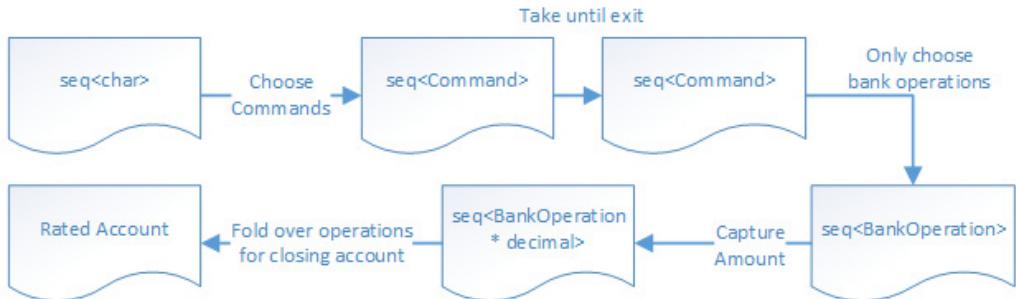


Figure 24.3 – Visualising the updated pipeline

Try testing this pipeline out in a script with a set of sample chars, like we did in the previous Capstone!

## 24.3 Applying Option types with the outside world

Let's now look at a few situations where it's useful to apply the `Option` type when dealing with external data that we're not necessarily in control of.

### 24.3.1 Parsing user input

If the user accidentally types a non-number when entering the amount to withdraw or deposit, the application crashes. Let's stop that by updating the `getAmount` function: -

#### **Listing 24.4 – Safely parsing user input**

```

let tryGetAmount command =
 Console.WriteLine()
 Console.Write "Enter Amount: "
 let amount = Console.ReadLine() |> Decimal.TryParse #A
 match amount with
 | true, amount -> Some(command, amount) #B
 | false, _ -> None #C

```

#A Safely parsing the input from the user  
#B Only return some output if the input is a valid decimal  
#C Invalid input results in No output

As you can see, F# plays very nicely with `TryParse` methods in the BCL by returning both the *boolean result* of the parse operation and the *parsed value itself* – so you can easily pattern match over it and deal with it as you see fit.

Once again, we've renamed this function to start with `try`, as it now returns an `option<decimal>` – which means that once again we'll need to update the pipeline to use `choose` rather than `map`, to skip out invalid answers.

### 24.3.2 Loading existing accounts

In the current implementation of the application, when the program starts up and you've entered your name, the file repository attempts to locate your bank account on disk. If the file repository can't locate the account, it creates a "default" account and returns that instead. This is nice in a "you don't need to know" kind of way, but it's a bit of a mismatch of responsibilities: the file repository module should simply retrieve the bank account from disk; if it can't be located, it should be up to the caller to decide what to do next. In other words, the call to `findTransactionsOnDisk` should return an option, rather than a concrete result.

If you dig into the code a bit, you'll find the `findAccountFolder` function, which is the root of our problem.

#### **Listing 24.5 – Unintentionally hiding optionality with a default value**

```
let private findAccountFolder owner =
 // code elided...
 if Seq.isEmpty folders then "" #A
 else
 let folder = Seq.head folders
 DirectoryInfo(folder).Name

let findTransactionsOnDisk owner =
 let folder = findAccountFolder owner
 if String.IsNullOrEmpty folder then ... #B
 else loadTransactions folder #C
```

#A "Missing" value represented by an empty string  
#B Checking for the empty string and returning an empty account  
#C Performing the "real" logic to load transactions from disk

Here, we used an empty string to "simulate" the absence-of-a-value – in other words, this is no better than using some magic number of -1 to indicate a missing number. And indeed, the calling `findTransactionsOnDisk` function has a check for an empty string which is used to indicate that no account exists on disk, and so we create a "default" account. Let's fix all this, starting from the ground up: -

1. Change `findAccountFolder` to `tryFindAccountFolder`. In other words, have it return `None` if no folder was found, or `Some folder name` if it was.
2. For bonus points, convert the sequence of directories to a `List` and pattern match over it instead of the `if / else` expression.
3. Now fix up `findTransactionsOnDisk`, which will be failing, by removing the empty string check and replacing it with a pattern match against the result of `tryFindAccountFolder`.
4. Next, change this function to return an option output: -
  - o If no account folder was found, return `None`
  - o If an account folder was found, wrap the results of `loadTransactions` in `Some`

### 24.3.3 Lifting Functions to support Options

Almost there! We've now fixed the low-level File Repository – it now only returns account details from disk if they exist - but now `Program.fs` is broken where we "plug together" two functions: -

```
^TransactionsOnDisk >> Operations.loadAccount
```

Type mismatch. Expecting a  
`(string * Guid * seq<Transaction>) option -> 'a`  
but given a  
`string * Guid * 'b -> Account`  
The type '(string \* Guid \* seq<Transaction>) option'  
does not match the type 'string \* Guid \* 'a'

Figure 24.4 – Incompatible functions cannot be composed together

Don't be scared by this error! Start by looking at the signatures of the two functions we're trying to compose together: -

```
tryFindTransactionsOnDisk: string -> (string * Guid * seq<Transactions>) option
loadAccount: (string * Guid * seq<Transactions>) -> Account
```

The output of `tryFindTransactionsOnDisk` is now an *optional* tuple. The input of `loadAccount` only accepts a tuple – it doesn't want an optional tuple. Remember, when composing two functions, the type of output of the first must match the input type of the second.

Now, the crude solution to this would be to "pollute" `loadAccount` to take in an optional tuple. Don't do this! A better approach is to create a new function that "lifts" the existing function to handle the "optionality": -

#### **Listing 24.6 – Manually lifting a function to work with an optional input**

```
let loadAccountOptional value =
 match value with
 | Some value -> Some(Operations.loadAccount value) #A
 | None -> None
FileRepository.tryFindTransactionsOnDisk >> loadAccountOptional #B
```

#A Wrapping the existing `loadAccount` function with optionality

#B Using the newly lifted function in place of the original incompatible one

However, there's a much, much quicker way to do this – *anywhere* you see this pattern of a match over an option where the `None` branch *also returns None*, you can replace the whole thing with either `Option.map` or `Option.bind` (the former if you had to manually "wrap" the result in the `Some` branch - as we have done here - and the latter if the result from the "lifted" function was already an `Option`)

**Listing 24.7 – Lifting a function to support options using Option.map**

```
let loadAccountOptional = Option.map Operations.loadAccount #A
FileRepository.tryFindTransactionsOnDisk >> loadAccountOptional #B

#A Lifting a function to support optionality using Option.map
#B Composing the newly-lifted function
```

This will now leave us with the composed `tryLoadAccountFromDisk` with the signature of `string -> Account option`. In other words it may return an account, but only if one existed on disk – perfect!

Having done this, the last thing to do is to (finally) leave the world of options and create a default account for a new user – the best place to do this is right at the top level, after capturing the user's name and trying to load their account from disk: -

**Listing 24.8 – Creating a default account in the appropriate location**

```
let openingAccount =
 Console.Write "Please enter your name: "
 let owner = Console.ReadLine()

 match (tryLoadAccountFromDisk owner) with #A
 | Some account -> account #B
 | None ->
 { Balance = 0M
 AccountId = Guid.NewGuid()
 Owner = { Name = owner } } #C
```

#A Trying to load the owner's account from disk  
#B Returning the account if it was found  
#C Creating a new account with a new Account Id if no account was found on disk

## 24.4 Implementing business rules with types

The last section will touch on how we can enforce a slightly higher-level business rule through the type system:

*A user can go into an overdrawn state i.e. draw out more funds than they have in their account. However, once they have become overdrawn, they cannot draw out any more funds. They can still deposit funds into the account, and once their balance returns into a non-negative state, they will be able to withdraw funds once again.*

This is something that we *could* achieve with some pattern matching during the withdraw process i.e. replace the current check in `withdraw` so that it checks whether the account is *already* overdrawn rather than the current behaviour of preventing users going overdrawn at all. However, let's try an alternate way: we'll enforce this behaviour through *types*. Firstly, let's formalise the business rule above: -

- An account can be in one of two states: *overdrawn* or *in credit*

- Only an account that is *in credit* can withdraw funds
- Any account can deposit funds

Now let's model that in F# by enhancing our domain model a little: -

#### **Listing 24.9 – Modelling a rated account**

```
type CreditAccount = CreditAccount of Account #A
type RatedAccount = #B
| Credit of CreditAccount
| Overdrawn of Account

#A Marker type for an account in credit
#B Categorisation of account
```

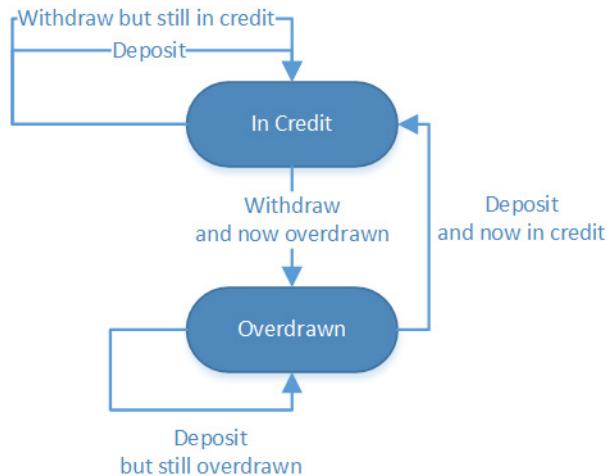


Figure 24.5 – Visualising the state transitions for our business rule

#### **24.4.1 Testing a model with scripts**

We've split an account in two paths, and have created a marker, or wrapper, type (`CreditAccount`) to indicate that a specific account is in credit. Now, we can update our core account operation functions: -

#### **Listing 24.10 – “Safe” operations on a bank account**

```
let rateAccount account = #A
 if account.Balance < 0M then Overdrawn account
 else Credit(CreditAccount account)

let withdraw amount (CreditAccount account) = #B
 { account with Balance = account.Balance - amount }
 |> rateAccount
```

```

let deposit amount account = #C
 let account =
 match account with
 | Credit (CreditAccount account) -> account
 | Overdrawn account -> account
 { account with Balance = account.Balance + amount }
|> rateAccount

#A Code to rate a "naked" account
#B Withdraw only works with credit accounts
#C Deposit works with both credit and overdrawn accounts

```

I'd advise you to explore these functions in isolation within a script to get a "feel" for how they work before incorporating into the code directly. In a nutshell: -

- `rateAccount` is a helper function. It takes in a "standard" account and categorises it for us based on the balance.
- `withdraw` now only takes in accounts that are *in credit*. It's not possible to call this function with an overdrawn account.
- `deposit` accepts both in credit and overdrawn accounts.

Both `withdraw` and `deposit` return a *rated account* back out – in other words, after performing the transaction, they check the current balance of the account – if overdrawn, the account is categorized as `Overdrawn`, and the whole process begins again. This single function, `rateAccount`, is the *only* place that is allowed to create a `RatedAccount`.

```

myAccount
|> deposit 50M
|> deposit 100M
|> withdraw 500M

```

Type mismatch. Expecting a  
`RatedAccount -> 'a`  
but given a  
`CreditAccount -> RatedAccount`  
The type 'RatedAccount' does not match the type 'CreditAccount'

Figure 24.6 – Compiler support for enforcing business rules

You can test this out easily within a script. As you see from Figure 24.6, the compiler blocks us from calling `withdraw` directly on an account which *might* be overdrawn. You can get around this by writing a simple wrapper function to assist here that safely tries to withdraw funds and can be used in place of `withdraw` in the pipeline from Figure 24.6: -

**Listing 24.11 – A safe withdrawal wrapper**

```
let withdrawSafe amount ratedAccount =
 match ratedAccount with
 | Credit account -> account |> withdraw amount
 | Overdrawn _ ->
 printfn "Your account is overdrawn - withdrawal rejected!"
 ratedAccount // return input back out
```

The point here is that we've now provided a "barrier" around our internal domain; the withdrawal operation doesn't need to perform any validation or checks on balances before carrying it out. Instead, this guarantee is performed by the compiler *for us*.

### 24.4.2 Plugging our new model back in

As an exercise, you should now try to plug this code back into the main application. You'll have to watch for several things: -

- The `loadAccount` and `processCommand` functions will need to be updated to work with the new model; both will need to explicitly handle the situation where an attempt is made to withdraw funds from an overdrawn account (rather than blindly passing it through to the underlying code). The nice thing with this approach, particularly with `processProcess`, is that it's now very easy to reason about whether an account is overdrawn or not – we can simply pattern match over the account (an alternative would be to use an *Active Pattern* – something we've not looked at, but you could in your own time)
- The `auditAs` function will need to be updated so that it works for both types of accounts. You'll need to refactor it quite carefully so that it works whilst retaining the ability to work with two types of accounts; take a look at the sample solution if you get stuck. The "`withAudit`" composed functions at the top of the Program will also need to be updated accordingly.
- You can probably completely remove the `Accepted` property from the `Transaction` record – it's now impossible to attempt a transaction which is rejected from our internal domain, so it doesn't make sense to keep it any more (this will also mean updating the serialization / deserialization code).

With a little bit of work, you should end up with something that looks similar to this: -

```
C:\Users\Isaac\Source\Repos\learnfsharp\src\code-listings\lesson-24\samp
Please enter your name: Isaac
Opening balance is £160
(d)eposit, (w)ithdraw or e(x)it: d
Enter Amount: 10

Account Isaac: deposit of 10
Current balance is £170
(d)eposit, (w)ithdraw or e(x)it: w
Enter Amount: 200

Account Isaac: withdraw of 200
Current balance is £-30
Your account is overdrawn!!
(d)edeposit, (w)ithdraw or e(x)it: w
Enter Amount: 10

You cannot withdraw funds as your account is overdrawn!
Current balance is £-30
Your account is overdrawn!!
(d)edeposit, (w)ithdraw or e(x)it:
```

Figure 24.7 – Testing out bank account app using a stronger domain model

The difference to what we had at the start is that the validation layer has been forced “up” above the core domain, and we’ve now gained extra safety to prevent accidentally calling a function (`withdraw`) when we shouldn’t do (when the account is overdrawn).

## 24.5 Summary

That’s the end of this Capstone. It’s probably the hardest one you’ll encounter in the entire book, as it focused on concepts that you’ve probably not touched on much in the past, such as using discriminated unions and a rich domain model to enforce business rules – so don’t feel too disheartened if you struggled a little!

# Unit 6

## *Living on the .NET platform*

We've now finished with the language side of the book! The rest of the units will cover areas relating to using those language features in a variety of scenarios of areas, from data access to web programming. This means that we'll be focusing on using the lessons you've learned so far on F# to perform similar tasks that you're doing today more quickly - and correctly - than you're used to. But don't think that this means there's nothing to learn – far from it!

The first area we'll cover is interoperating with the rest of .NET – which is dominated by C# codebases. You've already seen us work with the BCL throughout this book, but this unit will focus on looking at larger issues, such as how to design applications that work well in a multi-language solution, as well as under what circumstances to rely on C# or F#. We'll also cover how to use NuGet packages in F# as easily as possible.

# 25

## *Consuming C# from F#*

**Throughout this book, you've been working with C# types and objects from F#, and so you know how to deal with the basics already. However, there are a few extra areas that are worth touching upon in order to round off this area. We'll look at: -**

- Creating hybrid solutions
- Making use of Visual Studio tools
- Consuming assemblies from scripts
- Consuming classes and interfaces in F#
- Safely working with nullable objects in F#

### 25.1 Referencing C# code in F#

In addition to working with the BCL codebase, virtually any C# code you write today can be consumed from F#. This includes: -

- All BCL code
- .NET assemblies including NuGet packages (see lesson 26)
- Sibling projects in the same solution

In other words, virtually exactly the same things you do today in C#. So, with that bold statement in mind, let's try to create a solution that uses both C# and F# code right now!

---

#### **Dynamic F#**

One area that C# has richer support for compared to F# is dynamic typing. There *is* support for some dynamic typing through the custom ? operator in F# which allows you to handle dynamic member access in a similar vein to C#'s dynamic object types, but this feature is generally rarely used. F# does have some meta-programming features such as type providers, which we'll see in the next unit, which somewhat alleviates the need for dynamic typing.

### 25.1.1 Creating a hybrid solution

#### Now You Try

Let's start with a simple solution that has some C# utility code which we'll call from an F# project. We'll then expand upon this throughout the rest of this lesson. Open up Visual Studio, and create a new F# Console Application, called FSharpProject.

1. Add a new C# project called CSharpProject to the solution.
2. Open up Visual Studio, and create a new F# Console Application, called FSharpProject.
3. Add a new C# project called CSharpProject to the solution.
4. Add a reference to the CSharpProject from the FSharpProject using the standard Add Reference dialog.

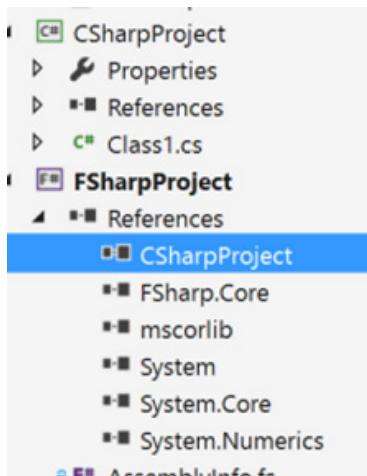


Figure 25.1 – A hybrid C# / F# solution.

That's it – we can now start to write code in C# and use it in F#! This sort of thing was the *original aim* of the .NET and CLR teams – to have the ability to mix and match languages across a common runtime, using each language where it fit best in terms of the problem domain at hand. Somewhat disappointingly, somewhere between 1999 and today, this message was more or less replaced with simply “use C# everywhere”.

#### Open the Class1.cs file and replace the contents of the class as follows: -Listing 25.1 A simple C# class

```
public string Name { get; private set; } #A
public Person(string name) { #B
 Name = name; }
public void PrintName() { #C
 Console.WriteLine($"My name is {Name}"); }
```

```
#A Public read-only property
#B Constructor
#C Public method
```

5. Build the CSharpProject.
6. Build the CSharpProject.

**Now let's look at consuming this code from F#. First, replace the contents of the Program.fs file with this: -Listing 25.2 Consuming C# code from F#**

```
[<EntryPoint>]
let main argv =
 let tony = CSharpProject.Person "Tony" #A
 tony.PrintName() #B
 0
```

```
#A Calling the Person constructor
#B Calling the PrintName method
```

7. Run the application.

### Quick Check

1. Can you share F# and C# projects in the same solution?
2. Name some types of assets that you can reference from an F# solution.
3. What kind of type is not well supported in F#?

### Answers

1. Yes.
2. Assemblies, sibling projects and BCL code.
3. Dynamic types

## 25.2 The Visual Studio experience

Working with multi-language solutions in Visual Studio is generally a pain-free experience – although not always. Let's look at some common tasks you'll be familiar with, and how they work with hybrid-language solutions: -

### 25.2.1 Debugging

#### Now You Try

First of all, you can debug an application that works across both languages, no problem. You can set breakpoints in Visual Studio, and even see cross-language call stacks. Let's try it out.

1. Set a breakpoint inside the definition C# Person class constructor.

- Run the F# console application. Observe that the debugger hits with the name value set to "Tony".

```
public Person(string name)
{
 Name = name;
}
```

The screenshot shows a debugger interface. A line of code 'Name = name;' is highlighted in yellow. Below it, a tooltip displays the variable 'name' with its value set to 'Tony'. The tooltip includes a search icon and a dropdown arrow.

Figure 25.2 – Debugging a C# function called from F#

- Also, observe the call stack is correctly preserved across the two languages.

| Call Stack                                                                    |  |  |          |
|-------------------------------------------------------------------------------|--|--|----------|
| Name                                                                          |  |  | Language |
| ● CSharpProject.dll!CSharpProject.Person.Person(string name = "Tony") Line 15 |  |  | C#       |
| HybridSolution.exe!Program.main(string[] argv = {string[0]}) Line 3           |  |  | F#       |

Figure 25.3 – A cross-language call stack with both F# and C# stack frames

### 25.2.2 Navigating across projects

We're used to many of the standard Visual Studio refactoring and navigation features. The Visual F# Power Tools adds many of these to F#, including Rename, Find All References and Go to Definition. Unfortunately, these *don't work across languages* (my understanding is that this is not an "F#" thing – it's the same across VB .NET and C#, too). For example, Find All References will only trap references in projects of the language that you're currently in; similarly, if you attempt to use the Go to Definition navigation feature on the C# Person constructor from F#, you'll instead be taken to the F# *metadata* view of the C# class:

```
namespace CSharpProject

type Person =
 new : name:string -> Person
 member Name : string with get, set
 member PrintName : unit -> unit
```

Figure 25.4 – F# Metadata view of a C# type

This is unfortunate, but (at least in my experience), it's not a massive show stopper – but it *is* something that you should be aware of.

### 25.2.3 Projects and Assemblies

You'll notice that I stressed during the previous exercise to first *build* the C# project *before* trying to access it from the F# project. That's because Visual Studio doesn't automatically cascade code changes as they are made across languages – you have to first build the *dependent* assembly before you can see the changes in the client project. In other words, if you make a change to some code in the C# project, you'll need to explicitly build that project before you can "see" the changes in F#.

### 25.2.4 Referencing assemblies in scripts

Just like a project file has project and assembly references, so can scripts. The difference is that a project file contains the references embedded within the `.csproj` or `.fsproj` file – there's no such notion as a "project file" for scripts, as they are self-standing. So instead, F# provides us with a few useful "commands" in F# scripts: -

**Table 25.1 F# script commands**

| Command            | Description                                                         | Example Usage                            |
|--------------------|---------------------------------------------------------------------|------------------------------------------|
| <code>#r</code>    | References a dll for use within a script                            | <code>#r @"C:\source\app.dll"</code>     |
| <code>#I</code>    | Add a path to the <code>#r</code> search path                       | <code>#I @"C:\source\"</code>            |
| <code>#load</code> | Loads and executes an F# <code>.fsx</code> or <code>.fs</code> file | <code>#load @"C:\source\code.fsx"</code> |

Using these directives opens up all sorts of interesting possibilities with scripts when working with third-party code, as you can quickly reference external code and experiment with them in a scratchpad environment – we'll see this in more detail later on in this unit.

#### Now you Try

Let's experiment with referencing an assembly within a script.

1. Create a new script as a new solution item called `Scratchpad.fsx`.
2. Open the script file and enter the following code.

#### Listing 25.3 Consuming C# assemblies from an F# script

```
#r @"CSharpProject\bin\debug\CSharpProject.dll" #A

open CSharpProject #B
let simon = Person "Simon"
simon.PrintName()

#A Referencing the CSharpProject from an F# script. Relative references work relative to the
script location.
#B Standard F# code to utilise the newly-referenced types
```

3. Execute the code in the script using the standard Send to F# Interactive behavior. Notice the first line that is output in FSI: -

```
--> Referenced 'C:\[path elided]\CSharpProject\bin\debug\CSharpProject.dll' (file may
be locked by F# Interactive process)
```

We use the @ to treat backslashes as literals. Note that if your script lives in a different location, the #r line might not work – if that's the case, simply navigate to where it is in Windows Explorer in order to identify what the correct path should be.

### 25.2.5 Debugging scripts

Visual Studio also allows you to *debug* F# scripts! I've purposely steered away from this because debugging can, in my opinion, be a costly way to identify issues as opposed to designing small, simple functions with minimal dependencies – but sometimes it's necessary, particularly if you're using your script as a harness with which to test e.g. C# code.

#### **Now You Try**

Let's debug the script that we've already got open to see how it operates in VS2015.

1. With the script from Listing 25.3 still open, right-click line 3 (the constructor call line) with your mouse.
2. From the pop-up menu, choose "Debug with F# Interactive".
3. After a short delay, you'll see the line highlighted. From there, you can choose the regular "Step Into" code as normal.

```
open CSharpProject
let simon = Person "Simon"
simon.PrintName() ≤ 3,532ms elapsed
```

Figure 25.5 – Debugging an F# script in Visual Studio 2015

4. You can also do the same using the keyboard shortcut CTRL + ALT + ENTER.
5. When you've finished debugging, you can click Stop from the toolbar, or hit SHIFT + F5 to stop the debugging session.

#### **Quick Check**

4. Can you debug across languages?
5. Can you go to definition across languages?
6. How do you reference a library from within a script?

#### **Answers**

4. Yes. This works out of the box in Visual Studio 2015.

5. Partially. Visual Studio will show you metadata for the defined type in the language you've just come from.
6. Using the #r directive.

## 25.3 Working with OO constructs

Let's move away from looking at Visual Studio tooling features now, and back to some language-level concerns, by seeing how F# *improves* on standard C# object-oriented constructs such as constructors and interfaces.

Notice how in Listing 25.3 that that we don't need to bother with the `new` keyword when calling constructors (or supply brackets for single-argument constructors). That's because F# considers a constructor to simply be a function that when called, returns an instance of the type (in this case, `Person`) – so you can actually use constructors in the same way as *any function*. For example, let's say we wanted to create five `Person` objects using a list of names. Here's two ways you can do this: –

### **Listing 25.4 Treating constructors as functions**

```
open CSharpProject

let longhand =
 ["Tony"; "Fred"; "Samantha"; "Brad"; "Sophie "]
 |> List.map(fun name -> Person(name)) #A

let shorthand =
 ["Tony"; "Fred"; "Samantha"; "Brad"; "Sophie "]
 |> List.map Person #B

#A Calling a constructor explicitly
#B Treating a constructor like a standard function
```

The first version would have been typical code in F#3, but since F#4 we can use the shorthand second version – in other words, the `Person` constructor is a function taking in a string, which is the single argument used in `List.map` here – so we can omit the argument entirely.

### 25.3.1 Working with Interfaces

Just like classes, interfaces are a fact of life in the .NET framework – and indeed, they can sometimes be useful in F# as well, particularly when working with pluggable pieces of code that need to change at runtime. As such, F# has good support for implementing them, both at the language and at the tooling level.

Let's try and create a simple instance of a standard BCL interface in F# – the `System.Collections.Generic.IComparer` interface. This interface simply allows us to tell whether one object is “greater” than, less than, or equal to another object of the same type. Let's see how to create, and consume, an instance this interface in F#: –

### Listing 25.5 Treating constructors as functions

```
open System.Collections.Generic

type PersonComparer() = #A
 interface IComparer<Person> with #B
 member this.Compare(x, y) = x.Name.CompareTo(y.Name) #C

let pComparer = PersonComparer() :> IComparer<Person> #D
pComparer.Compare("simon", Person "Fred")

#A Class definition with default constructor
#B Interface header
#C Implementation of interface
#D Creating an instance of the interface
```

The first few lines of this snippet are reasonably self-explanatory – although one point of interest is that the `Compare` function does not need any type annotations for `first` and `second` – these are inferred by F# by the generic type argument `<Person>`. What is important to note however is that we have to explicitly upcast from our `PersonComparer` type to `IComparer<Person>` in Line 8 using the `:>` operator. This is because F# implements interfaces *explicitly* – so without this cast, you would not see the `Compare` method to call in Line 9. You can also define explicit interface implementations in C#, but it's generally not used.

#### **Now you Try**

F# Power Tools comes with a handy refactoring to implement an interface for you.

1. Enter the first three lines from Listing 25.5.
2. Remove the “with” keyword from the third line.
3. Move the caret to the start of the `IComparer` in the same line.
4. You will be presented with a smart tag. Hit `CTRL + .` to open it.
5. Try out both forms of generation – the first (non-lightweight) will generate fully-annotated type signatures; the latter will omit type annotations if possible and place method declarations with stub implementations on a single line.

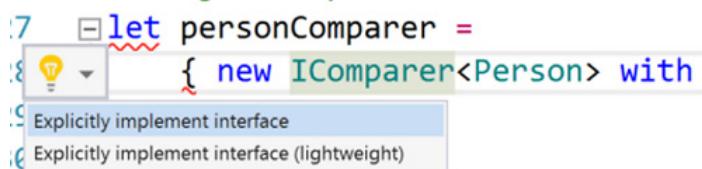


Figure 25.6 – The VFPT Implement Interface refactoring.

### 25.3.2 Object expressions

F# has another trick up its sleeves for working with interfaces, called *object expressions*. Object Expressions let you create an *instance of an interface* without creating an intermediary type. Sounds impossible, right? Here's what it looks like: -

#### **Listing 25.6 Using object expressions to create an instance of an interface**

```
let pComparer =
 { new IComparer<Person> with #A
 member this.Compare(x, y) = x.Name.CompareTo(y.Name) } #B

#A Interface definition
#B Interface implementation
```

The type of `pComparer` here is `IComparer<Person>` - its “real” name is generated by the compiler, and you can never see this (unless using reflection). Using object expressions allows us to skip around the need to manually construct a type to hold the implementation – we can simply create the implementation of the interface as an *object* in one step!

### 25.3.3 Nulls, Nullables and Options

We saw in Lesson 19 how to work with options. The problem is that C# classes and structs don't work natively with Options. So how do we marry these two worlds up? Well, since F# 4 we also have a few handy combinators in the `Option` module that allows us to easily jump between F# options and classes, which are always potentially null, and structs that are wrapped as Nullables.

#### **Listing 25.7 Option combinators for classes and nullable types**

```
open System

let blank:string = null #A
let name = "Vera"
let number = Nullable 10

let blankAsOption = blank |> Option.ofObj
let nameAsOption = name |> Option.ofObj
let numberAsOption = number |> Option.ofNullable

let unsafeName = Some "Fred" |> Option.toObj

#A Creating a selection of null and non-null strings and value types
#B Null maps to None
#C Non-null maps to Some
#D Options can be mapped back to classes or Nullable types
```

In this way, you can safely map from other applications or libraries (which return data that may or may be null) into a “safe” F# domain that allows us to more easily reason about

nullability of data. Then, when leaving F# and going back to the C# or VB .NET world, we can use the `Option.toObj` to go back to the “unsafe” world of potentially null classes.

### **Quick Check**

7. What is an object expression?
8. How do you convert between a Nullable and an Option in F#?

## **25.4 Summary**

Hopefully that was a relatively gentle introduction into the second half of this book! We saw: -

- How to build multi-language .NET solutions
- Looked at Visual Studio tooling support
- Learned some tricks for working with scripts
- Saw how F# language features allow us to easily work with C# constructs.

### **Try This**

Take any existing C# / VB .NET solution that you have, and add an F# class library project to it and reference your C# library (alternatively, create an F# script and reference the C# DLL using the `#r` directive). Try to create some of your C# types from F#. If your application has some form of Console runner, try to re-write it as an F# console application. Explore “driving” your existing OO code from F#.

### **Quick Check Answers**

1. Yes.
2. Assemblies, sibling projects and BCL code.
3. Dynamic types
4. Yes. This works out of the box in Visual Studio 2015.
5. Partially. Visual Studio will show you metadata for the defined type in the language you’ve just come from.
6. Using the `#r` directive.
7. An F# language feature which allows you to create instances of interfaces without a formal implementation type.
8. Using the `Option.ofNullable` and `Option.toNullable` helper functions.

# 26

## *Working with NuGet Packages*

In the last lesson, we spent time looking at working with non-F# projects from F#, as well as learning a few tricks on working with scripts. In this lesson, we'll move on to looking at how to work with **dependencies** within scripts and projects, rather than simply writing standalone scripts. We'll see: -

- How to work with NuGet packages in F#
- Tips and tricks when working with scripts
- What the Paket dependency manager is

### 26.1 Using NuGet with F#

#### 26.1.1 Working with NuGet with F# Projects

The good news is that NuGet packages work out of the box with F# projects in Visual Studio – there's no difference to working with C#! Let's first see how it works with F#.

##### **Now you try**

Let's download a NuGet package from the main public NuGet servers and use it within an F# Class Library.

1. Create a new F# Class Library in Visual Studio called `NugetFSharp`.
2. Using the standard "Manage NuGet Packages" dialog, add the `Newtonsoft.Json` package to the project.
3. You'll see the the package is downloaded and added as a reference, just like in your C# project - you can now use it directly from your F# source files.
4. Change the contents of the `Library1.fs` file to the following code: -

**Listing 26.1 Using Newtonsoft.Json in F#**

```
module Library1

open Newtonsoft.Json
type Person = { Name : string; Age : int } #A

let getPerson() =
 let text = """{ "Name" : "Sam", "Age" : 18 }"""" #B
 let person = JsonConvert.DeserializeObject<Person>(text) #C
 printfn "Name is %s with age %d." person.Name person.Age
 person

#A Defining an F# record
#B Some sample JSON text that matches our record structure
#C Using Newtonsoft.Json to deserialize the object
```

Some interesting points to note here: -

- We use triple-quoted strings here to allow us to use single quotes within the string without the need to prefix with a backslash.
- Newtonsoft.Json works out of the box with F# record types! It'll automatically map JSON fields to F# Record fields, just like with C# class properties.

The main take-away from this, though, is that you can use virtually any NuGet package with F#. This means that you can benefit from all of the existing libraries out there, whilst still making use of the stronger type system, more succinct syntax and powerful compiler of F#.

## 26.2 Experimenting with scripts

One of the nicer things that you can do with the F# REPL is to use it in conjunction with NuGet packages to quickly and easily explore and test out a new NuGet package, or to put together a quick proof of concept that you're working on. Let's see what I'm talking about.

Let's imagine that you need to test out a new NuGet package your team have decided to start using on an existing project. You'll probably look on the website of the package (if there is one), have a skim through the "getting started" section of the documentation (again, if there is one), and then download the package to your solution. Finally, you'll try to embed this package within the context of your application. This is a common approach, but usually one fraught with problems. For example, it might be difficult to test it out within the context of your application – what if that code is only called in very specific circumstances that are difficult to reproduce?

Alternatively, perhaps you'll use unit tests or a console application to "prove" how it works – but we've already discussed how those are poor ways to experiment and explore. It's much more productive to use a script to test out how a package works – *in isolation* – so that you can learn how to use it properly; only then, once you're confident with how it works, do you add it to your codebase.

## Now you Try

Let's now add another NuGet package to our project, and work with both this and Newtonsoft.Json from within a script.

1. Add the **Humanizer** NuGet package to the project. This package can take arbitrary strings and tries to make them more human-readable.
2. Open up the `Script1.fsx` that was already added to the project on creation.
3. We now need to reference the Humanizer assembly in our script using the `#r` directive we saw in Lesson 21. The simplest option is to open the references node in Solution Explorer, get properties of the Humanizer dll and then copy the entire path into the clipboard, and then enter the following code in Listing 26.2.

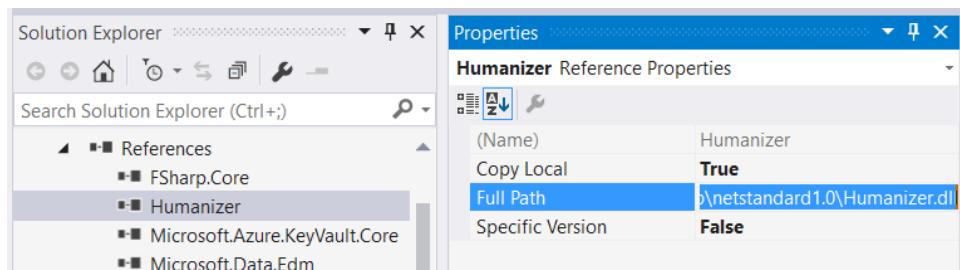


Figure 26.1 – Determining the full path of an assembly from a NuGet Package.

### Listing 26.2 Referencing an assembly from a NuGet package

```
#r @"<path to Humanizer.dll>" #A
open Humanizer
"ScriptsAreAGreatWayToExplorePackages".Humanize()
```

#### #A Referencing an assembly using #r

4. Execute the code; the output in FSI should be: "Scripts are a great way to explore packages"
5. Explore the overloads of the `Humanize` method in the REPL e.g. one takes in a `LetterCasing` argument.

The main point to see here is that we've used a script to quickly get access to a NuGet package that we've downloaded, and started to explore how it works in an isolated and safe environment. It's much quicker to do this in a script, as we get immediate feedback in the REPL for what's going on. Plus, because we're working in a script file, we can save the script and use it again later – or use it as a form of documentation to other developers to show them how to use a dependency correctly!

### 26.2.1 Loading source files in scripts

Let's now use our code we wrote earlier in Listing 26.1 within a script.

#### Listing 26.3 Loading a source file into a script with a NuGet dependency

```
#r @"<path to Newtonsoft.Json.dll>" #A
#load "Library1.fs" #B
Library1.getPerson() #C

#A Referencing the Newtonsoft.JSON assembly
#B Loading the Sample.fs source file into the REPL.
#C Executing code from the Sample module.
```

Here, we've loaded the `Library1.fs` source file and called a function exposed by it. An important point to note here is that we had to explicitly reference the `Newtonsoft.Json` assembly before loading the script – this is required as `Library1.fs` uses the `Newtonsoft` namespace. If you comment out the line that references `Newtonsoft.Json`, you'll see an error like this: -

`#load "Library1.fs"`

One or more errors in loaded file.  
The namespace or module 'Newtonsoft' is not defined

Figure 26.2 – Trying to load a source file without having referenced a required dependency

Also be aware that if you `#load` a `.fs` file into a script, any other `.fs` files that it depends on will need to be loaded first!

### 26.2.2 Improving the referencing experience

There's a couple of problems with the way we're referencing assemblies here. Firstly, we're copying the full absolute path in – which is useless if we're going to share a script with other developers on our team. Also, there's a load of repetition in the paths for both `Humanizer` and `Newtonsoft.Json`. We can fix that as follows: -

#### Listing 26.4 Loading a source file into a script with a NuGet dependency

```
#I "..\packages\" #A
#r @"Humanizer.Core.2.1.0\lib\netstandard1.0\Humanizer.dll" #B
#r @"Newtonsoft.Json.9.0.1\lib\net45\Newtonsoft.Json.dll"
```

#A Add the `..\packages\` folder to the search list using a relative path.  
#B Simplified NuGet package reference.

Firstly, we've moved to using a relative path to the script location i.e. "...\\packages\\" rather than an absolute path – this means we can share this script with the rest of the team, and secondly we've used the #I directive so that we don't have to retype that in every #r directive. We could also have not bothered with the #I directive, and simply copied that to the start of both the #r directives.

### NuGet and Projects references

The way NuGet interacts with .NET projects is going through something of a redesign in the next version of the .NET project system; it might even be that projects can reference nuget package directly (rather than assemblies in those packages). Similarly, there's talk of a #nuget directive that *might* be added to F# scripts - but that's just speculation at this point.

### 26.2.3 Auto-generated references

Even when using the #I trick, maintaining the list of dependencies in a script (in the correct order!) can be a pain. Thankfully, F# Power Tools has a solution to the whole issue of referencing project and source files.

#### Now you try

Let's use the auto-generated references from Power Tools to see how it can help matters.

1. Right-click the **References** node of the project and select the "Generate References for F# Interactive".
2. A new folder, `scripts`, will be created in the project, along with two files – `load-references` and `load-project`.

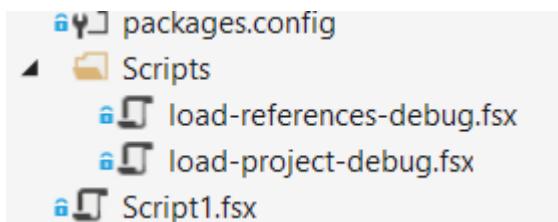


Figure 26.3 – Generated script files from Visual F# Power Tools

3. Open the two files. You'll see that the former contains #r directives for *all referenced assemblies*, whilst the latter contains a #load for *every source file in the project* (and calls the references file too!).
4. Remove all the #r and #load directives from the script you created, and replace them with a single #load @"Scripts\load-project-debug.fsx".

What's especially good about this is that the files are regenerated whenever new dependencies or files are added to the project – so you don't have to maintain them manually.

### **When can't I use auto-generated references?**

If you're sticking with Visual Studio and F# projects, auto-generated references files work fine. However, if you have a C#-only solution but still want to use F# scripts for exploration (which is not uncommon), you have two options. Firstly, whilst you *can* use standalone F# scripts in a C# project (or as a solution item), you *won't* get the auto-generated reference scripts, as these are only available to create in F# projects. The alternative is to create an F# project which contains nothing but your exploratory scripts, and within which you can also create the auto-generated reference scripts.

### **Quick Check**

1. Can you use NuGet packages with F# projects?
2. What do you need to be aware of when calling `#load` on F# files?
3. How can we make life easier when referencing assemblies and files?

### **Answers**

1. Yes.
2. Any dependent assemblies and F# files will need to be imported first using `#r` or `#load`.
3. Use Visual F# Power Tools ability to auto-generate references and project load scripts.

## **26.3 Working with Paket**

NuGet is a great tool – it simplifies sharing dependencies across .NET projects and acts as a central repository for reusable .NET components. However, it does have several shortcomings, partly due to the way in which it first came into being, and partly due to design decisions that were made over the years.

Paket is an open source, flexible and powerful dependency management client for .NET which aims to simplify dependency management in .NET. It's backwards compatible with the NuGet service, so you can continue to use existing NuGet packages - but it provides an alternative *client-side* application which replaces the existing NuGet client which also adds a whole host of new features. It's written in F#, but fully compatible with C#, VB .NET and F# projects and solutions. *Disclaimer: I'm one of the contributors to the Paket project.*

There's another reason I'm showing you Paket – if you look at virtually any F# open source project, or start to read any F# examples etc. online, virtually all of them will use Paket rather than NuGet – so it's worth getting up to speed with it sooner rather than later.

### **26.3.1 Issues with the NuGet Client**

Here's just a few issues you've probably come up against in the past with NuGet: -

- **Invalid references across projects:** NuGet does not prevent you from adding the

same package to two projects but with *different versions*. In other words, you may have a reference to Newtonsoft.Json version 6 in Project A, and version 7 in Project B within the same solution. You won't get a compile error, but might get a run-time error, depending on any number of factors.

- **Updates project file on upgrade:** Every time you update a NuGet dependency, the project file changes, because NuGet packages store the version in the physical path. This can cause merge conflicts, as well as unnecessary changes to the project file.
- **Hard to reference from scripts:** Because the physical path is stored in the packages folder, scripts are tightly coupled to them. If you update a package, your scripts will break (unless you're using the generated references file from VFPT).
- **Difficulty managing:** NuGet is difficult to reason about on large solutions (or multiple solutions sharing projects), because NuGet doesn't really have a unified view of dependencies across all projects or solutions (although admittedly NuGet 3 did put in some UI tricks to make this experience a little better). How often have you worked on a large solution or project and deliberately put off upgrading NuGet packages because you're afraid that upgrading will *somewhat* break something?

### 26.3.2 Benefits of Paket

Paket addresses *all* of the issues above, as well as adding several new features, including: -

- **Dependency resolver:** Paket understands your dependencies across all projects in your solution (or repository), and will keep all your dependencies stable across all projects. It will not allow you to accidentally upgrade a version of a dependency for only a *part* of your solution.
- **Easy to reason about.** You do not have to worry about "child" dependencies of nuget packages. Paket allows you to focus on the "top level" dependencies, whilst it internally manages the children for you without you needing to worry about them.
- **Fast:** Paket is extremely fast, with an intelligent resolver and caching mechanism so that restoring packages occurs as quickly as possible.
- **Lightweight:** Paket is a command-line first tool. It has an extension for Visual Studio as well, but this is essentially a wrapper around the command line tool, rather than the other way around. You don't *need* a GUI to add packages to your solution or project – the configuration files are plain text, lightweight and easy to maintain.
- **Source code dependencies:** You can have a dependency on e.g. a specific commit of a GitHub file. This is extremely useful when working with tiny dependencies e.g. helper or utility modules or the like, that don't justify creating a NuGet package for.

Why am I showing you Paket here? One of the things you'll notice if you start working with any F# open source projects in the future is that virtually all of them use Paket rather than NuGet – not only does it have many advantages over NuGet, but it also plays much more nicely with F# - scripts work more easily with Paket-sourced dependencies, whilst it also

doesn't couple you to Visual Studio if you decide you want to e.g. use Visual Studio Code (or any other IDE for that matter).

### **Now you try**

In this exercise, we'll convert our existing package from NuGet to Paket.

1. In the existing solution, add the `WindowsAzure.Storage` nuget package.
2. Open the `packages.config` file. Observe that it has approximately 50 nuget packages. Which package is dependent on which? Why are we seeing 50 packages when we only asked for three (Humanizer, Newtonsoft.Json and WindowsAzure.Storage)?
3. Navigate to the latest Paket release and download the `Paket.exe` application (<https://github.com/fsharp/Paket/releases/latest>) to the root folder (alongside the solution file).
4. Delete the *entire* packages folder.
5. Open a command prompt and navigate to the root folder of the solution.
6. Run `paket convert-from-nuget`. Paket will now convert the solution from NuGet tooling to Paket. Observe the following: -
  - o All packages are downloaded in the packages folder but *without version numbers*. Paket does not include version numbers in paths. This makes referencing NuGet packages much easier from F# scripts!
  - o Two new files have been created – `paket.dependencies` and `paket.lock`. The former file contains a list of all “top level” dependencies and is designed to be human-readable and editable; the latter contains the tree of inter-dependencies.
  - o Your project will have been updated so all nuget packages reference the new (version-free) paths.
7. Run `paket simplify`. This will parse the dependencies and “strip out” any packages from the `paket.dependencies` file that are not “top-level” ones. Observe that the dependencies file now only contains two dependencies – Humanizer, and WindowsAzure.Storage (Json is a dependency of WindowsAzure.Storage). The lock file still maintains the “full” tree of dependencies.
8. You can now open `Script1.fsx`; observe that the references are currently broken. Rebuilding the solution will re-generate the references script to point to the correct locations. Notice that the paths *no longer have the version numbers in them*. In future, updating nuget dependencies in Paket will not break scripts that reference assemblies simply because the version changed.

There's tons more on Paket than I've shown you here, so it's well worth looking on the Paket website at the documentation there. It contains guidance on all the features, quick starts, plus it has a responsive team that will answer questions on GitHub or Twitter.

### 26.3.3 Common Paket commands

Here are some command Paket commands: -

1. `paket update` – Updates your packages with the latest versions from NuGet. By default, Paket will select the *highest* version of any package available, and will intelligently ensure that the latest versions are compatible across all your dependencies.
2. `paket restore` – Brings down the current version of all dependencies specified in the lock file. Useful for CI processes etc. to ensure repeatable builds.
3. `paket add` – Allows you to add a new nuget package to the overall set of dependencies e.g. `paket add nuget Automapper project NugetFSharp` gets the latest version of the Automapper nuget package and adds it to the NugetFSharp project.

There's also a Visual Studio extension for Paket (available in Visual Studio Extensions and Updates) which provides much of this functionality directly within Visual Studio.

---

#### Paket as an example of open source collaboration

Paket is a good example of how open source, community-led projects can work. What started as a small project with just a couple of developers now has dozens of contributors and is used in many organisations. It's a good example of how a set of developers saw what they felt was room for improvement in the existing NuGet story, and were able to create a new tool rapidly that fitted the needs of many developers. Many of the features in Paket were originally thought to not be important by the NuGet team, but now there are signs that some of them might now be introduced into NuGet as well at some point.

I should point out though that this has not always been a pain-free journey. The Paket and NuGet teams have not always had the most positive relationship (although this has improved over time). Also, if you're used to the somewhat lethargic pace of NuGet updates, you might be in for a shock with Paket – it's not unusual to have *intra-day* updates and fixes to the tool!

---

#### Quick Check

4. Why can it be difficult to work with NuGet packages from F# scripts when using the NuGet tool?
5. What does the `paket.dependencies` file contain?

## 26.4 Summary

In this lesson, we got our hands dirty with NuGet and F#.

- We saw how to, and why we might want to, use NuGet packages within F# scripts.
- We learned some tips for how to make working with scripts even easier through Visual F# Power Tools.
- We gained a brief introduction to the Paket dependency manager.

### **Try This**

Take an existing .NET solution you've been working on. Try first to create an F# project in the solution, referencing the same NuGet packages as the C# project. Then, generate a references script to the project and see if you can start to work with those nuget packages from your own script! Then, try converting the solution from NuGet to Paket. You might even find that Paket refuses to convert, if there are discrepancies in your NuGet configuration (such as different versions of the same package)!

### **Quick Check Answers**

1. Yes.
2. Any dependent assemblies and F# files will need to be imported first using `#r` or `#load`.
3. Use Visual F# Power Tools ability to auto-generate references and project load scripts.
4. NuGet uses the version number of the package within the path of the package, so every time you update a package, your F# scripts will break.
5. The set of top-level dependencies for your solution.

# 27

## *Exposing F# types and functions to C#*

In this unit (and indeed throughout this book) we've concentrated at how to consume C# from F#. However, it's actually common to also go the other way, and write libraries in F# and consume them in C#. We'll look at: -

- F# data types
- Namespaces and Modules
- F# functions
- Gotchas when consuming F# code from C#

Even in an existing codebase that's essentially mostly C#, you'll still have cases where you wish to work with F#. We'll cover more of this in the coming lessons, but it's not that an unusual situation – particularly when working on larger projects or with existing code-bases so here are some examples of cases where you're interoperating between F# and C#.

### 27.1 Using F# types in C#

We're fortunate that the F# team spent time looking at the situations that arise when exposing data created in F# to C#, because they've done a really great job in nearly all common cases. As you'll see, F# data types actually all boil down to primitives that you already know – it's simply that the F# language allows us to work with those same primitives at a higher level. Let's first look at the common F# data types we've discussed in the first half of the book.

#### 27.1.1 Records

Records map extremely well in C#. They appear as regular classes, with a *non-default* constructor that takes in all fields exposed in the record. A default constructor will *not* normally be generated – so it won't be possible to create the record in an uninitialized (or partially initialised) state. Each field will appear as a public getter-only property, and the class

will implement various interfaces in order to allow structural equality checking. Also, although I've not shown it in this book, you can create *member functions* on a record. These are exposed as methods in C#.

### Now you try

Let's create a mixed solution, which we'll use to create a number of F# types and explore how they render in C#.

1. Create a new solution in Visual Studio, **Interop**.
2. Create an F# class library, **FSharpCode**.
3. Create a C# console application, **CSharpApp**.
4. Reference the **FSharpCode** project from the **CSharpApp** project so that you can access the F# types from the C# project.

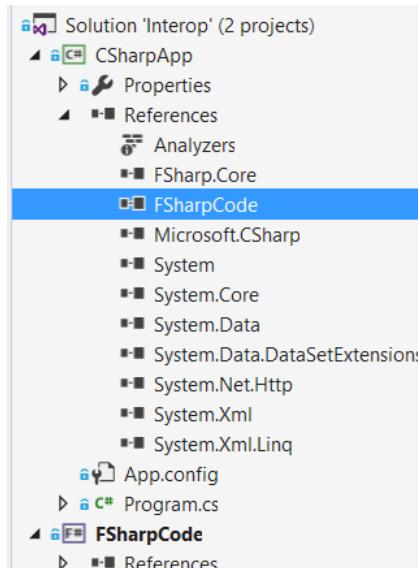


Figure 27.1: Creating a mixed-language solution and referencing F# from C#

5. Open `Library1.fs`, remove the sample class definition and rebuild the solution.
6. Now, in `Library1.fs` create a simple record type to model a car, taking care to change the namespace as per below.

#### **Listing 27.1 An F# record to be accessed from C#**

```
namespace Model

/// A standard F# record of a Car. #A
type Car =
 { /// The number of wheels on the car. }
```

```

 Wheels : int
 /// The brand of the car.
 Brand : string }

```

#A Record definition using lightweight triple-slash comments.

7. Now that you've created the type, go to `Program.cs` and within the `Main()` method, try typing `Model` to get intellisense for the namespace. You'll see that nothing appears! This is because C# and F# projects can (currently) only see changes in code once you've *compiled* the child project – in this case, the F# project.
8. Go ahead and rebuild the solution. You'll see that you can now create an instance of the F# record type, although it appears as a class to C#.

```
var car = new Car(4, "Supacars");
```

Car.Car(int wheels, string brand)

Figure 27.2: Creating an F# record from C#

9. Notice that you can access getter-only properties on the car that map to the fields. Also observe that the triple-slash comments show in tooltips.
10. Try to use **Go To Definition (F12)** from the C# project when the caret is over the `Car` type. Observe that a C# rendering of the F# type, based on its IL metadata, is shown. Also notice the interfaces that are implemented for structural equality e.g. `IEquatable<T>` and `IComparable<T>`, as well as override `Equals` and `GetHashCode`.

## 27.1.2 Tuples

Tuples in F# are simply instances of the standard `System.Tuple` type, and so they appear as such when consumed from C#, with standard `Item1`, `Item2` and `ItemN` properties etc. The standard .NET tuple type only supports up to 8 items, so F# has a trick up its sleeve here – if you have a tuple wider than 8 items – let's say 10 items – the eighth element of the Tuple will itself be *another* tuple which has the last three items. However, I'd strongly recommend avoiding ever getting in a situation where you have a tuple more than three items wide – stick to Records for such a case.

---

### Tuples in C#7

C#7 will almost certainly have *language support* for Tuples, much like F# currently has. However, this will actually be a brand new type (most likely called `System.ValueTuple`) which, unlike `System.Tuple`, is a *value* type. In order to seamlessly interop with this, the next version of F# will also introduce a new `struct` keyword which will use tell the F# compiler to also use the `System.ValueTuple`, so that exposing this to C# should allow you to take advantage of C# language support as well, but for now there's nothing to worry about.

---

### 27.1.3 Discriminated Unions

Remember that discriminated unions in F# are roughly equivalent to a class hierarchy in C#, except that they are a *closed* set of classes. And, sure enough, if you try to consume a discriminated union from F#, that's exactly what you'll see – a set of classes (one per case), along with a set of static helper methods to allow you to both easily check which "case" the value is, and also to create instances of a case yourself.

However, be aware that there's one fundamental problem with using discriminated unions in C# - without any support for pattern matching (something that will be *partially* rectified in C#7 – it'll probably have support matching over type checks), you'll quickly find that it can be painful to reason about a discriminated union in C#. Remember that in C# with inheritance, behaviours are part of the class and we use polymorphism to access different implementations through virtual dispatch. Discriminated unions don't have behaviours on them but rather separate standalone functions that operate over *all* cases through pattern matching.

#### Now you try

Let's now enhance our solution to illustrate Tuples and Discriminated Unions.

1. Update `Library1.fs` as follows: -

#### Listing 27.2 Creating Tuples and Discriminated Unions in F#

```
/// A standard F# record of a Car.
type Car =
 { /// The number of wheels on the car.
 Wheels : int
 /// The brand of the car.
 Brand : string
 /// The x/y of the car in meters
 Dimensions : float * float } #A

/// A vehicle of some sort.
type Vehicle = #B
 /// A car is a type of vehicle.
 | Motorcar of Car
 /// A bike is also a type of vehicle.
 | Motorbike of Name:string * EngineSize:float

#A A property on the record that is a tuple.
#B A discriminated union using both record and inline arguments.
```

2. Rebuild the F# project, and then correct the C# to create a Car – you'll need to pass in a `Tuple<Float, Float>` for the new argument e.g. `Tuple.Create(1.5, 3.5)`.
3. Notice that if you access the `Dimensions` property on the car, you'll simply get `Item1` and `Item2` for X and Y.

Let's now look at discriminated unions. We created a discriminated union in Listing 27.2 which has two cases – **Motorcar** and **Motorbike**.

4. In Intellisense, navigating to Model.Vehicle will look as follows:

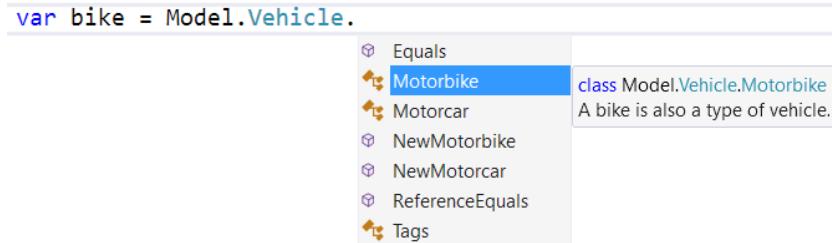


Figure 27.3 – How to create cases of a discriminated union from C#

You can see both cases as nested types underneath the `Vehicle` type. Beware - there are no constructors available for either case – instead, we're provided with *builder* methods such as `NewMotorbike` which allow us to create instances of the cases.

5. Create an instance of a `Motorbike`.
6. The type of value returned back will be a `Vehicle` – not a `Motorbike`! In order to “test” which type the variable is, you’ll need to use the `IsMotorbike` and `IsMotorcar` properties on the vehicle instance, and then cast it as appropriate. Only then will you be able to access the properties on the `Motorbike` itself.

In effect, what this is forcing you to do is adopt a clear separation between data and behavior, and use type-based pattern matching (which in C#6 means resorting to casts or the `as` keyword etc.) to access the “real” data.

### **Quick Check**

1. How are Records represented in C#?
2. Why are Discriminated Unions sometimes difficult to reason about in C#?

## **27.2 More on F# interop**

### **27.2.1 Namespaces and Modules**

We've seen how to use namespaces and modules to logically group types and functions together in F#. How are they exposed in C# though?

#### **F# NAMESPACES IN C#**

Namespaces in F# are not only logically the same as in C# - they're essentially exactly the same thing. So, if you make a type within a namespace in an F# assembly, you can reference it in C# using the exact same namespace. Easy.

## MODULES IN C#

A module is rendered in C# as a *static class*. Any simple values on the module such as an integer or a record value will show as a public property. Functions will show as methods on the static class and types will show as nested classes within the static class. As you can see, there's a pretty good mapping for these. In fact, you probably won't even know that you're accessing an F# module from C#!

### 27.2.2 Using F# functions in C#

As we know, functions in F# come in two forms –  *tupled* and *curried*. F# will render both to C# as though they were tupled – so all arguments will be required at once, unlike curried functions in F# where you can pass in just a subset to return a new function. There's one exception where this breaks down – if you expose an already *partially-curried* function to C#, it'll look pretty strange. If you can avoid trying to read the intellisense, it actually works reasonably well – but it's completely non-idiomatic C#.

#### Now You Try

Let's now experiment with some F# functions first-hand to see how they render in C#.

1. Enter the following code at the bottom of `Library1.fs`.

```
Listing 27.3 Exposing a Module of functions to C#
module Functions =
 /// Creates a car
 let CreateCar wheels brand x y = #A
 { Wheels = wheels; Brand = brand; Dimensions = x, y }
 /// Creates a car with four wheels.
 let CreateFourWheeledCar = CreateCar 4 #B

#A Function in curried form
#B Partially applied function
```

2. Rebuild the F# project.
3. Call the `Model.Functions.CreateCar` function from C#. It appears as a normal static method.
4. Call the `Model.Functions.CreateFourWheeledCar` function from C#. It appears as a property of type `FSharpFunc`, which has an `Invoke` method on it that takes in a single argument (you'll need to add a reference to `FSharp.Core` to see this – take the newest one you can find, which is 4.4.0.0 at the time of writing).
5. Observe that calling `Invoke` will return another property with another `Invoke` method! Each call relates to one constructor argument (except the first, which has been supplied in the F# code already!): -

#### Listing 27.4 Calling F# functions from C#

```
var somewheeledCar = Model.Functions.CreateCar(4, "Supacars", 1.5, 3.5); #A
```

```

var fourWheeledCar =
 Model.Functions.CreateFourWheeledCar #B
 .Invoke("Supacars")
 .Invoke(1.5)
 .Invoke(3.5);

```

#A Calling a standard F# function from C#.  
#B Calling a partially applied F# function from C#.

To cut a long story short – try to avoid providing partially applied functions to C#; if you absolutely must, wrap them in a “normal” F# function that takes in all arguments required by the partially-applied version and supplied them to it manually.

### Quick Check

3. How are modules declared in F# rendered in C#?
4. Can you use F#-declared curried functions in C#?

## 27.3 Summarising F# to C# Interop

This table roughly summarises how well (or not) different elements operate in C#.

**Table 27.1 – Summarise F# to C# interoperability**

| Element              | Renders as                   | C# compatibility |
|----------------------|------------------------------|------------------|
| Records              | Immutable class              | High             |
| Tuples               | System.Tuple                 | Medium / High    |
| Discriminated Unions | Classes with builder methods | Medium / Low     |
| Namespaces           | Namespaces                   | High             |
| Modules              | Static Classes               | High             |
| Functions            | Static Methods               | High / Medium    |

### 27.3.1 Gotchas

This section covers a few edge cases where you might need to do something a little different in order to use a specific type in C#.

#### INCOMPATIBLE TYPES

There are a few types in F# which simply do not exist in C#. Generally, this is because there's no CLR support for them, and they're erased at compile time. The two main elements are unit of measure (which we haven't touched on) and Type Providers (which we'll deal with in the upcoming lessons).

### **CLI MUTABLE**

There are rare occasions where you'll need to create an F# record from C# in an uninitialized state (i.e. without having provided all fields to a constructor), or without getter-only properties. Primarily this is important for interop with third-party libraries that create objects using reflection – they'll typically create an uninitialized object first, and then set each property one at a time e.g. MongoDB and Azure Web Jobs SDKs are two examples. By default, these libraries won't work with F# records, so to get around this, you can place the [`<CLIMutable>`] attribute on a record. This won't change anything from an F# point of view, but will affect the underlying IL that is emitted so that C# code will be able to access a default constructor, and properties will have setters.

### **OPTIONS**

You *can* consume F# option types in C#, once you add a reference to `FSharp.Core`. However, as with other discriminated unions, they're not particularly idiomatic to work with in C#. Adding a few well-placed extension methods that remove the need for supplying type arguments can help though, so it's worth looking at this if you want to use F#'s Option type in C#.

### **ACCESSIBILITY MODIFIERS**

F# also supports accessibility modifiers, just like C# e.g. `public`, `private` and `internal`. Unlike C#, things are *public* by default in F#, but if you want to make a function or value hidden from C# code, simply mark it as `internal`.

### **COLLECTIONS**

F# arrays are simply standard .NET arrays, so work without a problem. Likewise, sequences appear as `IEnumerable<T>` to C# code. However, the F# list is not the same type as the standard .NET generic list (known in F# as `ResizeArray`). Again, without pattern matching (and the `List` module), it's of limited use – although as it implements `IEnumerable<T>` you *can* use LINQ on it. My advice is to avoid exposing it to C# clients – Arrays and Sequences work fine, and you won't place a dependency on the `FSharp.Core` assembly on callers with those.

#### **Quick Check**

5. What do Tuples render as in C#?
6. What is the purpose of the [`<CLIMutable>`] attribute?

## **27.4 Summary**

In this lesson, we looked at how to expose code from F# to C#.

- We saw how records, tuples and discriminated unions can be consumed in C#

- We saw how to work with namespaces, modules and functions
- We learned what parts of F# don't map to well into the mostly-OO C# language
- We saw how to help smooth interoperability issues for certain common cases

### **Try This**

Take an existing application that you already have that's written in C#. Try to port your domain model from C# to F# and then reference it from C# - try both doing a "simple" mapping, without using any "advanced" F# modelling features, before trying out a more complex model that uses e.g. Discriminated Unions and Options. Then, try moving business logic from C# to F# and functions on modules, rather than stateful classes.

### **Quick Check Answers**

1. As classes with a default constructor, and public getter-only properties.
2. C# does not have rich pattern matching support which can make DUs difficult to work with.
3. As static classes.
4. Yes, although they can be unusual in C# to work with.
5. The System.Tuple type.
6. CLIMutable provides a non-default constructor and public setters for all properties.

# 28

## Architecting Hybrid Language Applications

The final lesson of this unit will look at all the elements we've discussed so far within the context of a larger, cohesive element. As such, it won't have specific step-by-step exercises as we'll be reviewing a pre-built codebase. After this lesson, there'll be a capstone exercise which will build on this in the context of the banking application you've been writing throughout this book. We'll look at:

- Crossing the boundaries from F# and C#
- Playing to strengths of a language
- Case study – driving a WPF application from F#

### 28.1 Crossing language boundaries

We've touched on this briefly earlier in this unit, but it's worth reviewing this point in a little more depth. Whilst beneath the covers F# and C# share the same runtime, at *compile-time* F# affords us much more safety, thus allowing us to focus on solving business problems rather than checking for nulls and so on throughout our codebase. Let's look briefly at how we can work between the two languages in an attempt to get the best of both worlds – taking advantage of the F# type system whilst still being friendly to C#.

#### 28.1.1 Accepting data from external systems

It's extremely useful to use the F# type system to model our domain effectively – but when interoperating with other systems (or languages), there may be an "impediment mismatch" between F# and the "other" side. For example, F# features such as discriminated unions (sum types) and non-nullability by default don't exist in C#. As such, there will be times when you want to model something in such a way that you need to choose one of two approaches to take.

On the one hand, if we expose data structures that are foreign to C# developers, we make our API tricky to consume. On the other hand, using a simple data model that's usable everywhere means giving up features in F#. And this is about more than about exposing APIs to C# - sometimes, you'll be forced to deal with "dirty" data from external sources e.g. JSON, CSV etc., or HTTP endpoints etc. etc. – in other words, data that you don't "trust" to adhere to a particular schema, or one that doesn't allow defining domains as richly as F#.

A good way around this is to define an "internal" F# domain which contains all the niceties that we've seen throughout this book – discriminated unions, option types, records etc. etc. while *also* using a "public" API designed to be easy for consumers to work with (e.g. C# developers). When we move in and out of the F# world, we "marshal" data between the two formats – this is particularly important when going from the "weakly typed" external shape to the internal "stricter" F# shape. This is shown in Figure 28.1 to go from data structures that cannot encode rich schema information into the F# world.

### **SQL and C# - an impedance mismatch of types**

The designers of SQL got one thing spot on – they allowed *any* data type to be marked as either null or non-nullable. In the early days of C#, there was no concept of `Nullable<T>`, so when reading e.g. nullable ints from a database, you needed to pick a "default" integer value to map to in case there was no value on the database.

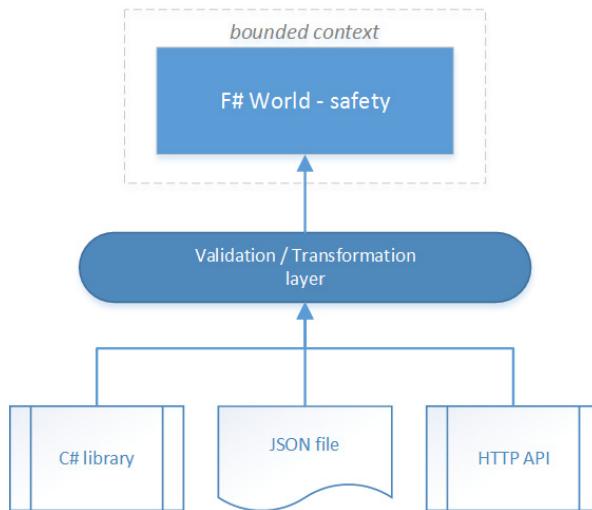


Figure 28.1 – Providing a gateway into the F# world as a means of ensuring type safety

Let's see a simple example for working with a simple domain that we wish to use from C# from a "public API" point of view – but wish to do all the actual calculation from within F#. This model represents a simple order system. An order has an id, a customer name, the set of

items to order, plus an optional way to contact the customer to keep them updated of shipping progress.

### **Listing 28.1 A simple domain model for use within C#**

```
type OrderItemRequest = { ItemId : int; Count : int }
type OrderRequest =
 { OrderId : int
 CustomerName : string // mandatory #A
 Comment : string // optional #B
 /// One of (email or telephone), or none
 EmailUpdates : string #C
 TelephoneUpdates : string
 Items : IEnumerable<OrderItemRequest> } // mandatory

#A A mandatory string through convention
#B An optional string
#C A set of related properties
```

Even a relatively simple domain model such as this has a set of “implicit” rules that are documented through code comments and the like. We’re using Records here as they are extremely lightweight and works well in C# – but there are better ways to model this in F#: -

### **Listing 28.2 Modelling the same domain in F#**

```
type OrderId = OrderId of int
type ItemId = ItemId of int
type OrderItem = { ItemId : ItemId; Count : int }
type UpdatePreference =
 | EmailUpdates of string
 | TelephoneUpdates of string
type Order =
 { OrderId : OrderId
 CustomerName : string #A
 ContactPreference : UpdatePreference option #B
 Comment : string option #C
 Items : OrderItem list }
```

```
#A CustomerName should never be null
#B Improved modelling for shipping updates
#C Comment is explicitly marked as optional
```

It’s relatively simple to go from a weaker model to a stronger model - at the “entrance” to our F# module, we accept the weak model, but immediately validate and transform it over to our “stronger” model. Once in this shape, we no longer have to check for nulls or otherwise invalid data – and can immediately benefit from the improved modelling capabilities.

### **Listing 28.3 Validating and transforming data**

```
{ CustomerName =
 match orderRequest.CustomerName with #A
 | null -> failwith "Customer name must be populated"
 | name -> name}
```

```

Comment = orderRequest.Comment |> Option.ofObj #B
ContactPreference = #C
match Option.ofObj orderRequest.EmailUpdates, Option.ofObj orderRequest.TelephoneUpdates
with
| None, None -> None
| Some email, None -> Some(EmailUpdates email)
| None, Some phone -> Some(TelephoneUpdates phone)
| Some _, Some _ -> failwith "Unable to proceed - only one of telephone and email should
be supplied" }

```

#A Simple null check  
#B Explicitly marking an optional string  
#C Safely creating a discriminated union from flattened data

In this (simplified) example, we perform a number of checks before entering our “safe” F# world: -

- Null check on a string
- Convert from a string to an optional string
- Confirm that the source request has a valid “state” – if the incorrect mix of fields are populated, the request is rejected.

### Working with strings in F#

In F#, I prefer to be explicit about nullable fields – sadly, it’s not possible in F# to make strings non-nullable, as they come from the BCL (the same as any other C# class) – so essentially every string *could* be null, even if we know it would never be. So, whilst theoretically, you should wrap *all* reference types in F# Options in order to be completely safe, I tend to take a more pragmatic approach: -

1. If you know that a string field could conceivably be null, convert it to an option type using `Option.ofObj`.
2. If it shouldn’t ever be null, check at the F# boundary and reject the object if it’s null; if it’s not null, just leave it as a string.

In this way, we still gain a safety net, and can model the distinction between “optional” strings and “mandatory” strings – whilst not incurring the cost of placing option types throughout your codebase.

### 28.1.2 Playing to the strengths of a language

At the start of this book, I briefly distinguished between features that are natural to F# and those in C# – for example, mutability, expressions and the like. There are times when you’ll still want to use C# in a large system – either because of tooling, or the domain at hand. Here are some examples: -

- **ASP .NET MVC GUIs** – C# has rich support with Razor syntax for creating HTML GUIs on the server. There *are* third-party templates (downloadable directly from within Visual Studio) to allow you to creating an MVC application in F#, but a low-frills way to get going is to create all your views and web host in a C# project, and delegate to F# for your controllers (or core business logic) onwards. Given the stateless nature of web

applications, F# is a great fit – and we’ll see in a couple of units’ time how simple this is to do.

- **Windows Forms GUIs** – C# again benefits from code generation for GUIs, but more than that, local client GUIs are generally mutable by nature – again, not necessarily an idiomatic fit for F# (although you can do it). There are also some good third party libraries such as Fody that make binding C# classes to WPF applications with MVVM very easy. I know of developers that use F# for the entire WPF stack, including code-behind views, through the ingenious *FsXaml* and *FSharp.ViewModule* projects, others that leave code-behind as C# but use F# for view models, and yet others that have their view models in C# but services and below in F#. There’s no “right” answer here – experiment with different blends of F# and C# to see what feels more natural to you.
- **Dynamic code** – When working with *truly* dynamic data structures, F# isn’t a great fit. There are two options – the `? operator`, which allows you to perform a limited amount of dynamic coding, or using type providers – again, see the next unit for this.
- **Entity Framework** – I have to say, since working with F#, I’ve gone completely “off” Entity Framework, simply because F# has far better data access libraries for most use cases. Nonetheless, if after working through the next unit, you *still* want to use EF, I’d exercise caution about defining your code-first models *in* F#. EF is designed to work with inherently mutable classes, using virtual methods etc. – things that you *can* definitely model in F#, but again it’s probably a better fit for C#.

### Quick Check

1. What features from the F# type system might be missing for simpler domains such as JSON or CSV?
2. Why should you consider having a rich internal and simpler external domain?
3. Can you name a scenario where C# might be a better fit than F#?

## 28.2 Case Study – WPF Monopoly

Let’s now look at a slightly larger application that uses a combination of C# and F# projects to provide a cohesive end-to-end experience for a WPF application that models most of the rules of the classic Monopoly game. Rather than writing all the code ourselves, you can download the source code here <https://github.com/isaacabraham/monopoly> (I make changes to this repository occasionally – the specific commit that we’ll be working from is 092c53d). Go ahead and download, build and open the MonopolyStats solution – let’s take a look into some of the aspects of it.

### 28.2.1 Application Overview

Without getting into the nitty gritty of Monopoly, the essence of it is that it’s a board game in which several players take turns throwing dice to move around a board of land properties, buying and selling them whilst making money by renting them out when other players land on

their properties. While our application doesn't model the *entire* game, it *does* model the core parts: -

- All of the board pieces
- Rules regarding rolling on special places such as "Chance" and "Go to Jail"
- Rules such as throwing consecutive doubles

The application then allows you to roll dice randomly either one at a time, or repeatedly roll the dice hundreds (or thousands) of times, accumulating statistics of which properties on the board are landed on the most.

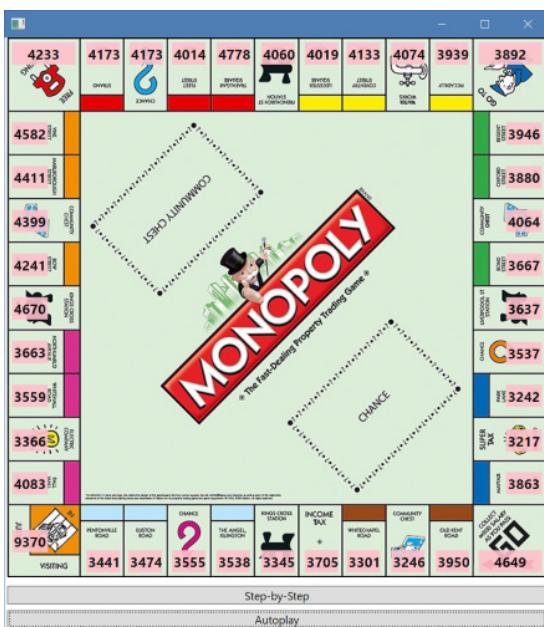


Figure 28.2 – A C# WPF GUI application running on top of an F# back-end

The interesting parts of this application center around how the application is modelled in F#, and then how we interact with it from C#.

### 28.2.2 Separating UI concerns from domain logic

The application is separated into four real components: A core domain written in F#, as well as three "clients": the WPF application you see here, plus an F# script used during development as well as a simple F# console application. It might be interesting to note that the WPF GUI was actually written *last* – the codebase was developed using the REPL and script-first, before adding a console and WPF application on afterwards.

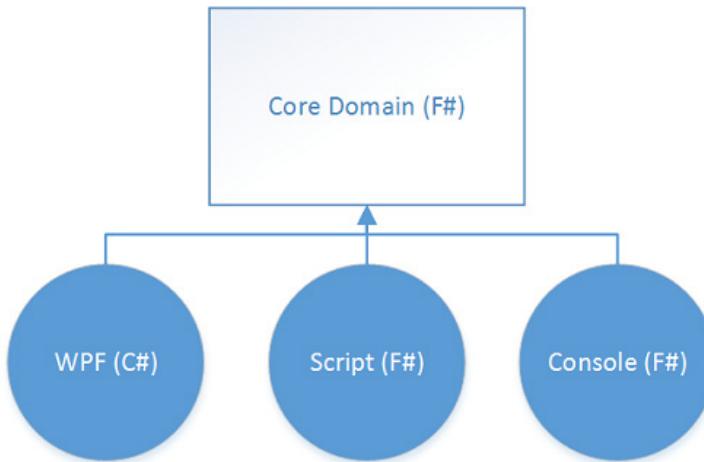


Figure 28.3 – A core domain that can service multiple consumers

The core domain is set in three files in the MonopolyStats project: -

- **Types.fs** – Contains the different domain types used in Monopoly, such as the board pieces and decks of cards.
- **Data.fs** – Contains the entire data model based on the defined types – so the layout of the board and the two decks of cards
- **Controller.fs** – Contains the application logic and rules to drive the main program, and represents our public API. The code in here is split into a module with our standalone, stateless functions and a (simpler) public class for consumption in C#.

### Active Patterns

In Controller.fs, you'll see two examples of *Active Patterns* – a lightweight form of discriminated unions. In the first one, we classify a number of doubles as either ThreeDoubles or LessThanThreeDoubles which we can then pattern match over later – this can improve readability and enables simple reuse of classifications.

The Test.fsx script contains some simple operations that we can perform on the game logic, including experimenting with some standalone functions before testing the "full" Controller class out to play a game. It's worth looking through this code in your own time, but there are some key points to make note of, which I'll address now – have the solution open as you go through this!

#### 28.2.3 Expressions at the core

The actual logic for the game is handled in the `Monopoly.Functions` module, such as `moveBy` (given a current board position, moves by a specific distance), `playTurn` (plays a single move)

and `tryMoveFromDeck` (optionally moves the player when landing on a card deck). Observe that all of these functions are *expression-based*, operating on state that is supplied to them and returning a new state. Even the overall `playGame` function returns a list of all the moves that occurred; it builds the history up through `scan` (itself a derivative of `fold`) – no mutation is involved.

These functions are not all guaranteed to be *entirely* pure (some of them use functions that *may* be impure e.g. `rollDice` has a signature `unit -> int * int`), but nevertheless you can call each of these functions directly from a script with *repeatable* results and no “hidden” dependencies, which is crucial for easy testing and exploration. Remember – it’s relatively easy to “dumb down” an expression-based API to become statement-based – but it’s *very* difficult to go the other way, so you should always strive to start with expressions if possible.

#### 28.2.4 C# Interop

When called from C#, I’ve actually wrapped the calls into the `Functions` module into a `Controller` class with a single method on it, `PlayGame()` (notice the Pascal casing, which is also C#-friendly). This class exposes an `OnMoved` event that is fired whenever a move occurs (so that they can be shown in the GUI as they happen). Notice that this event is decorated with the `CLIEvent` attribute, which is needed to expose events to the C# world – unfortunately, you can’t have `CLIEvents` on modules.

#### 28.2.5 WPF and MVVM

The entire WPF and MVVM layer has been implemented in C#. There are two commands in the application – one to play 50,000 turns in succession; the second to play one turn at a time. The commands both set up event handlers to the `OnMoved` event from the `Controller` class, and use that to update the GUI with the number of times a position was landed on. There’s nothing here that should be unfamiliar to a C# WPF developer – the only difference is that we’re calling F# to “drive” our view models rather than C#. Here’s a snippet from the `AutoPlayerCommand` class that shows us calling F# from a C# class: -

##### **Listing 28.4 Calling F# from a C# view model**

```
var controller = new Controller(); #A
controller.OnMoved += (o, e) => #B
 positionLookup[e.MovementData.CurrentPosition.ToString()].Increment();
Task.Run(() => controller.PlayGame(50000)); #C

#A Create an instance of the F# controller class
#B Adding a standard event handler to capture game events
#C Having the F# code play 50000 turns on a background thread
```

#### 28.2.6 Randomness

The last point of interest from an FP point of view is how we handle randomness within the application. In the “real” application, we use the `System.Random` class to generate dice

throws and pick Chance cards. But when testing and exploring, we want to have repeatable, deterministic results. Looking at the `playTurn` function, we can see the first two higher-order functions it takes in are to roll dice and pick a card. Both are simple functions that have *no dependency* on `System.Random`. By doing this, we can now easily test this function from a script as follows: -

### **Listing 28.5 Using deterministic functions for exploration**

```
let rollDice() = 3, 4 #A
let pickCard() = 5 #B
let startingPosition = { CurrentPosition = Go; DoubleCount = 0 }
let move = Functions.playTurn rollDice pickCard ignore startingPosition #C

#A Always roll 3, 4
#B Always pick card 5
#C Play from Go using these functions
```

Note that if you can't decouple yourself from `Random`, you can also achieve deterministic random behaviour by passing in a "seed" value when creating the `Random` object. Again, observe that we have a "pure" (or mostly pure!) F# domain using specific F# types etc., but have then wrapped it up with a simplified façade of a class with a simple event record to make it more "C# friendly".

### **Quick Check**

4. Why are expressions useful from a development and testing point-of-view?
5. Are there any restrictions on exposing C#-compatible events from F#?
6. Is it easier to move from expression to statement-based code or vice-versa?Summary

In this lesson, we looked at an example of how to expose F# to C# within the context of a slightly larger application.

- We saw how to work with *dual* domains – one F# for internal modelling, and a simplified one for consumption by e.g. C#
- We looked at a hybrid application that uses WPF / MVVM in C# with an F# engine.
- We looked at separating deterministic behaviour from non-deterministic functions such as `System.Random`.

### **Try This**

Try enhancing the Monopoly application by recording the cost of landing on each property, and adding that to the state of the application. Alternatively, add support for saving the state of the game to and from disk – and finally, if you're a WPF expert, look at how you could port the application from C# to F# using the `FsXaml` F# library.

### **Quick Check Answers**

1. Basic types, but also features such as Discriminated Unions and custom records.

2. Richer domain allows for better modelling internally, whilst simple external domain can make life easier for consumers coming from simpler type systems.
3. Systems where code-generation is essential or very important; frameworks or libraries that are inherently mutable or designed for C#.
4. They enable repeatable results on functions in a deterministic manner.
5. You cannot expose C#-compatible events from an F# module.
6. It's easy to move from expressions to statements, but not the other way around.

# 29

## *Capstone 5*

**Now that we've finished this unit, let's wrap up by applying these lessons back to our Bank Account application that we've been working on. We'll see how to: -**

- Integrate our existing F# codebase with a C# WPF application
- Use third-party NuGet libraries within our F# codebase
- Observe how F# domains resolve up to C# in a real solution

### **29.1 Defining the problem**

The objective of this capstone will be to plug a C# Windows Presentation Foundation (WPF) GUI on top of the existing F# codebase – essentially, replacing the console program runner with an event-driven UI. The system will provide the same withdraw and deposit commands as the console, as well as providing an updating transaction history.

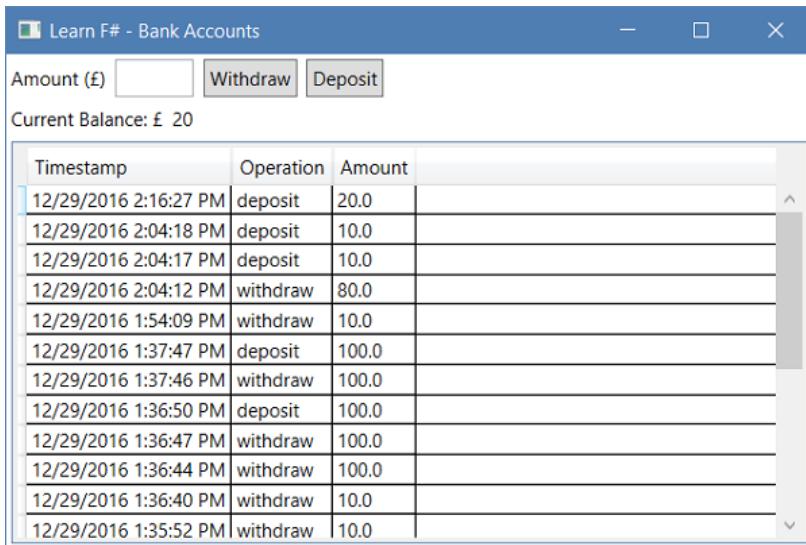


Figure 29.1 – Sample WPF GUI for the Bank Accounts application

### I don't know WPF!

As you may not be familiar with WPF (or even GUI programming at all), you'll find a pre-built GUI solution here using the Model-View-ViewModel (MVVM) design pattern. If you're a WPF whizz, feel free to scrap the supplied solution and start again! However, the objective of this capstone *isn't* to get bogged down in the depths of WPF (and if you've done any WPF or Silverlight in the past, you'll know just how complex things can get...). Instead, we want to focus on the integration of the C# and F# worlds from a language point of view, as well as how the OO and FP *paradigms* work together.

#### 29.1.1 Solution overview

`src/lesson-29` has both a starting solution and a completed version in the sample-solution folder. There are two projects – the F# core application, and the WPF C# front-end, of which the latter depends on the former (if you're a fan of the dependency inversion principle, you'll know that this breaks that pattern – but for the purposes of this lesson, it's overkill to add it in).

I've also changed the F# project from a Console Application to a Class Library, although I've left the old `Program.fs` code in there for you to re-use if you wish – by the end of this capstone, you'll be able to remove that file completely.

### Can't you do WPF in F#?

You can definitely write WPF applications in F#. Admittedly, the out-of-the-box experience is a bit of a let-down, as Visual Studio won't do any of the code gen that you get with C# etc. However, there are two excellent libraries available

on NuGet that make using WPF in F# relatively pain-free. Firstly there's FsXaml (<https://github.com/fsprojects/FsXaml>), a library that removes the need for the code-behind code gen through a type provider (see Unit 7). Secondly, if you're a fan of the MVVM pattern, there's the FSharp.ViewModel project that allows you to quickly and easily create view models that support INotifyPropertyChanged and command bindings etc. (<https://github.com/fsprojects/FSharp.ViewModel>). I'm deliberately avoiding them in this lesson, but I use both libraries all the time and can definitely recommend looking into them to enable 100% F# solutions (if that's what you want).

## 29.2 Plugging in a third-party NuGet package

Let's start with a fairly lightweight change to our F# code. In the previous incarnation of our app, we wrote a simplistic serialization routine for persisting bank transactions to disk - let's update that to work with the Newtonsoft.Json library.

### Now you Try

1. Add a NuGet package reference to the Core F# project for Newtonsoft.Json. It shouldn't be too hard to find, as it's the #1 most popular package on NuGet.

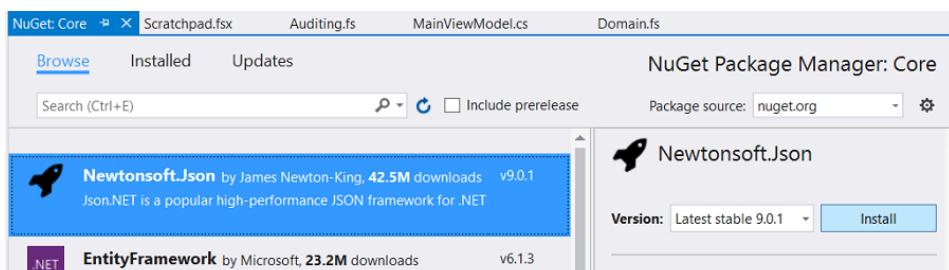


Figure 29.2 – Adding Newtonsoft.Json to an F# project.

2. Open `scratchpad.fsx` and add a `#r` reference to the **Newtonsoft.Json** package, or use VS to generate a load references script and `#load` that in instead (which is what Listing 29.1 uses).
3. Create a dummy Transaction record and serialize it to a string using the `Newtonsoft.Json.JsonConvert.SerializeObject` method. What's nice is that (as with any non-generic, single-argument method), you can use this method natively with the pipeline operator.
4. Observe the string that is emitted looks like plain, standard JSON.
5. You can use the `DeserializeObject` method to go back to an F# record again.

### Listing 29.1 – Using JSON .NET with F# records

```
#load @"Scripts/load-project-debug.fsx"
open Capstone5.Domain
open Newtonsoft.Json
```

```

open System

let txn =
 { Transaction.Amount = 100M
 Timestamp = DateTime.UtcNow
 Operation = "withdraw" }

let serialized = txn |> JsonConvert.SerializeObject #A
let deserialized = JsonConvert.DeserializeObject<Transaction>(serialized) #B

#A Serializing an F# record into a plain string
#B Deserializing a string back into an F# record

```

6. Now that you know how to use Json.NET with F#, you can replace the implementation of the serialization functions in the `Domain.Transaction` module with new ones that use Json.NET.

## 29.3 Connecting F# code to a WPF front-end

Now let's move onto the main event – connecting our F# bank account code to C# WPF.

### 29.3.1 Joining the dots

You'll find a module in the F# project, `Api.fs`. This module contains a set of functions which act as our façade over the top of the real bank account code base. In effect, this code should provide the same functionality that lives inside the `Program.fs`, except for anything to do with command handling and parsing. There are four functions you'll need to implement –

- **LoadAccount** – This should return a full account object based on the current state of the transactions for an owner – similar to what `tryLoadAccountFromDisk` does.
- **Deposit** – Essentially should perform the same logic as the `depositWithAudit` function in `program.fs`.
- **Withdraw** – As per `withdrawWithAudit`. You'll have to manually unwrap the `creditAccount` into an account in order to get at the `AccountId` and `Owner` fields.
- **LoadTransactionHistory** – should try to find any transactions on disk for the owner. Note that you only want to return the transactions, not the owner or account id – so whilst you can use `tryFindTransactionsOnDisk`, you'll need to be selective about what parts of the function output you pass back out.

You'll notice that, unlike the original F# code, all the functions here work off `Customer`, and not `Account`. The rationale behind this is that before performing any operation you should load the latest version of the account from disk rather than using an in-memory account that's provided to you. Doing this guarantees that the UI can't repeatedly pass you the same "version" of an account and withdraw funds using it. The easiest way to load in the account from disk is for the last three functions to simply call `LoadAccount` as the first thing that they do.

Also, be aware that both `LoadAccount` and `LoadTransactionHistory` should return absolute values rather than option types – so both of them will either have to pattern match or use `defaultArg` to return a default value if the account does not yet exist on disk.

Once you've implemented the API, you should test it out in a script to make sure it works as expected. Then, once you're happy it's all working, you should be in a position to run the app!

### 29.3.2 Consuming the API from C#

Let's briefly look at how we consume the API from C#. Again, we're not too fussed at the intricacies of WPF and MVVM, but how the F# API sits within C#.

Firstly, observe that we create an instance of a `Customer` record within C#, no problem – it appears as though it's a normal class, and we can use it as a property on the view model.

```
public MainViewModel()
{
 Owner = new Customer("isaac");
 ...
}
```

Figure 29.3 – Creating an F# record from C#

The API itself also appears to C# as though it was a normal static class, and even though the functions are in curried form in F#, they show as “regular” methods in C#:

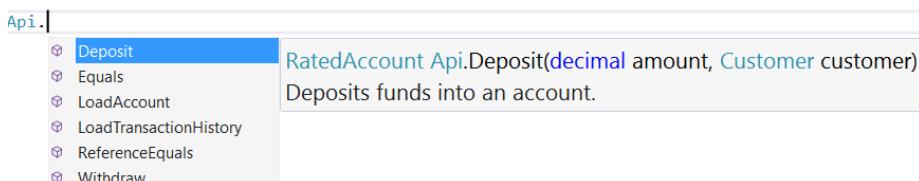


Figure 29.4 – Accessing an F# API from C#

So far, so good.

Notice that we also have intellisense comments from the F# triple-slash declarations shown here in C#. Let's now confirm we also have intellisense comments from the F# triple-slash declarations shown here in C#. Ensure that you have the **XML Documentation File** checked in the Build tab of the Properties pane of the F# project: -

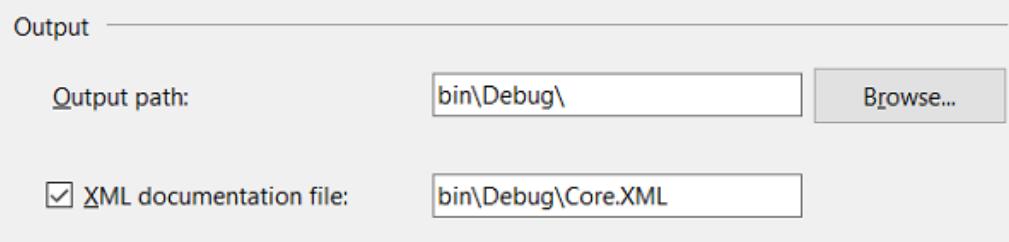


Figure 29.5 – Turning on XML comments for a .NET project in Visual Studio 2015

When it comes to displaying the `Transaction` records on screen, it *just works*. We create an `ObservableCollection` in the `ViewModel` (essentially, a Collection that also emits events for item changes that WPF can listen to and force rebindings of the UI etc.). This is then bound in XAML to a `CollectionViewSource` which in turn is connected to the `DataGridView` UI control. If this doesn't mean anything to you, don't worry – but if you *have* used WPF or Silverlight before, this should be pretty standard fare.

### 29.3.3 Using types as business rules in C#

We've spent some time looking at business rules in code, using types to distinguish between `Overdrawn` and `In Credit` bank accounts. Can we do the same in C#? Well, not *really*. C# will have a limited form of pattern matching that we'll be able to use to match an object against different types – but not much more.

However, in our case, we can use the properties that F# emits with compiled discriminated unions to simplify a rule for our GUI – whether or not to enable the `Withdraw` button. This button should only be available if the bank account is in `credit`. How do we do this?

One option would be to duplicate some logic in C# to check whether an account is "in credit" (alarm bells should be ringing as soon as I mentioned "duplication of logic"). Alternatively, we could add a function to our API that checks if the account is in credit or not, or lastly, we could simply always enable the button and simply leave the API to handle this (which it does anyway).

But we can go one better: remember that a `RatedAccount` is a discriminated union which is either `Overdrawn` or `InCredit`, and only when the account is of type `InCredit` do we want to enable the button. If you right-click `RatedAccount` and choose "Go to Definition" from C#, you'll see it's represented as a base class, with `Overdrawn` and `InCredit` as two subclasses: -

#### **Listing 29.2 – Viewing a discriminated union from C#-generated metadata**

```
public abstract class RatedAccount { #A
 public Boolean IsInCredit { get; } #B
 public Boolean IsOverdrawn { get; }
 public class InCredit : RatedAccount { #C
 public CreditAccount Item { get; }
 }
}
```

```

 public class Overdrawn : RatedAccount { #D
 public Account Item { get; }
 }
}

```

```

#A Base class
#B Runtime type check tags
#C InCredit subclass
#D Overdrawn subclass

```

And handily for us, F# also generates properties to allow us to easily check *which subclass* a rated account is. Let's look at the C# code for the Withdraw Command object, which is the code that is called whenever the user clicks on the button, as well as a parse function (to parse the textbox into an integer) and a function which says whether or not the button should be enabled.

### Commands in WPF

I'm deliberately skimming over the WPF side of things here, but it's worth understanding what a Command in WPF is. A standard command contains two methods: `Execute()` and `CanExecute()`. The former is called whenever it is bound to e.g. a Button and the user clicks the button. The latter is called by WPF to determine whether or not to "enable" the control – for example, to disable a Button so it can't be clicked.

### Listing 29.3 – Using F# types to enforce UI rules

```

WithdrawCommand = new Command<int>(
 amount => UpdateAccount(Api.Withdraw(amount, Owner)), #A
 TryParseInt, #B
 () => account.IsInCredit); #C

#A Command Execute method - withdraw funds from the account
#B Helper to convert from a string to an int
#C Command CanExecute method - checks if the button should be enabled or not

```

What's absolutely beautiful here is how we check if the button should be enabled or not – we simply check the `IsInCredit` tag that's generated by F#. In other words – if this account is of the type `InCredit`, we enable the Withdraw button! We're not running any code here to check bank balances etc. – we're simply using the *type system* to enforce a business rule! And because we refresh the command after every transaction occurs, this will automatically refresh as needed. Lovely.

## 29.4 Common Fields on Discriminated Unions

One thing we want to do in the app is display the *balance* of the account. Unfortunately, to do this, we'd normally have to first check whether the account is Overdrawn or InCredit, pull out the Account, and then get the Balance. In F# this is more or less bearable but in C# it'd be

awful. So, to work around this, we can add a *member property* to the `RatedAccount` which will do the work for us: -

#### **Listing 29.4 – Creating a member field on a discriminated union**

```
type RatedAccount =
 | InCredit of CreditAccount
 | Overdrawn of Account
 member this.Balance = #A
 match this with #B
 | InCredit (CreditAccount account) -> account.Balance
 | Overdrawn account -> account.Balance

#A Member declaration
#B Self-matching to access nested fields
```

If you add this code to the `RatedAccount` code in F# and recompile, you'll see that there is now a `Balance` property visible from C#, and you can now uncomment the line in the `UpdateAccount()` method so that the balance is correctly shown on the GUI (note – in the full solution, you could simplify the code above by using the `GetField` helper member that's already on there). We've not really seen member properties before – they're occasionally useful but generally not necessary for many F# workloads.

## **29.5 Polishing up F# APIs for consumers**

Here's a few more tasks that you can complete in order to make the experience even better from a C# perspective.

### **29.5.1 Encapsulation**

We've not bothered too much about this so far but when exposing an API to external consumers it's common to hide internal implementation details. Not only does this stop developers accidentally taking dependencies on things you might change in the future, it also makes things simpler for external developers – they can much more easily explore an API if only a limited subset is made visible. Currently, if we dot into `Capstone5`, you'll see all the F# modules and namespaces are available.

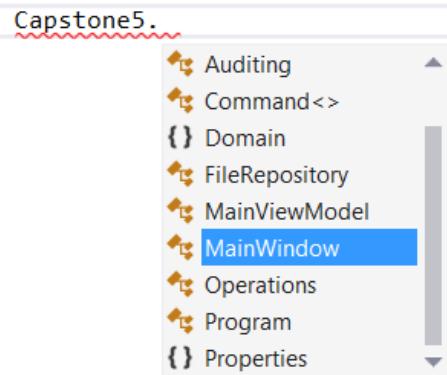


Figure 29.6 – Unnecessarily showing internal implementation modules to consumers

You can easily rectify this by placing the `internal` or `private` modifier for any module that you don't want to show e.g. module `internal Capstone5.Operations`. Recompile and you'll see in C# that these modules no longer show up.

### 29.5.2 Naming conventions

Whilst in F# it's common to use camel casing for functions e.g. `classifyAccount`, in C# we use pascal casing e.g. `ClassifyAccount`. If you're exposing code publicly to C# callers, make the effort to adhere to this naming convention (alternatively, decorate the function with the `[<CompiledName>]` attribute to change the function name post-compile!).

### 29.5.3 Explicit Naming

In F# we occasionally omit argument or field names from types – such as fields on a case on a discriminated union, or when we compose functions together – this doesn't render so well in C#. Unnamed DU fields will be rendered as `Item1`, `Item2` etc., whilst unnamed function arguments will be named `arg1`, `arg2`. In such cases, try to explicitly name function arguments.

## 29.6 Working with pure functions in a mutable world

This final section briefly discusses issues to be aware of when mixing OO and GUI worlds with the FP world. How do we mix stateless functions with mutable data on a GUI? The answer is ultimately more complex – and subjective – than one can really discuss completely here, but here in a nutshell are my thoughts.

Firstly, I would recommend by default trying to work with a stateless F# layer, and thread state *between* the GUI and back-end layers from one call to the next. In other words, each call to the F# layer takes in some state, and returns a *new* state. In our project, this doesn't really

hold true, but in effect we're generating new versions of the Account and Transaction History after every action. In other words, instead of mutating an stateful Account after each action, we exchange the current Account with a new Account after every API call.

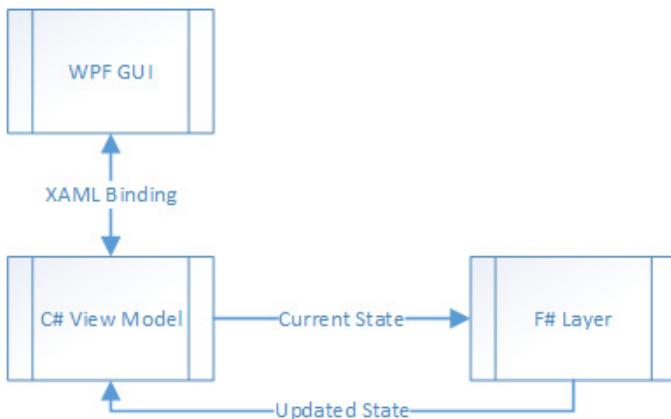


Figure 29.7 – Mixing mutable and immutable worlds between C# and F# with WPF.

On the other hand, there's obviously a performance cost associated with rebinding *everything* to a GUI on every call. So, you'll need to be cautious about this sort of approach if you're working with a GUI that's sending thousands or millions or events a second to a stateless F# layer. In this case, you might, for example, design your F# API to only return back *changes* to state from the previous state, and then manually apply those changes to the GUI.

There's no hard and fast rule – what I tend to believe is to do the *simplest* thing first and only optimise if there's good reason to.

## 29.7 Summary

Hopefully that wasn't too bad! You've now spent some time first hand trying to hook up a C# WPF application to an F# code-base, as well as used some NuGet packages into the mix – showing that F# *can* interoperate with the rest of the .NET ecosystem easily and effectively.

# Unit 7

## *Working with Data*

Working with data is (in my opinion!) one of the most exciting sections of this book. It focusses on working with external data sources such as JSON, CSV, SQL etc. whilst narrowing the “impedance mismatch” between data sources and writing code in a way you’ll have probably never seen before, through *type providers*. If you’ve been working with C# for a reasonable amount of time, you’ll remember the “aha” moment you got when you first saw LINQ. Type Providers provide a similar “shift” in mindset of how you work with code and data, only magnified by 100 times. When combined with the REPL and scripts, F# opens up entirely new opportunities for ad-hoc data processing and analytics.

Remember - free your mind!

# 30

## *Introducing Type Providers*

Welcome to the world of data! The first lesson of this unit will: -

- Gently introduce us to what type providers are
- Get us up to speed with the most popular type provider, **FSharp.Data**.
- After this lesson, you'll be able to work with external data sources in a number of formats more quickly and easily than you'll have ever done before in .NET – guaranteed!

### 30.1 What are Type Providers?

Type Providers are a language feature first introduced in F#3.0: -

An F# type provider is a component that provides types, properties, and methods for use in your program. Type providers are a significant part of F# 3.0 support for information-rich programming.

<https://docs.microsoft.com/en-us/dotnet/articles/fsharp/tutorials/type-providers/index>

At first glance, this sounds a bit fluffy – we already know what types, properties and methods are. And what does “information-rich programming” mean? Well, the short answer is to think of Type Providers as T4 Templates on steroids – that is, a form of code generation, but one that lives *inside* the F# compiler. Confused? Read on.

#### 30.1.1 Understanding Type Providers

Let's look at a somewhat holistic view of type providers first, before diving in and working with one to actually see what the fuss is all about. You might already be familiar with the notion of a compiler that parses C# (or F#) code and builds IL from which we can run applications, and if you've ever used Entity Framework (particularly the earlier versions) - or old-school SOAP web services in Visual Studio - you'll be familiar with the idea of code generation tools such as

T4 Templates or the like. These are tools which can generate C# code from another language or data source.



Figure 30.1 – Entity Framework Database-First code generation process

In this example, Entity Framework has a tool which can read a SQL database schema and generate an .edmx file – an XML representation of a database. From here, a T4 template is used to generate C# classes which map back to the SQL database.

Ultimately, T4 Templates and the like, whilst useful, are somewhat awkward to use. For example, you need to attach them into the build system to get them up and running, and they use a custom markup language with C# embedded in them – they're not great to work with or distribute.

At their most basic, Type Providers are themselves just F# assemblies (that anyone can write) that can be plugged into the F# compiler – and can then be used at *edit-time* to generate entire type systems for you to work with *as you type*. In a sense, Type Providers serve a similar purpose to T4 Templates, except they are much more powerful, more extensible, more lightweight to use, and are extremely flexible – they can be used with what I call “live” data sources, as well as offering a gateway not just to data sources but also to other *programming languages*.

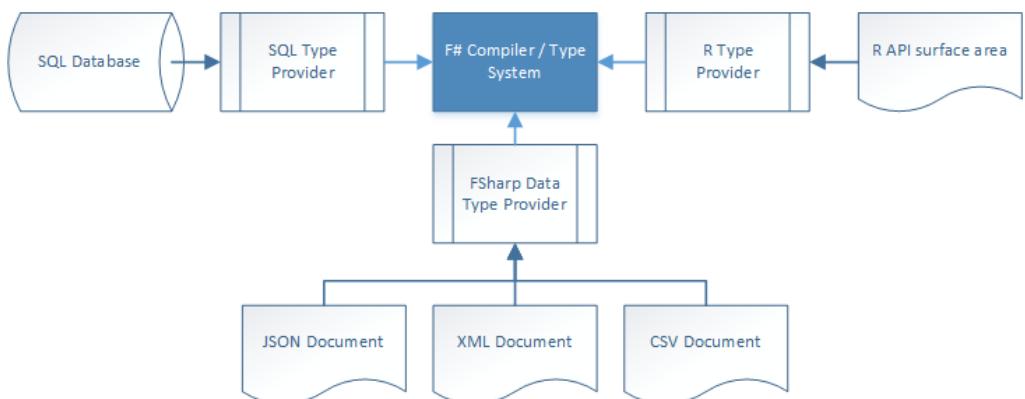


Figure 30.2 – A set of F# Type Providers with supported data sources

Unlike T4 Templates, Type Providers can affect type systems without re-building the project, since they run in the background *as you write code*. There are dozens, if not hundreds of type providers out there, from working with simple flat files such as CSV, to SQL, to cloud-based data storage repositories such as Microsoft Azure Storage or Amazon Web Services S3. The term “information-rich” programming refers to the concept of bringing disparate data sources *into* the F# programming language in an extensible way.

Don’t worry if that sounds a little confusing – we’ll take a look at our first type provider in just a second.

### Quick Check

1. What is a Type Provider?
2. How do type providers differ from T4 templates?
3. Is the number of type providers fixed?

## 30.2 Working with our first Type Provider

Let’s look at a simple example of a data access challenge not unlike what we looked at in Lesson 13 – working with some soccer results, except rather than work with an in-memory dataset, we’ll work with a larger, external data source – a CSV file, located in `learnfsharp/data/FootballResults.csv`. You need to answer the following question: which three teams won *at home* the most over the whole season.

### 30.2.1 Working with CSV files today

Let’s first think about the typical process that you might use to answer this question: -

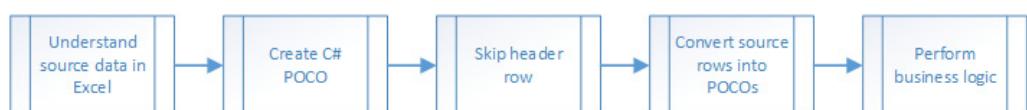


Figure 30.3 – Steps to parse a CSV file in order to perform a calculation on it.

Before we can even begin to perform the calculation, we first need to *understand* the data. This normally means looking at the source CSV file in Excel or similar, and then designing a C# type to “match” the data in the CSV. Then, we do all of the usual boilerplate parsing – opening a handle to the file, skipping the header row, splitting on commas, pulling out the correct columns and parsing into the correct data types etc. Only after doing *all* that can you actually start to work with the data and produce some actual value. Most likely, you’ll use a console application to get the results, too. This process more like typical software engineering – not a great fit when we want to explore some data quickly and easily.

### 30.2.2 Introducing FSharp.Data

We could quite happily do the above in F#; at least using the REPL affords us a more “exploratory” way of development. However, it wouldn’t remove the whole boilerplate element of parsing the file - and this is where our first Type Provider comes in – FSharp.Data.

FSharp.Data is an open source, freely distributable NuGet package which is designed to provide generated types when working with data in either CSV, JSON or XML formats. Let’s try it out with our CSV file.

---

#### Scripts for the win

At this point, I’m going to advise you move away from heavyweight solutions and start to work exclusively with standalone scripts – this fits much better with what we’re going to be doing. You’ll notice in the `learnfsharp` code repository a `build.cmd` file. Run it – it uses Paket to download a number of NuGet packages into the `packages` folder, from which you can reference directly from your scripts. This means we don’t need a project or solution to start coding – we can just create scripts and jump straight in. I’d recommend creating your scripts in the `src/code-listings/` folder (or another folder at the same level e.g. `src/learning/`) so that the package references shown in the listings here work without needing changes.

---

#### Now you try

Let’s try our first experiment with a type provider – the CSV type provider in FSharp.Data.

1. Create a new standalone script in Visual Studio using **File -> New**. You don’t need a solution here – remember that a script can work standalone.
2. Save the newly created file into an appropriate location as described in “Scripts for the win”.
3. Enter the following code: -

---

#### Listing 30.1 – Working with CSV files using FSharp.Data

---

```
#r @"..\..\packages\FSharp.Data\lib\net40\FSharp.Data.dll" #A
open FSharp.Data
type Football = CsvProvider<"..\..\data\FootballResults.csv"> #B
let data = Football.GetSample().Rows |> Seq.toArray #C

#A Referencing the FSharp.Data assembly
#B Connecting to the CSV file to provide types based on the supplied file
#C Loading in all data from the supplied CSV file
```

That’s it. You’ve now parsed the data, converted it into a type that you can consume from F# and loaded it into memory. Don’t believe me? Check this out: -

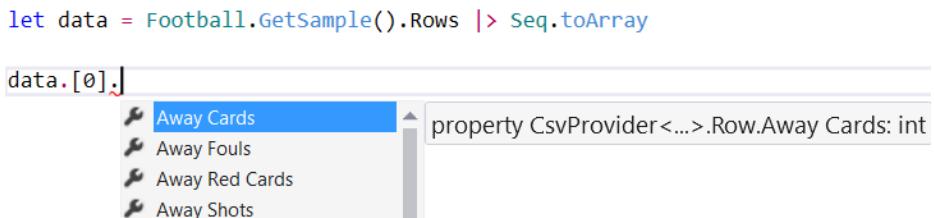


Figure 30.4 – Accessing a provided type from FSharp.Data.

You now have full intellisense to the dataset – that's it! You don't have to manually parse the data set – that's been done for you. You also don't need to "figure out" the types – the type provider will scan through the first few rows and infer the types based on the contents of the file! In effect, this means that rather than using a tool such as Excel to "understand" the data, you can now begin to use F# as a tool to both understand *and* explore your data.

### Backtick members

You'll see from the screenshot above, as well as from the code when you try it out yourself, that the fields listed have spaces in them! It turns out that this isn't actually a type provider feature, but one that's available throughout F# called *backtick members*. Just place a double backtick (` `) at the beginning and end of the member definition and you can put spaces, numbers or other characters in the member definition. Note that Visual Studio doesn't correctly provide intellisense for these in all cases e.g. let-bound members on modules, but it works fine on classes and records. We'll see some interesting uses for this when dealing with Unit Testing.

Whilst we're at it, we'll also bring down an easy-to-use F#-friendly charting library, XPlot. This library gives us access to charts available in Google Charts as well as Plotly. We'll use the Google Charts API here, which means adding dependencies to `XPlot.GoogleCharts` (which also brings down the `Google.DataTable.Net.Wrapper` package).

4. Add references to both the `XPlot.GoogleCharts` and `Google.DataTable.Net.Wrapper` assemblies. If you're using standalone scripts, both packages will be in the `packages` folder after running `build.cmd` – just use `#r` to reference the assembly inside one of the `lib/net` folders.
5. Open up the `XPlot.GoogleCharts` namespace.
6. Execute the following code to calculate the result and plot them as a chart.

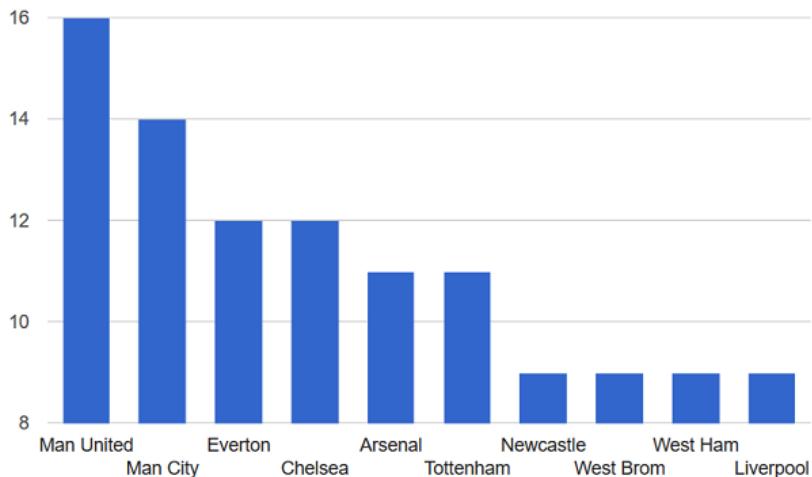


Figure 30.5 – Visualising data sourced from the CSV type provider

### Listing 30.2 – Charting the top ten teams for home wins

```

data
|> Seq.filter(fun row ->
 row.``Full Time Home Goals`` > row.``Full Time Away Goals``)
|> Seq.countBy(fun row -> row.``Home Team``) #A
|> Seq.sortByDescending snd
|> Seq.take 10
|> Chart.Column #B
|> Chart.Show #C

#A countBy generates a sequence of tuples (team vs number of wins)
#B Converting the sequence of tuples into an XPlot Column Chart
#C Showing the chart in a browser window

```

In just a few lines of code, we were able to open up a CSV file we've never seen, explored the schema of it, performed some operation on it and then charted it in less than 20 lines of code – not bad! This ability to rapidly work with and explore datasets that we've not even seen before, whilst still allowing us to interact with the full breadth of .NET libraries that are out there gives F# unparalleled abilities for bringing in disparate data sources to full-blown applications.

### Type Erasure

The vast majority of the type providers fall into the category of *erasing* type providers. The upshot of this is that the types generated by the provider exist only at *compile* time. At runtime, the types are *erased* and usually compile down to plain objects; if you try to use reflection over them, you won't see the fields that you get in the code editor.

One of the downsides is that this makes them extremely difficult (if not impossible) to work with in C#. On the flip side, they are extremely efficient – you can use erasing type providers to create type systems with *thousands* of types without any runtime overhead, since at runtime they’re just of type `Object`.

Generative type providers allow for run-time reflection, but are much less commonly used (and from what I understand, much harder to develop).

### 30.2.3 Inferring types and schemas

One of the biggest differences in terms of mindset when working with type providers is the realisation that the type system is *driven* by an external data source, and this schema may be *inferred* – as we saw with the CSV provider. Let’s see a quick example of how this can affect your development process.

1. In your script, change the data source for the `CSVProvider` from `FootballResults.csv` to `FootballResultsBad.csv`. This version of the CSV file has had the contents of the `Away Goals` column changed from numbers to strings.
2. You’ll immediately notice a compile-time error within your query:

```
> row.``Full Time Away Goals``)
 Home Team``)
This expression was expected to have type
 int
but here has type
 string
```

The screenshot shows a F# Interactive session. A tooltip is displayed over the code `row.``Full Time Away Goals``)`, indicating a type mismatch: the expression was expected to have type `int` but here has type `string`. The code `Home Team``)` is also visible.

Figure 30.6 – Changes in inferred schema causes compile-time errors.

3. This is because the type provider has inferred the types based on the contents of the sheet.

This point is crucial to grasp – not only within the context of a script, but also a full-blown application. Imagine you’re compiling your application off of a CSV file provided by your customer, and one day they provide you with a new version of the format. You can simply supply the new file to your code and instantly know where incompatibilities in your code are – any breaking changes will simply *not compile*. This sort of “instant” feedback is much, *much* quicker than either unit tests or run-time errors. Instead, we’re using the compiler and type system – the earliest possible stage – to show us exactly where code breaks. We’ll see more of this in the rest of this unit. Also – just in case you’re wondering, type providers support the ability to *redirect* from one file to another so that you can compile against one but run against another. We’ll deal with this in the coming lessons in this unit.

Lastly – note that when it comes to schema *inference*, some type providers work differently from others. For example, `FSharp.Data` allows you to manually override the schema by supplying a custom argument to the type provider. Others can use some form of schema

guidance from the source system – for example, SQL Server provides rich schema information from which a type provider does not need to infer types at all. We'll see more of this in the coming lessons, too.

### **Writing your own Type Providers?**

Sorry – but this book won't be covering how to write your own type providers! The truth is that they're not easy to develop – particularly testing them whilst you develop them – but there are a few decent resources worth looking at, such as the Starter Pack (<https://github.com/fsprojects/FSharp.TypeProviders.StarterPack>), as well as online video courses. If you're really interested in learning how to write your own, I would strongly advise you to first look at some of the simple demonstration ones out there to start with, or try to contribute to one of the many open source TPs out there – this is probably the best way to learn how they work.

### **Quick Check**

4. What are erased types?
5. What are backtick members?

## **30.3 Summary**

In this lesson, we took our first look at Type Providers. The remainder of this unit will introduce you to some other forms of type providers and give you an idea of just how far they can go.

- We saw what type providers are at a high level and learned about some of their uses.
- We explored the FSharp.Data package and saw how to work with CSV files.
- We also saw the XPlot library, a charting package that is designed to work well with F#

### **Try This**

Find any CSV file that you have on your PC. Try to parse it using the CSV type provider and perform some simple operations on it, such as list aggregation or similar. Or download a CSV from the internet containing some data and try with that!

Alternatively, try creating a more complex query to compare the top five teams that scored the most goals and display it in a pie chart.

### **Quick Check Answers**

1. A flexible code generation mechanism supported by the F# compiler.
2. Type Providers are supported within the F# compiler directly, and allow edit-time type generation – there's no code generation as with T4 templates.
3. No. Type Providers can be written, downloaded and added to your applications as separate, reusable components.
4. Erased types are types that exist at compile-time only; at runtime, they are “erased” to objects.

5. Backtick members are members on a type surrounded with double backticks, that allow you to enter spaces and characters that would normally be forbidden in the name.

# 31

## *Building Schemas from Live Data*

**Hopefully, you enjoyed using Type Providers in the previous lesson. This lesson will build on that one and explore the notion of building types from *live data sources* that exist outside of your codebase. We'll see**

- How we can create schemas from type providers from remote data sources
- Mixing local and remote data sources
- How to avoid issues when working with remote data sources

### 31.1 Working with JSON

#### 31.1.1 Live and Local files

In the previous lesson, we saw how to work with a type provider operating against a *local* data source – a CSV file placed on the local file system. As it turns out, many type providers also offer the ability to work against *remote* data sources – indeed, some providers are designed to work against remote data sources as the primary way of working.

These may be resources that you own, but they might just as easily be publicly available resources that you don't own, and aren't in control of. A good example of this can be seen when working with JSON data. You might use JSON as a local storage mechanism – for example, configuration files or simply local data storage. But JSON is also commonly used as a data transfer format for HTTP-enabled APIs – particularly RESTful APIs.

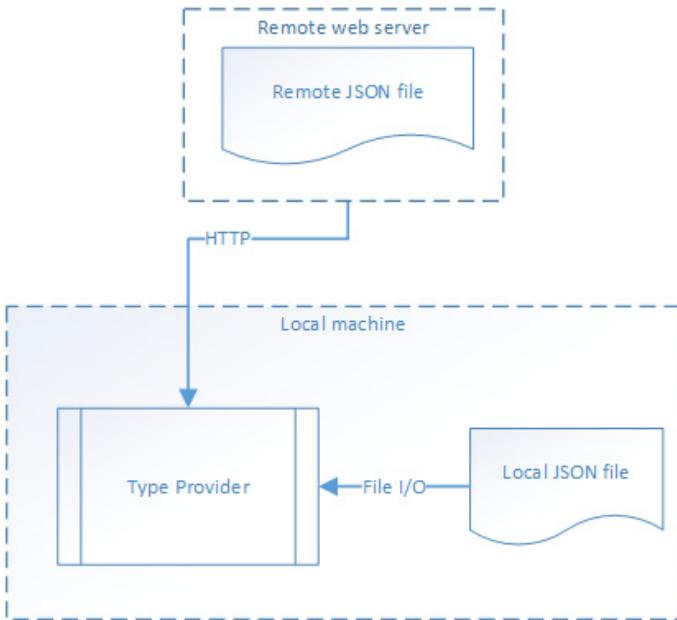


Figure 31.1 – Type Providers can operate over local and remote data sets

Let's see how we can access data from a *remote* JSON resource quickly and easily – in this case, using publicly available TV listings from the UK's BBC website.

### **Listing 31.1 – Opening a remote JSON data source**

```

#r "..\packages\FSharp.Data\lib\net40\FSharp.Data.dll" #A
open FSharp.Data
type TvListing =
 JsonProvider<"http://www.bbc.co.uk/programmes/genres/comedy/schedules/upcoming.json">
 #B
let tvListing = TvListing.GetSample() #C
let title = tvListing.Broadcasts.[0].Programme.DisplayTitles.Title

#A Referencing FSharp.Data
#B Creating the TvListing type based on a url
#C Creating an instance of the type provider

```

The type provider will pull down the resource over HTTP for us, parsing it as though it were a local file; as you “dot into” the `tvListings` value, you’ll see that you’re presented with a full type hierarchy representing the entire JSON document - the JSON provider automatically infers schema based off the full document content. There’s no need to manually download some data locally to start to use it in F# - you can point to a public, remote resource (as per Figure 31.1) and instantly start to work with it.

### DUs and Records in Type Providers

One of the (current) restrictions on Type Providers is that they can't generate discriminated unions. This does somewhat limit the ability of generated schemas – for example, if the JSON type provider sees different types of data across rows for the same field, it'll generate a type such as `StringOrDateTime`, which will have both optional `string` and `DateTime` properties, one of which will contain `Some` value. A more idiomatic way to achieve this in F# would be a discriminated union with two cases, `String` and `DateTime`.

### 31.1.2 Examples of live schema type providers

Here are some examples of type providers that can work off public data sources: -

- **JSON Type Provider** – provides a typed schema from JSON data sources
- **HTML Type Provider** – provides a typed schema from HTML documents
- **Swagger Type Provider** – provides a generated client for a Swagger-enabled HTTP endpoint, using Swagger metadata to create a strongly-typed model
- **Azure Storage Type Provider** – provides a client for blob / queue / table Storage assets
- **WSDL Type Provider** – provides a client for SOAP-based web services

#### **Now you try**

Let's now try to use a live schema from another data source via the HTML Type Provider. Given an HTML page, this type provider can find most lists and tables within it and return a strongly-typed dataset for each of them. It handles all the HTTP marshalling as well as type inference and parsing. Let's try it out on Wikipedia to visualize the number of films acted over time by Robert DeNiro. The HTML Type Provider is already included in the `FSharp.Data` package, so there's nothing more to download.

1. Within the script you already have opened, add references the to `Google.DataTable.Net.Wrapper` and `GoogleCharts` dlls – you can find them in the packages folder.

2. Open up the `XPlot.GoogleCharts` namespace.

3. Create a handle to a type based on a URL that contains an HTML document:

4. `type Films =`

```
HtmlProvider<"https://en.wikipedia.org/wiki/Robert_De_Niro_filmography">
```

5. Create an instance of `Films` called `deNiro` by calling `Films.GetSample()`

6. If you browse to the URL in your web browser, you'll see a number of tables in the document, such as **Film**, **Producing** and **Directing**. These tables map directly to type that are generated by the provider, and each table has a `Rows` property which exposes an array of data – each row is typed to represent a table row.

```

19 deNiro.Tables.Film.Rows
20 |> Array.countBy(fun row -> string row.)
property HtmlProvider<...>.Film.Row.Director: string
23
24
25

```

Director  
Film[1][2]  
Role  
Year

Figure 31.2 – Accessing a generated type based on an HTML table.

7. Write a query that will count the number of films per year and then chart it. Firstly, use the `Array.countBy` function to count films by Year – this will give you a sequence of tuples, which are used by the chart as X and Y values. You'll want to convert the `Year` property to a `string` though (rather than an `int`) so that the chart will create the correct number of elements in the x-axis.
8. Pipe the result into the `Chart.SteppedArea` function, and then into `Chart.Show`.

The output should look something like this – although, of course, as this is taken from live data, the results may be slightly different! If you couldn't write the query, don't worry – there's a complete example included in the code samples.

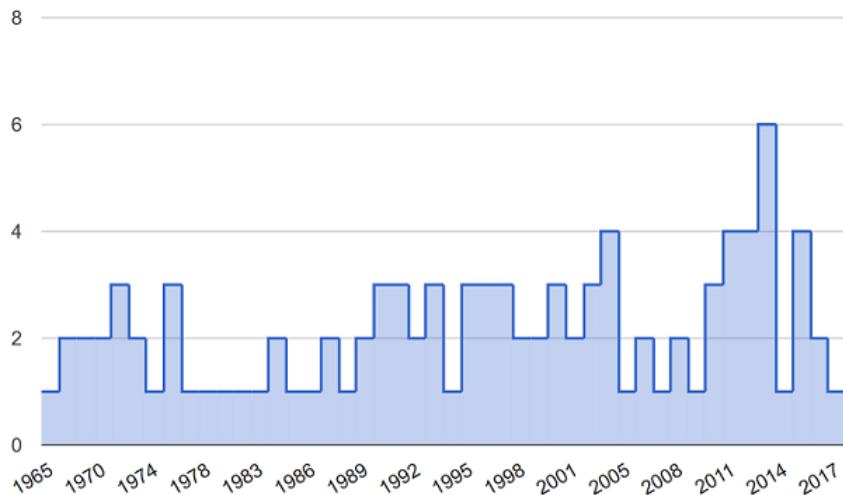


Figure 31.3 – Charting the results taken from a Wikipedia table.

In just a few lines of code, we've retrieved data from a remote HTML resource over HTTP, parsed the data into a strongly-typed object, before doing some simple analysis of it. More than that, bear in mind that although we're using a script here, this approach would work equally well over a fully-compiled console, web or windows application. Even better, if the source data changed schema, you'd know it instantly – the next time you open up Visual

Studio, you'd immediately have "red squiggles" under your code. You *wouldn't even need to run the application* and wait for a deserialization exception to know that there was a problem – the compiler does that for you. This is because every time you open the solution, the VS editor effectively uses the F# compiler to validate the code (and provide intellisense etc.) which in turn will connect to the data source. And lastly – you now know how to use Wikipedia itself as a database directly from within F#!

### **Quick Check**

1. Which Type Provider would you use to read data from a SOAP service?
2. Which package do the JSON and HTML Type Providers live in?

## **31.2 Problems with live schemas**

On the one hand, working directly with *remote* data to generate schema allows us with just a few lines of code to instantly start working with data – we don't have to download anything locally, or simulate an external service etc. However, working with remote schemas does raise a few interesting issues, primarily because a Type Provider essentially links the compilation of our code to some remote data source that we might not be in control of. Let's look at some of these issues: –

### **31.2.1 Large data sources**

One obvious problem is – what do you do if you need to work with a relatively large dataset e.g. 500MB. Do you need to load the entire dataset in order for the type provider to work? Won't that be really slow or memory intensive?

### **31.2.2 Inferred schemas**

Another problem is when inferring schemas. What if you have a CSV file with a field that's missing for the first 9,999 lines and is only populated in the final line? Does the type provider need to read through the entire dataset in order to infer that type? Or if there are many resources all of which follow the same schema – which one should you use?

### **31.2.3 Priced schemas**

Some data sources charge you for every request you make. A common example of this is when accessing data from one of the two major cloud vendors (Microsoft Azure and Amazon Web Services). Both vendors charge you (admittedly, tiny amounts) for reading data from cloud storage – even just *listing* the files that are stored in them! The Azure type provider provides the ability to "dot into" your cloud storage, so you can navigate through blobs of data directly from within Visual Studio. Unfortunately, doesn't this mean that you'll be paying every time you "dot into" a container?

### 31.2.4 Connectivity

One last issue is if you're working against a remote resource, you need to have the ability to connect to that resource in order to generate types. Let's assume you lose internet connectivity for a while – what happens then? The type provider won't be able to connect to the data source, which in turn will prevent the compiler from generating types from which you can develop against.

There are some solutions that are "built in" to many type providers that can help with these sorts of problems – for example, many type providers allow you to limit the number of lines that the data source runs against for schema inference – so you might want to use the first 50 lines to generate types for a CSV file, even though it has 1 million lines in it. Alternatively, some type providers such as the CSV Provider allow you to pass in a string that defines the headers e.g. "Name:string;Age,int;Dob,DateTime" etc. etc. However, there's a much better way of solving these sorts of problems once and for all, which we'll now look at.

---

#### **Unparameterised type providers**

Not all type providers take in type arguments – some, such as the COM or File System Type Providers work off the local machine and don't need any arguments. However, the majority of type providers support some form of parameterisation.

---

#### **Quick Check**

3. How do type providers help us to work with large data sets?
4. Why are can working with live data sources be an issue if you lose internet access?

## 31.3 Mixing Local and Remote data sets

What's important at this point is to realise that type providers really have two "phases" of operation: -

- A *compile* phase, which generates types based off some point-in-time schema (whether that's local or remote is irrelevant); this is the bit that gives you "red squiggles" if you mistype something etc. Think of this as the equivalent of the "custom tool" phase of T4 templates, if you've ever had to work with them before. The phase kicks in both at *edit-time* (i.e. when you're typing code into VS) and at *compile-time* – when you perform a build.
- A *runtime* phase, which uses the previously compiled types to work with some data that matches that schema. This might be the same data that was used to generate the schema, but it can also be some other data source.

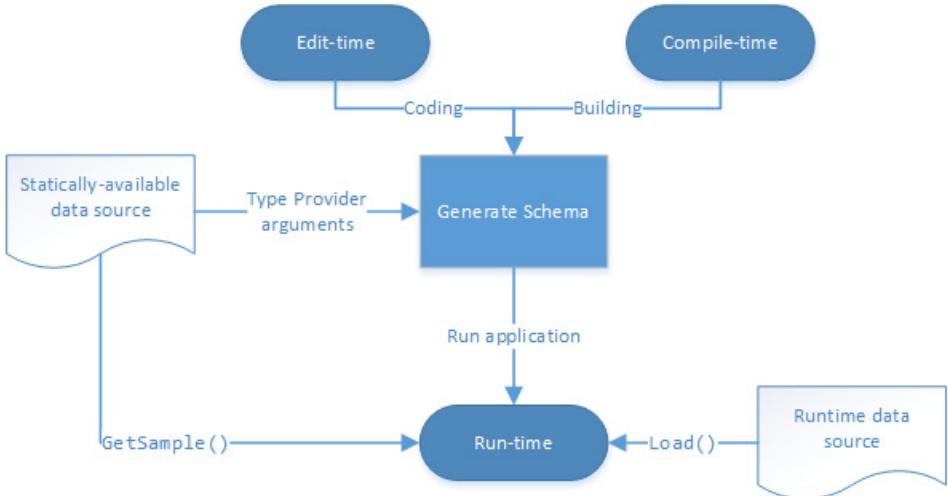


Figure 31.4 – Understanding the relationship between edit, compile and run-time with Type Providers

When either *coding* (e.g. editing in Visual Studio) or *building* (compiling code in VS or even MSBuild), the Type Provider will use the static data source to generate a schema and push that into the type system. Later on, at runtime, you might use another data source with the same schema against the type provider – which is what we’re going to look at right now!

### 31.3.1 Redirecting type providers to new data

You’ll have noticed with the type providers that we have used so far that we generate a type provider instance using the `GetSample()` function on the type. What this really means is: load in all data from the file that we used to generate the *schema* as the actual data source. But, there’s actually nothing to stop you *re-pointing* a type provider at a secondary data source that has the *same schema*. You’ll often do this to overcome the issues I mentioned earlier. For example: -

- You use a local CSV file with a small subset of data used for schema generation, but you point the CSV Provider to a larger file when working with data.
- You use a small, local JSON file which you create to “guide” the JSON type provider to infer types correctly, but you point to a real remote file when working with data.
- You use the local Azure storage emulator when developing with the Azure Storage Type Provider – thus saving you money and time – but point to a real storage account at runtime.

In other words, we use a local data file – typically one that is committed into source control – as part of our compile-time source code. This data file represents the *schema*, and is used by

the type provider for type generation, but is not necessarily the actual data that we'll work with.

Later in this unit, we'll talk more about working with Type Providers in full-blown applications, dealing with things like securing connection strings and the like – so relax if you're worrying about how you could use these within the context of e.g. CI servers or live applications.

### **Local and Live data mismatches**

There's one issue with working with "local" data sources for schema and separate sources for "live" data – you run the risk that the shape of the "live" data changes and you don't update your "local" schema file. In such a case, you'll be back to the world of runtime errors (which in itself is no worse than the world we currently live in). To be honest, that's a sacrifice you might have to make, and if you own the "live" data endpoint (e.g. an internally hosted REST services etc.) then you won't have to worry about the case that a schema changes without your knowledge.

### **Now you try**

Let's use the Nuget website as a data source to work with some statistics about nuget packages using the HTML Type Provider, which is able to parse the download figures tables on a given package. We'll use a *local copy* of a webpage to give us a schema – and enable us to develop offline - but then redirect it to a *live page* for retrieving the actual data.

1. Create an instance of the HTML Type Provider which points to `sample-package.html` in the `data` folder (if you use a relative path, watch out to make sure it's correct). If you open this, you'll see it's a sample of the NuGet package details page.
2. Using the `GetSample()` method, you can interrogate the `Tables` property to discover that there's a `Version History` property, which has members that reflect the equivalent table located in the HTML: -

## Version History

| Version                           | Downloads     | Last updated           |
|-----------------------------------|---------------|------------------------|
| <b>NUnit 3.5.0 (this version)</b> | <b>36,334</b> | <b>04 October 2016</b> |
| NUnit 3.4.1                       | 314,342       | 30 June 2016           |
| NUnit 3.4.0                       | 40,367        | 25 June 2016           |
| NUnit 3.2.1                       | 268,165       | 19 April 2016          |

Figure 31.5 – A Version History table from the NuGet website

3. Now, instead of using `GetSample()`, use the `Load()` method to load in some data from a live URI. This takes in a string URI for the "real" data.

```
let nunit = Package.Load "https://www.nuget.org/packages/nunit"
```

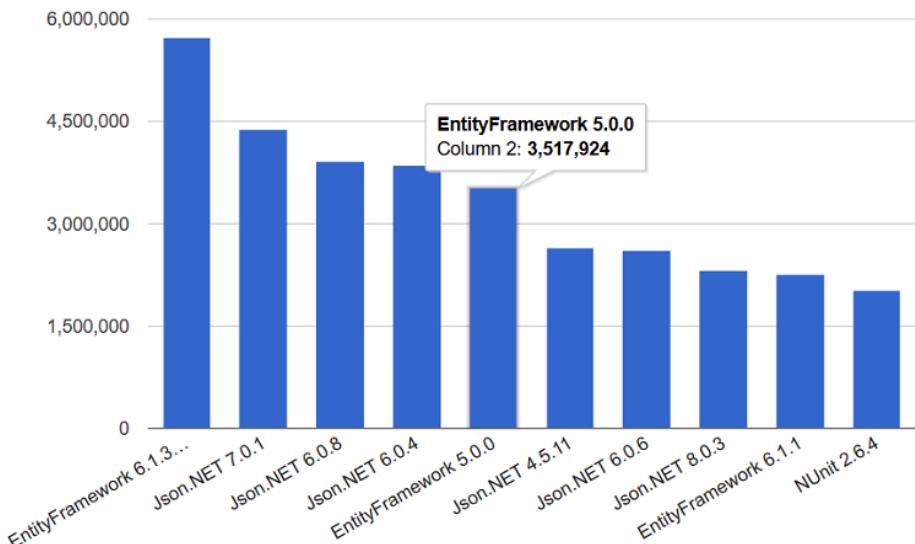
4. Repeat the process to download package statistics for the **Entity Framework** and **Newtonsoft.JSON** packages. The URI is the same as above, but use `entityframework` or `newtonsoft.json` in place of `nunit`.
5. Now that we have package statistics for all three packages, let's find the most popular *specific* versions of all of three packages combined.
6. Retrieve the `Version History` rows for all three packages and combine them into a single sequence; use `Seq.collect` to combine a sequence-of-sequences into a flattened sequence. Notice that this is the exact same code we would use with "normal" records or values – we can use provided types in exactly the same way.

### **Listing 31.2 – Merging sequences of provided values**

```
[entityFramework; nunit; newtonsoftJson] #A
|> Seq.collect(fun package -> package.Tables.``Version History``.Rows) #B

#A Creating a list of package statistics values
#B Merging all rows from each package into a single sequence
```

7. Sort this combined sequence in descending order using the `Downloads` property.
8. Take the top 10 rows.
9. Map these rows into tuples of `Version` and `Downloads`.
10. Create a `Chart.Column` which is then piped into `Chart.Show`.



**Figure 31.6 – Identifying the most popular NuGet release across multiple packages**

We used a *local* file for our schema (which in this case was itself taken directly from a web browser), but downloaded multiple datasets using the same type provider instance. You could equally do the same for JSON or CSV etc. Then, we merged the data into a simple shape from which we could graphically represent it.

### **Quick Check**

5. What does it mean to redirect a Type Provider at runtime?

## **31.4 Summary**

That's a wrap for this lesson. We worked with some more type providers and learned about the distinction between schema and data and the different "phases" of type providers.

- We worked with the HTML and JSON type providers
- We learned about some of the issues when working with schemas that point to "live" data sources
- We saw how to distinguish between schema and data by generating types from a static, local schema but point to live sources for data

### **Try This**

Using the HTML Type Provider and the Wikipedia page listing all music tracks by the band Dream Theater ([https://en.wikipedia.org/wiki/List\\_of\\_songs\\_recorded\\_by\\_Dream\\_Theater](https://en.wikipedia.org/wiki/List_of_songs_recorded_by_Dream_Theater)), calculate the year that they released the most albums. Visualise this as a line chart showing the number of tracks they released per year.

### **Quick Check Answers**

1. Which Type Provider would you use to read data from a SOAP service?
2. Which package do the JSON and HTML Type Providers live in?
3. How do type providers help us to work with large data sets?
4. Why are can working with live data sources be an issue if you lose internet access?
5. Redirecting a type provider is the act of using a different data source at run-time to the one used at compile time.

# 32

## *Working with SQL*

**In this lesson, we'll look at how we can use type providers with a database that you're probably familiar with – Microsoft SQL Server. We'll see: -**

- How to quickly and easily execute queries and commands against a SQL database
- Insert data quickly and easily
- Work with reference data in code

In my experience, .NET developers working with SQL server typically fall use frameworks that fall into one of two categories: -

- Using a full-blown object-relational mapper (ORM) tool, such as Entity Framework or NHibernate. These tools attempt to provide a layer of abstraction over the top of a relational DB by “mapping” relational tables into .NET hierarchical object models. They also typically provide features such as state tracking and conversion from `IQueryable` queries into (occasionally well-optimized) T-SQL.
- Using query / command patterns using either ADO .NET or a “Micro ORM” wrapper, such as Dapper or PetaPoco. These provide you with the ability to write direct SQL against a database, and automatically map the results against some DTO that you’ve created. They tend not to provide complex state tracking but are designed to be lightweight, performant and simple to use.

Both of these approaches have pros and cons – and I’m not about to be drawn into another debate about which one you should go with; there are plenty of resources and opinionated pieces online that you can read up on. Instead, I’ll show you how F# fits in with the data access story, particularly with type providers, and then let you make up your own mind.

### What's SQL?

Don't worry if you're not a expert – or even a beginner - in SQL. This lesson will cover relatively basic areas of SQL and will instead focus on the F# side of the story. Even if you've never used SQL before, you'll still find this a useful lesson – particularly as it shows just how type providers can effectively bridge the gap between a foreign language and F# - in this case, T-SQL, which is the query language for SQL.

#### 32.1.1 Creating a basic database

What we'll first do in this lesson is create a simple database with some pre-populated data that we'll then use in the remainder of this lesson to work with. The database – Adventure Works Light – is a simple order management database that we'll use to experiment with. Don't worry – you don't need to install the full-blown SQL Server (or even SQL Express) to do this! Instead, we'll use the lightweight LocalDB – a free, lightweight in-proc SQL database that is perfect for developing against.

#### Now you try

Let's deploy a sample database locally that we'll use for the remaining of the lesson.

1. In Visual Studio, navigate to the **SQL Server Object Explorer** (view menu).
2. Expand the SQL Server node.
3. You should see a node underneath which begins with `(localdb)`. I can't tell you which one it'll be, as the SQL team keep changing the format with each release – but it'll look something like `(localdb)\ProjectsV12` or `(localdb)\MSSQLLocalDB`.
4. If you don't see any `(localdb)` nodes, you probably will need to install **SQL Server Data Tools (SSDT)**. This is a lightweight installer that should be directly available from within **Extensions and Updates** in Visual Studio.
5. Now, we'll import a database onto the server. Right-click on the Databases node within the localdb server instance and choose **Publish Data Tier application**.

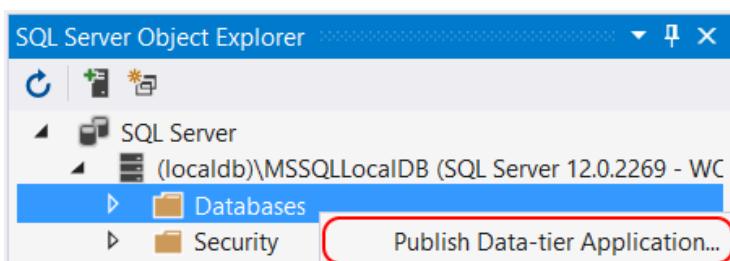


Figure 32.1 – Publishing a DACPAC to a SQL Server

6. From the dialog that appears, set the **File on Disk** as the `adventureworks-lt.dacpac` file from the data folder of the `learnfsharp` repository, and set the **Database Name** as `AdventureWorksLT`.

7. Hit **Publish** (ignore any warnings that may appear about overwriting data). Within a few seconds, Visual Studio will create a new database and populate it for us to test.

That's it - we now have a local database that we can experiment with. You can explore the database from within Visual Studio directly by right-clicking on any table and choosing **View Data**. Of course, we don't want to be relying on point-and-click GUIs, but using F# as a way to view the data directly without needing to leave the code editor!

## 32.2 Introducing the SQL Client Type Provider

Let's start by looking at the open-source and free to use SQL Client Type Provider, a data access layer designed specifically for MS SQL in the "Micro ORM" school of thought. As a type provider, it has several tricks up its sleeve that make it much, much more powerful than e.g. Dapper. In fact, since I started using this library, I've completely moved away from ORMs in general.

### 32.2.1 Querying data

#### **Now you try**

Let's start by connecting to the database that we just created and running a simple query to retrieve some data from it.

1. Open up a new script and add a reference to the assembly in the `FSharp.Data.SqlClient` package. This is located in the `packages` folder and you can manually #r the dll. If using a full solution, you can opt to download the package from NuGet and generate a references script through VFPT (see Lesson 22).
2. Open the `FSharp.Data` namespace.
3. Enter the following code: -

#### **Listing 32.1 – Querying a database with the SqlClient type provider**

```
let [<Literal>] Conn =
 "Server=(LocalDb)\MSSQLLocalDb;Database=AdventureWorksLT;Integrated Security=SSPI" #A
type GetCustomers =
 SqlCommandProvider<"SELECT * FROM SalesLT.Customer WHERE CompanyName = @CompanyName", Conn>
let customers =
 GetCustomers.Create(Conn).Execute("A Bike Store") |> Seq.toArray
let customer = customers.[0]

#A A standard SQL connection string
#B Creating a strongly-typed SQL command
#C Executing the command to return a dataset
```

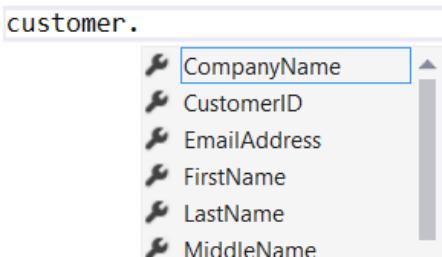
There's a few things happening here, so let's go through it one step at a time.

- o Firstly, we create a value to hold the connection string to SQL so that we can reuse

it (rather than passing it inline to the type provider). This value is also marked with the [`<Literal>`] attribute – effectively, marking it as a compile-time constant (also notice that the value is pascal cased – it's best practice to do this for literals) – which is needed when passing values as argument to type providers.

- Next, we generate a strongly-typed command, `GetCustomers`, based off the connection string and *an embedded SQL query*. I imagine that right now most likely you're recoiling in horror at this – don't worry, we'll talk about this more shortly.
- Lastly, we execute the query, again passing in the connection string. This time though, the connection string is used as the *run-time* data source, rather than for compile-time schema generation. The latest versions of the `SQLClient` do not allow you to implicitly re-use the static connection string, to protect you accidentally pointing to e.g. a development database at run-time – so you *have* to pass in a connection string when executing a query (even if, as in our case, it's the same one). In other words, it's as if there was no `GetSample()` function for the `CSV` provider, and we had to always use `Load()`. I'm not particularly fond of this decision, but I understand why it was made.

Now let's address the most obvious question – isn't embedding SQL directly into your application inherently "bad"? Leaving aside the point that you can actually pass in a path to a `.sql` file instead of embedding it in, actually – no, in the case of the `SQLClient`, it's not a bad thing to do at all. That's because the SQL we've entered in there isn't just a "magic string" – it's actually used by F# in a number of ways to provide us with *compile-time* safety. Firstly, the TP automatically generates a type for us based on the result set from SQL, for free – as we "dot into" `customer`, we're presented with intellisense and suggests properties for us: –



**Figure 32.2 – Working with data supplied from SQL by the `SqlClient` type provider**

Since SQL has an actual type system (with non-nullability across all types, unlike C#), there's also no need for trying to "infer" the type based on a sample of data – we can use the actual schema from SQL and cascade it into F# directly. So, for example, if you navigate to the `MiddleName` property, you'll see that this has been rendered as a

string option type, since it's a varchar null in SQL. And the type provider goes even further than this - as you're about to find out.

4. Write some code to print out the text <firstname> <lastname> works for <company name> for the customer value. You'll need to either use %A for CompanyName, or use defaultArg to safely unwrap the value from an option.
5. Now, change the SQL to read: SELECT TOP 10 FirstName, LastName FROM SalesLT.Customer. Observe that your code now no longer compiles, as the SQL only outputs FirstName and LastName. Also, notice that the error occurs in the exact place you would expect – where the CompanyName field is accessed.
6. Change the table name in the SQL query from Customer to Foo. Observe that the query itself now no longer compiles, with the error: Invalid object name 'SalesLT.Foo'.
7. Now change the query to read SELECT \* FROM SalesLT.Customer WHERE CompanyName = @CompanyName. This time, you'll see that the compiler breaks on the next line: the Execute() method now expects you to pass in the CompanyName as an argument as required by the query!

How does the type provider know which table names are valid in our database? How does it know to create a method taking in a value matching the parameter specified in the SQL? This is possible because the type provider validates the TSQL against the *SQL server itself* whilst generating types (i.e. at compile-time only – it doesn't happen at runtime) – and if the SQL is invalid, it automatically cascades the errors as a compiler error. Similarly, it uses this information in order to understand what arguments the SQL query expects, and then cascades them to the generated types.

### **SQL restrictions in the type provider**

The SQL you can use here can be far more complex than simple SELECT statements; you can use Joins, CTEs, Stored Procedures – even Table Valued Functions. However, there are a few SQL commands that the TP doesn't support - have a look on the official documentation at <http://fsprojects.github.io/FSharp.Data.SqlClient/>.

#### **32.2.2 Inserting data**

There are two ways you can work with inserting data with the TP – the first is by generating insert or update commands and executing them as above, but an alternative is to use a handy wrapper around good old .NET Data Tables – if you've been using .NET since before the days of LINQ, you'll remember these.

#### **Now you try**

Let's add some data to the Product Categories table. We'll use a second type provider included in the SqlClient package, called the `SqlProgrammabilityProvider`.

1. Create a new AdventureWorks type using the code type AdventureWorks = SqlProgrammabilityProvider<Conn>.
2. Now create an instance of the ProductCategory table type in F#. You can navigate to this via AdventureWorks.SalesLT.Tables; it's just a regular class.
3. In fact, this type is a standard DataTable, except it has a number of added provided members on it – such as a strongly typed AddRow() method.
4. Add three items to the table using the AddRow() method: -
  - o Mittens (Parent Category Id 3)
  - o Long Shorts (Parent Category Id 3)
  - o Wooly Hats (Parent Category Id 4)
5. As the parent category id is nullable on the database, you will have to wrap the id as an option e.g. Some 3.
6. You can then call the Update() method on the table. This does all the boilerplate of creating a DataAdapter and the appropriate insert command for you.
7. Back in **SQL Server Object Explorer**, check that the new items have been added by right-clicking the table and selecting **View Data**. You should see the extra rows added at the end of the table.

There's also a BulkInsert() method that is added to data tables – this allows you to insert data using SQL Bulk Copy functionality – which is highly performant and great for large one-off inserts of data. You can also use the data table for updates and deletes, or via T-SQL commands.

### 32.2.3 Working with Reference Data

One last area that the SqlClient addresses is working with Reference Data. Normally, in any data-driven application, you'll have static (or relatively stable) sets of "lookup data" – categories, country lists, regions etc. etc. which need to be referenced both in code and data. You'll normally have a C# enum (or perhaps a class with constant values in it) that matches a set of items scripted into a database. Obviously, you'll need to be careful to keep them "in sync" – for example, whenever a new item is added, you have to add it to the enum and also to the database with the same id.

The SqlClient package introduces the `SqlEnumProvider` to help us by automatically generating a class with values for all reference data values based on an arbitrary query. Here's an example of that for ProductCategories: -

#### **Listing 32.2 – Generating client-side reference data from a SQL table**

```
type Categories = SqlEnumProvider<"SELECT Name, ProductCategoryId FROM
 SalesLT.ProductCategory", Conn> #A
let woolyHats = Categories.``Wooly Hats`` #B
printfn "Wooly Hats has ID %d" woolyHats
```

#A Generating a Categories type for all product categories

#B Accessing the Wooly Hats integer ID.

What's really interesting is that we've now essentially bound our F# type system to the *data inside a remote system*. So, if you were to delete the Wooly Hats row from the database and then close and reopen the script in Visual Studio (to force the compiler to "re-generate" the provided types), you'd see that the code no longer compiles. In this way, it's impossible at compile-time for your code and data to become out of sync. Of course, if you deploy the application and *then* delete the data from the DB, you're out of luck – remember that type providers are a compile-time only feature.

### Quick Check

1. How does the SqlClient type provider remove the risk of "stringly-typed" queries?
2. In addition to manually create INSERT statements, how else does the SqlClient let you perform data insertion?

## 32.3 The SQL Provider

I'm aware that not everyone is a fan of the sort of "low level" SQL queries that the SqlClient provider library forces us down. Personally, I'm a big fan of it as the sort of stateless model is a good match with FP practices, is a good fit for many applications, and it completely avoids many of the anti-patterns that you can end up with with large ORMs. Nonetheless, there's a fantastic type provider called the SqlProvider (<http://fsprojects.github.io/SQLProvider/>) which offers an alternative way to work with SQL databases. I should also point out at this point that the SqlProvider isn't bound to SQL Server only – it also works with Oracle, SQLite, Postgres, MySQL and many ODBC data sources. Nonetheless, let's see how we'd achieve the same sort of functionality as with the SqlClient.

### 32.3.1 Querying data

As an ORM, SqlProvider supports the IQueryble pattern. Like many IQueryble providers, it's not 100% complete – but it's powerful enough to do the things you'll need on a day-to-day basis – and a lot more. Here's an example of an F# *query expression* to get the first 10 customers: –

#### **Listing 32.3 – Querying data using the SqlProvider library**

```
#r "..\..\packages\SQLProvider\lib\FSharp.Data.SqlProvider.dll"
open FSharp.Data.Sql

type AdventureWorks = SqlDataProvider<ConnectionString = "<connection string goes here>",
 UseOptionTypes = true> #A

let context = AdventureWorks.GetDataContext() #B

let customers =
 query {
```

```

 for customer in context.SalesLt.Customer do
 take 10
 } |> Seq.toArray #C

let customers = customers.[0]

#A Creating an AdventureWorks type using the SqlDataProvider
#B Getting a handle to a sessionised data context
#C Writing a query against the Customer table

```

Again, let's quickly review this. Having referenced the `SqlProvider` assembly, we create a provided type for the database. Notice that this type provider takes in some further parameters – in this case, whether or not to generate option types for nullable columns (otherwise, a default value will be generated instead). Next, we create a handle to a `DataContext`. If you've used Entity Framework before, you'll know what a data context is – essentially a stateful handle to a database within which client-side operations can be tracked and then a set of changes sent back to the database. Lastly, we write a *query expression* to take the first 10 customers from the database.

## Query Expressions

F# *Query Expressions* are another form of *computation expression* – similar to what we saw earlier with `seq { }` (and will see again in the next unit with `async { }`). Essentially, a query expression can be thought of as equivalent to a LINQ query in C#. They operate in a similar way – an expression tree is formed which is then parsed by a specific provider to convert the tree into another language (in this case, T-SQL). Unlike the LINQ query syntax, which is fairly limited, F# query expressions have a large set of operations, such as `sortBy`, `exists`, `contains`, `skip` etc. – see <https://docs.microsoft.com/en-us/dotnet/articles/fsharp/language-reference/query-expressions> for the full list. Query expressions can be used in F# over any `IQueryable` data source, so you can use them anywhere you would write a LINQ query in C#.

You're not restricted to using the table entities either – you can project results to your own custom records or use tuples (remember that F# does not have anonymous types).

### **Listing 32.4 – Projecting data within a more complex query**

```

query {
 for customer in context.SalesLt.Customer do
 where (customer.CompanyName = Some "Sharp Bikes") #A
 select (customer.FirstName, customer.LastName) #B
 distinct #C
}

#A A filter condition within a query expression
#B Projecting a set of tuples as the result
#C Selecting a distinct list of results

```

One of the nicest things about the `SQLProvider` is that, unlike the `SQLClient`, we get full intellisense on the tables and columns within it – this makes it excellent for exploration of a

database, because we don't have to leave F# to understand the database contents etc. – we get everything through intellisense.

### 32.3.2 Inserting data

Adding data to the database is simple – we simply create new entities through the data context, set properties and then save changes – basically the same pattern that one uses with Entity Framework.

#### **Listing 32.5 – Inserting new data**

```
let category = context.SalesLt.ProductCategory.Create() #A
category.ParentProductCategoryId <- Some 3 #B
category.Name <- "Scarf"
context.SubmitUpdates() #C

#A Creating a new entity attached to the ProductCategory table
#B Mutating properties on the entity
#C Calling SubmitUpdates to save the new data
```

It's worth running this yourself – observe that all entities track their own state and have a `_State` property on them. If you create a new entity, you'll see that its initial state is `Created`, but after calling `SubmitUpdates()`, its state changes to `Unchanged`. It's also interesting to note that unlike the `SQLClient` the `SqlProvider` uses a data context, which by its very nature is stateful – updates are performed by first loading the data from the database, mutating the records and then calling `SubmitChanges()`.

### 32.3.3 Working with Reference Data

Working with reference data with the SQL Provider is incredibly easy thanks to a feature called *Individuals*. Every table on the context has a property called `Individuals`, which will generate a list of properties that match the *rows in the database* – essentially the same as the `Enum` Provider. You also have sub-properties underneath that allow you to choose which column acts as the “text” property e.g. `As Name`, `As ModifiedDate` etc.

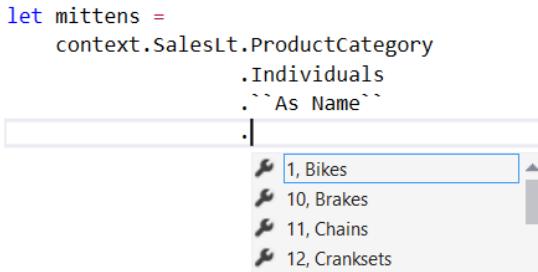


Figure 32.3 – Navigating through reference data using Individuals by Name

Again, like the SQLClient, the items in this list are based on the database contents – so if data is removed, you'll instantly know that your application is out of sync.

### What about Entity Framework?

I don't want to rule out EF as a data access technology for F# - it's simply that I don't feel it's as powerful or useful as either of the above type providers, both of which give you much stronger type checking than EF in different ways, and allow you to start exploring databases much more quickly – in just a few lines of code, as we've seen. Furthermore, to create entities that work with EF requires creating types that have mutable virtual properties and the like – essentially, the complete opposite to the kind of types that F# makes it easy to do. For these reasons, I'd suggest either using one of the above technologies (or similar) rather than EF. If you really must use EF, I recommend creating a C# layer for your entity model etc. and accessing that from F#.

### Quick Check

3. What is the key distinction between the SqlClient and SqlProvider?
4. Is the SqlProvider coupled to just MS SQL Server?

## 32.4 Summary

In this lesson, we saw how we can use two type providers to work with relational data stores, and how simple they can make our life. They fit especially well with the “exploratory” nature of scripts, and give us much stronger typing than we might be used to when working with SQL and .NET.

- We saw the SQLClient library, a lightweight wrapper on top of ADO .NET that gives us strongly-typed and validated SQL commands directly within F#
- We then saw the SQLProvider, an alternative type provider that uses an ORM-style data context model for exploring and working with SQL
- We learned about F# query expressions, another form of *computation expression* that allows us to model queries against data
- We saw how type providers can be used to bring *data* into the *type system* directly with

e.g. reference data and *individuals*.

### Try This

Connect to a database that you already have. Experiment with using both type providers against the data source. Also, look at reference data tables that you have in your application layer; see if you can replicate this using both type providers and compare this to the approach you've currently taken.

### Quick Check Answers

1. How does the SqlClient type provider remove the risk of “stringly-typed” queries?
2. In addition to manually create INSERT statements, how else does the SqlClient let you perform data insertion?
3. SqlClient is a low-level type provider in which you write your own SQL. The SqlProvider is an ORM that generates queries and commands based on query expressions.
4. No, the SqlProvider works with several other SQL databases.

# 33

## *Creating Type Provider-backed APIs*

In this unit, we've so far looked at various types providers; hopefully you now get the gist of how they typically operate as well as the sorts of features and pitfalls to be aware of, so that you are able to explore and try out other ones yourself. This lesson will look at how to quickly create APIs that are driven by type providers that other components can easily consume. You'll see:-

- How coupling to Type Providers can affect your application
- Creating APIs over type providers
- When to create decoupled APIs

Just so that we're on the same page here, let's just recap what I mean by a "type provider-backed API". So far, we've looked at using type providers in an *exploratory* mode – for example, doing some analysis on data within a single script. However, there's nothing to stop you integrating type providers within the context of a *standalone application* as well – be they console applications or web apps, whether 100% F# or hybrid language applications. There are, however, some things that it's worth being aware of before trying to integrate a type provider into an application.

At their most basic, type provider APIs are no different from standard data-oriented APIs that you create; these typically follow the Gateway, Facade or Repository design patterns, by providing a simple "layer" on top of a lower-level data access layer. Even if you don't know what those design patterns are, you've probably done this sort of thing a hundred times before with data access technologies such as Entity Framework or ADO .NET to provide a layer of abstraction between your application and the underlying e.g. SQL database.

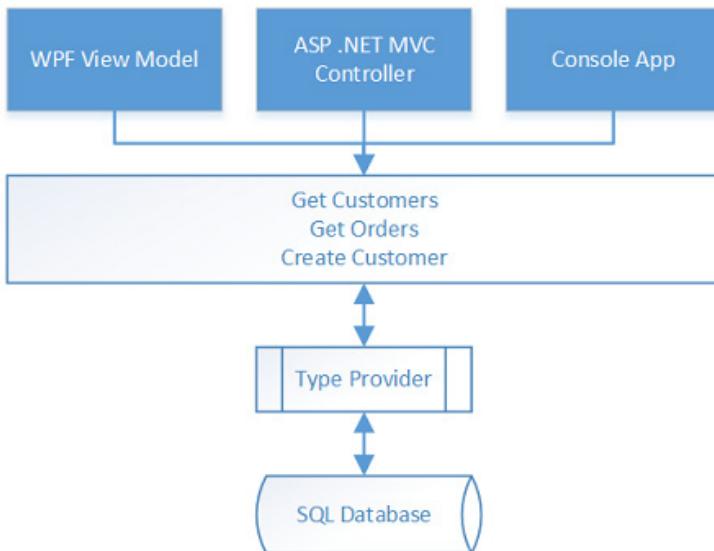


Figure 33.1 – A simple façade over a SQL database via a Type Provider as the data access layer.

### 33.1 Creating a tightly-coupled Type Provider API

Let's start by creating a simple API that follows on from a previous lesson – one that can provide statistics on NuGet packages. We'll create a simple API to do the following: -

- Retrieve the total number of downloads for any given package
- Retrieve details for a specific version of a NuGet package
- Retrieve details of the latest *stable* release of a NuGet package

#### 33.1.1 Building our first API

##### **Now you try**

Let's build our first API, which directly exposes data and types generated by a type provider.

1. Start by creating a new F# Console project and add the `FSharp.Data` nuget package.
2. Create a script file called `NuGet.fsx` in which we'll write API. We'll port this into an `fs` file later, but for now we'll stick with a script and the REPL whilst we're in "exploration" mode.
3. Reference `FSharp.Data` within the script either using `#reference` manually or a Power Tools-generated script and `#loading` it.
4. Open the `FSharp.Data` namespace and create an instance of the HTML Type Provider which points to the sample version history dataset that we used in Lesson 26.

5. Create a function, `getDownloadsForPackage`, which given a NuGet package name, will return the total number of downloads for the package name.
6. Your code should look something like this: -

#### **Listing 33.1 – Creating our first type provider-backed API function**

```
type Package = HtmlProvider< @"..\..\data\sample-package.html"> #A

let getDownloadsForPackage packageName =
let package = Package.Load(sprintf "https://www.nuget.org/packages/%s" packageName) #B
package.Tables.`Version History`.Rows
|> Seq.sumBy(fun p -> p.Downloads)

#A Create a static instance of the type provider
#B Creating a live url based on a function argument
```

That was pretty easy! The most important part is that we're dynamically building up the URL for the "live" HTML Provider endpoint based on the package name that's supplied, after which we perform a simple in-memory query. You'll notice that the function has a simple signature of `string -> decimal` – there's no clue for the caller that they're using a Type Provider.

7. Try calling this function from the script for a set of packages e.g. EntityFramework and Newtonsoft.Json.
8. Now let's try the next API function. Create a new function, `getDetailsForVersion`. It should download the package as before, except this time the query will be different – we'll try to find the row where the Version *contains* some text that will be provided as an argument. You can use `find` or `tryFind` in the `Seq` module for this (essentially the F# equivalents to LINQ's `First` and `FirstOrDefault`).
9. Notice that this function returns a strange-looking type called `HtmlProvider<...>.VersionHistory.Row` (and if you used `tryFind`, as a wrapped option). This is the *static name of provided type* that represents one of the version rows; we're exposing this type directly outside of our API. Now that's not necessarily a problem, but it *is* something we'll discuss again further on in this lesson.

#### **33.1.2 An exercise in refactoring**

At this point, let's take a quick diversion and do a little refactoring of the code to get some reuse across both functions, as they both do something very similar. A basic refactor might be to simply create a helper function that loads the package by name so that we can call it from both functions – something like this: -

#### **Listing 33.2 – Trying to gain code reuse across multiple functions**

```
let private getPackage packageName =
packageName |> sprintf "https://www.nuget.org/packages/%s" |> Package.Load #A
let getDetailsForVersion versionText packageName =
```

```

let package = getPackage packageName #B
package.Tables.`Version History``.Rows |> Seq.tryFind(fun p -> p.Version.Contains
versionText)

#A Creating a helper function to load package data
#B Using the helper function in a higher-level function

```

That's not bad, but we can do better. Here's a more tightly refactored version that makes use of function composition to reduce the size of our API functions to the bare essentials: -

### **Listing 33.3 – Further refactoring an API implementation**

```

let private getPackage = sprintf "https://www.nuget.org/packages/%s" >> Package.Load #A
let private getVersionsForPackage (package:Package) = package.Tables.`Version History``.Rows
 #B
let private loadPackageVersions = getPackage >> getVersionsForPackage #C

let getDownloadsForPackage = loadPackageVersions >> Seq.sumBy(fun p -> p.Downloads) #D
let getDetailsForVersion versionText = loadPackageVersions >> Seq.tryFind(fun p ->
 p.Version.Contains versionText)

#A Retrieves package data
#B Navigates to the Version History rows
#C Composes the first two functions together
#D Using the composed function at the API level

```

Let's go through this step by step. Firstly, we can simplify the `getPackage` function using *function composition*. Remember the `>>` operator? Well, we can use it here to eliminate the explicit `packageName` argument being passed in – in other words, `sprintf "...%s"` can itself be thought of as a function that takes in a string and returns another string – which we then “connect” as the input for the `Package.Load` function. And, since the input of `Package.Load` is a string, the signatures are compatible.

Next, we create a simple helper function `getVersionsForPackage` that navigates to the rows of the `VersionHistory.Rows` collection of a given package. We could have written this as an inline lambda, but I think that this is more readable.

Thirdly, we compose *those* two functions together to build an even bigger function, `loadPackageVersions`. Again, compare the inputs and outputs: -



Figure 33.2 – Composing functions together to build progressively more complex behaviours

We can compose `getPackage` and `getVersionsForPackage` together since the *output* of the first is the same type as the *input* of the second function. The composed function takes in the same type as the *input* of the first function, and returns the same type as the *output* of the last function. This just leaves the main two API functions, which we compose *again* (!) – this time using the appropriate query function that we wanted to use before. In this way, you can start to quickly build up tiny composable and reusable behaviours quickly and easily.

Now create the final function, `getDetailsForCurrentVersion`. This should do the same as the `getDetailsByVersion`, except that the text we're looking for is always the same - "(this version)". You should be able to create this function simply by calling the `getDetailsForCurrentVersion` and only supplying the first argument (the version text to search for). Whilst this mini-exercise we've just gone through doesn't have anything to do specifically with type providers, it's a good lesson in how to compose small functions together to build more complex ones that can be used as the basis for an API (or indeed any DSL).

### Quick Check

1. Can you build APIs from Type Providers?
2. What rule must be followed in order to compose two functions together?
3. Can we reference provided types statically in function signatures?

## 33.2 Creating a decoupled API

What we've done will work just fine for us. However, there are times when exposing provided types directly as an API won't be suitable – or even possible. In this case, you'll need to manually construct types – F# records and discriminated unions etc. – and then map from the provided types to these manually created types. In this case, we create a truly decoupled API from the type provider in the sense that we expose no types at all from the type provider to callers.

### 33.2.1 Reasons for not exposing provided types over an API

Here's some issues you might come across when working with type providers and reasons why you might want to decouple yourself from provided types within your application code-base: -

- Simple abstraction and decoupling – the business domain may not fit exactly with the data supplied by a type provider (particularly providers that do not allow projection of data), so you may need to map between the two.
- Remember that (at the time of writing) provided types *cannot* create records or discriminated unions. This limits the richness of the types that can be emitted from a type provider, which may in turn lead to writing extra code to "compensate" for this. It might be better to map to a richer domain earlier, which will simplify the rest of your code base.
- Provided types generally cannot be consumed outside of F#. So, if you have a hybrid language solution, you can forget about consuming an API such as the one above from

C#, because in two of the methods we expose *provided types* in our *public API*.

- Most type providers create types that are erased at runtime. This means that you can't reflect over them, and therefore any code that uses reflection or similar to generate outputs won't work – so you wouldn't be able to use the SQL Provider directly with a framework like Newtonsoft.JSON to create JSON from provided types, because at runtime there are no types to reflect over – everything's just a `System.Object`.

### Working with provided types at runtime

There are a few workaround to get around the “erasing types” runtime issue that type provider authors use. One is to create generative type providers, which emit “real” types at runtime that *can be reflected over*. There are pros and cons to this approach – some type providers are specifically designed to be erasing as they carry no runtime overhead in terms of types. The alternative is that some type providers expose a weakly-typed dictionary of key/value string pairs for the properties which *can be accessed* at runtime.

### 33.2.2 Enriching a domain using F# types

Let's create a slightly more expressive domain for our NuGet packages for our API. Here's a set of F# types that represents our domain more expressively than what we're getting back from the HTML Type Provider: -

#### Listing 33.4 – Creating a custom domain for NuGet package statistics

```
Open System
type PackageVersion = #A
| CurrentVersion
| Prerelease
| Old
type VersionDetails = #B
{ Version : Version
 Downloads : decimal
 PackageVersion : PackageVersion
 LastUpdated : DateTime }
type NuGetPackage = #C
{ PackageName : string
 Versions : VersionDetails list }
```

#A Classifier of package version  
#B Representation of a single package version  
#C Representation of an entire Package

This model has some nice properties compared to our original model. For example, we no longer have the package name repeated through every version – it's now only stored once at the “top” of the package. In addition, each version is now no longer just a string, but has a proper `System.Version` as well as a classification of whether this version is a pre-release, current or historical package. With this model, you could more easily reason about the versions in a package – so it would be easy (and much safer!) to find out what the current version of a package is, or to determine if there have ever been any beta versions etc.

### 33.2.3 Mapping between provided types and F# domains

Of course, we now need to write the code to map from our provided types to this rich domain! I'd encourage you to have a crack at doing this yourself – but I've also supplied a sample solution below (and, of course, in the source code sample repository).

#### **Now you try**

Let's try to create the code to map from the provider types to our strongly-typed F# domain.

1. Write a function, `parse`. This function should take in a single string representing the `Version` property of a nuget `VersionHistory` row, and return back a tuple of a `System.Version` object (based on the version in the supplied string) and the classification of `PackageVersion`. So, the string "Json.NET 8.0.3" should return (8.0.3, Old) whilst "9.0.2-beta1" should return (9.0.2, Prerelease).
2. You can split the string on both space and – to determine what sort of version it is, and then use pattern matching on the resultant array to identify what sort of string it is. Watch out when splitting strings because package names *themselves* can have spaces in them e.g. F# Data 2.2.3. You can either use standard Array indexing logic, or use pattern matching. You might struggle with the pattern matching, so you can refer to my suggested solution for that. There are two cases: -
  - o If the string ends in "(this version)", then the package name is all words except the last three words, which will be e.g. "2.2.3 (this version)".
  - o Otherwise, the package name will be all words except for just the last word, which will just be the version.
3. Write a function, `enrich`, which takes in a sequence of `VersionHistory.Row` objects (from the type provider) and returns a `NuGetPackage` (our new domain model).
4. To determine the package name, simply take the package name from the first row in the collection (again, you'll need to do some work to parse to just get out the name part). We'll assume that it's repeated across all rows so it's safe to use the first row.
5. You can create the `Version` list by creating a `VersionDetails` record for each row, copying across the `Downloads` and `LastUpdated` fields, and parsing the `Version` field.

Here's a sample solution for the `parse` function if you got stuck! The rest of the solution is provided in the source code samples.

#### **Listing 33.5 – Creating a custom domain for NuGet package statistics**

```
let parse (versionText:string) =
 let getVersionPart (version:string) isCurrent = #A
 match version.Split '-', isCurrent with
 | [| version; _ |], true
 | [| version |], true -> Version.Parse version, CurrentVersion
 | [| version; _ |], false -> Version.Parse version, Prerelease
 | [| version |], false -> Version.Parse version, Old
 | _ -> failwith "unknown version format"
```

```

let parts = versionText.Split ' ' |> Seq.toList |> List.rev #B
match parts with
| [] -> failwith "Must be at least two elements to a version"
| "version" :: "(this" :: version :: _ -> getVersionPart version true #C
| version :: _ -> getVersionPart version false

#A Inner function to parse version number from a string
#B Converting an array of strings to a reversed list
#C Matching on two cases of strings to parse version number and identify current / old version

```

This is more complex pattern matching than we've previously seen. Looking at the lowest pattern match (i.e. *not* the one in `getVersionPart`), we do two things: -

1. Firstly, we split the string on spaces, convert it to a list and then *reverse* it.
2. Next, we pattern match on the *list* of strings. You can also match on arrays, but one of the nice things we can do with lists is to decompose *part* of a list.
3. The first match clause says that the list must contain *at least* three elements, the first two of which must be "version)" and "(this", and the third part will bind to the symbol `version`. The remaining elements are ignored.
4. Otherwise, we say that the first element of the list is bound to `version`, and then ignore the rest.
5. We do similar pattern matching for the helper function, which parses the version string to get out the version number and the pre-release / current version status. Notice that when we're pattern matching here, we match against a *bounded* array using `[|a;b;c;|]` syntax (and with lists, it's `[a;b;c;]`), whereas previously we checked against an unbounded list using `[a::b::c::_]`. It's this "unbounded" syntax that only works for lists – and unbounded lists work *forwards-only*, so we reverse the list before matching.

The benefit of using pattern matching for this sort of parsing rather than list / array indexing etc. is that we use the F# *language* to get to individual parts of the array. With this approach, we can much more concisely and precisely state what we want to check rather than using things such as array length values and so on. It's also impossible to "accidentally" index into an item that doesn't exist when using pattern matching against.

### 33.2.4 Updating our API with a new domain

Lastly, let's update our API to use a new domain. It's very, very easy to do, thanks to both function composition and type inference.

#### **Listing 33.6 – Updating our API with our latest domain model**

```

let loadPackageVersions = getPackage >> getVersionsForPackage >> enrich >> (fun p ->
 p.Versions) #A
let getDetailsByVersion version = loadPackageVersions >> Seq.find(fun p -> p.Version =
 version)
let getDetailsForCurrentVersion = loadPackageVersions >> Seq.find(fun p -> p.PackageVersion =
 CurrentVersion)

```

```

let details =
 "Newtonsoft.Json" |> getDetailsForVersion (Version.Parse "9.0.1")#B

#A Adding our enrich function into the existing pipeline.
#B Getting details for a specific version of Newtonsoft.Json

```

As you can see, it's really not hard – the main “work” is to add the `enrich` function into `loadPackageVersions`, and then add a small inline function which just returns the versions. We could have left that last part out, but then all callers would need to do it. Then, we simply fix the compiler errors. In the case of `getDetailsForCurrentVersion`, it involves more or less a total rewrite because it can no longer “reuse” the `getDetailsForVersion` function – but when a “total rewrite” of a function consists of half a line of code, that's not a massive problem.

### Saving data with Type Providers

We're seeing in this unit how effective type providers are at reading data. What about saving data back out? Some type providers support this, although because of the nature of them, it's probably not the standard behaviour. Often, you'll need to look at putting the data into Records etc. and then persisting them manually.

#### 33.2.5 Converting to a standalone application

The last part of this lesson will quickly port our code over to a full-blown application. It's actually pretty simple!

#### Now you try

We'll now convert our script to a standalone application.

1. Rename the file to a “.fs” file.
2. Remove any `#load` or `#references` you have.
3. Remove any “exploratory” code in the file.
4. Place a module declaration at the top e.g `module NuGet`
5. Ensure that the following remains: -
  - o Open the `FSharp.Data` namespace
  - o Create the type provider type
  - o Define your custom domain model
  - o Mark all functions as private except those that you want to expose in the API
6. From `Program.fs`, you should be able to call your API e.g.

```
getDetailsForCurrentVersion "entityframework" |> printfn "%A"
```

#### Quick Check

4. Give one reason you might use a decoupled API over type providers.

5. What benefit does pattern matching over lists give us versus indexing in directly?
6. What does the :: symbol mean in the context of pattern matching over lists?

### 33.3 Summary

This lesson showed us how we can start to use type providers within the context of a running application, rather than simply as within scripts.

- We saw how to create a simple API façade over a type provider
- We discussed times when provided types are not always suitable for use within an application, and where we might want to override
- We saw how to create a rich F# domain and map from provided types to it
- You practiced refactoring code using composition and type inference
- We saw an example of more advanced pattern matching to parse strings

#### **Try This**

Following on from a previous Try This, create an API that can return the songs for any given Dream Theater album using Wikipedia as a data source. Try simply returning strings to start with; then build up to creating an explicit domain model for Albums and Tracks, hydrating the model from the provided HTML Provider types. Then, expose this as a WPF application written in C#.

#### **Quick Check Answers**

1. Yes, by using the provided types as the “domain”.
2. The output of the first function must be the same type as the input of the second.
3. Yes.
4. Richer types e.g. DUs etc., as well as improved interoperability with C#.
5. Compile-time safety.
6. Splits a list into a head and tail.

# 34

## *Using Type Providers in the real world*

This is the final lesson on Type Providers. We've so far looked at a number of different providers in the data space, and now understand how and where to use them, what their strengths are, as well as when and where you might not use them. In this lesson, we'll wrap up by looking at some considerations of working specifically with type providers in a real-world development process.

So, you'll see: -

- Working with configuration files
- Manually redirecting type providers
- Using type providers in a continuous integration (CI) environment

### 34.1 Securely accessing connection strings with type providers

We've already looked at how to point to "live" data sources in a type provider, by redirecting from a "static" data source to a remote one at runtime. Sometimes, this works quickly and easily – as we saw with public sources such as the public NuGet feed – but occasionally you'll find that you need to point to a secure resource that means that you need to pull in some secret key at run-time in order to access the "real" data. This could be a NuGet key, or a SQL connection string or username and password – it doesn't matter. The question is, how can you provide that "secret" value safely to the type provider? We certainly don't want to "hard code" our secret connection string into the application – so what can we do?

#### 34.1.1 Working with configuration files

One option that many type providers offer is the ability to use *application configuration files* to source a connection string, rather than having to supply it directly within code as a literal value (which is what we've done so far). This provides a number of immediate benefits: -

- Config files are a well-understood concept in .NET and will be familiar to you
- Config file values can be replaced at deployment or runtime without any changes to

- your application code or binaries
- No secure strings reside in your codebase

### Now you try

Let's quickly use the configuration file support within the SQL Client type provider to show how we can easily use configuration files.

1. If you haven't yet done so, run the `build.cmd` in the root directory of the source listings to ensure you have the SQL Client type provider nuget package on disk.
2. Open the ready-made `TypeProviderConfig` solution in `code-listings\lesson-34`. This contains a simple console application, **SqlDemo**, that retrieves the first 50 customers from the database and prints their company and name to the console.
3. Observe that the connection string is hard-coded into the application in order to create the Command type.
4. Observe that the same connection is used in order to execute the query (line 8).
5. Open the `app.config` file in the project and locate the `AdventureWorks` connection string element.
6. Set the value of the `connectionString` attribute to the connection string that is being used in `Program.fs`.
7. Remove the connection string symbol (`Conn`) from F# and update the `GetCustomers` definition as follows: -

#### **Listing 34.1 – Supplying connection details to a type provider via config**

```
type GetCustomers = SqlCommandProvider<"SELECT TOP 50 * FROM SalesLT.Customer",
"Name=AdventureWorks"> #A
```

#A Supplying the connection string name to the SQL Client type provider

8. Remove the explicit `Conn` value being supplied to the `GetCustomers.Create()` call – it's no longer needed (note: this is a specific design decision of the SQL Client type provider).

We've now removed the hard-coded connection string from code and replaced it with a reference to a connection string in a configuration file.

9. Go back into the `app.config` file and deliberately change the connection string to something invalid e.g. "XXX"
10. Rebuild the solution. Observe that your solution no longer compiles – you will have to correct the connection string in the configuration file first.

### 34.1.2 Problems with configuration files

Configuration files are a familiar way to store connection strings – they’re well supported in the .NET ecosystem, with dedicated support for connection strings as well. However, there are a couple of issues with configuration files that can make life difficult to work with them and type providers, particularly when you use it as the source for both compile-time *and* run-time data.

The most important issue is that working with configuration files from within scripts is a *real pain*. By default, the `app.config` that the script will be bound to is *not* your project’s config file, but rather than one that is used to host F# Interactive - `FSI.exe`! Visual Studio’s REPL (FSI) is actually just a standard .NET application, and has its own `app.config` file.

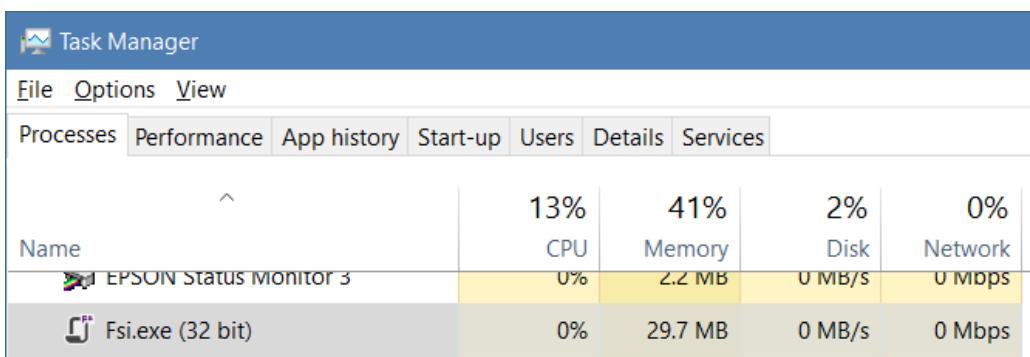


Figure 34.1 – `FSI.exe` is a standard .NET application that can be viewed in Task Manager.

Thus, if you try to bind to a type provider that is driven from a connection string from within a script (or `#load` a file that contains code that does this), there’s a good chance that the type provider will try to search within `FSI.exe.config` rather than your application config file. This is almost certainly not what you are going to want – and when working on a team of developers, each one will need to remember to do this on their own environment – if they forget, they won’t get errors when compiling (as compile-time will correctly use the `app.config` in the project!) but when `#loading` the same code from within a script, they *will* get errors etc. etc. Take it from me, it will end in tears - don’t do it.

#### SQL Client and configuration files

The SQL Client type provider actually does support some form of redirection here, and you can override the default behaviour and supply a specific path to an `app.config` at compile time so that a different config file is used. However, it’s complicated to manage and confusing to reason about – which config file is being used at compile time? What about at runtime?

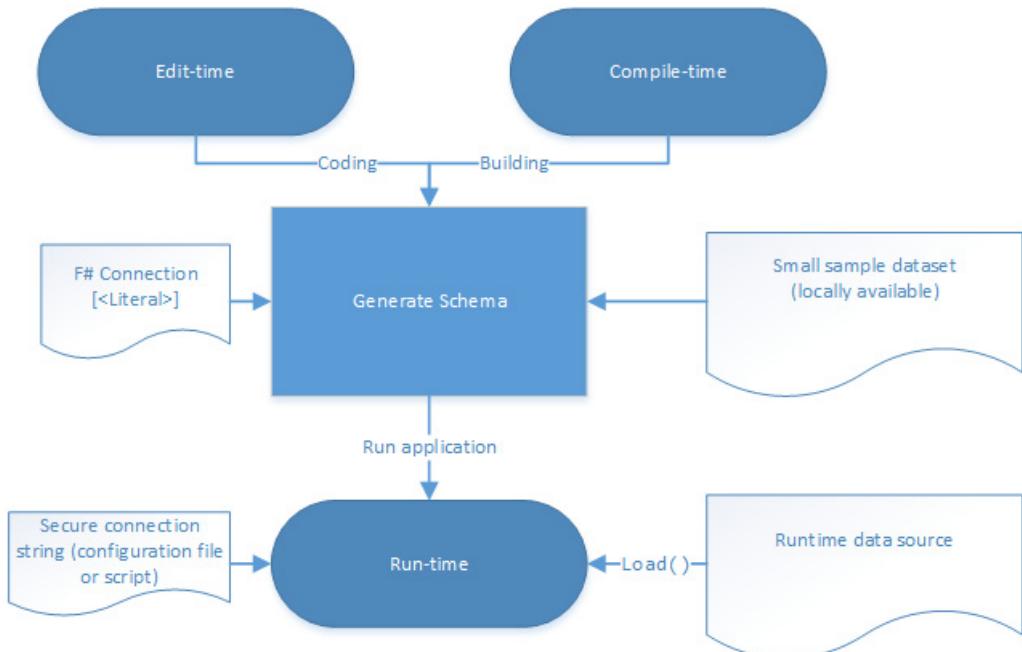
So, if you're working on a large project, with many developers, and you want to retain the ability to `#load` code that performs data access with type providers *through scripts*, be aware of the limitations. Of course, there's nothing to prevent you saying that any code you `#load` through scripts *can't* connect to a database directly, and that data (or functions that return data) must be supplied via higher order functions. In other words, code that *operates* on data should never be responsible for directly *retrieving* that data itself. It may be supplied by a function that loads data, but this is done in a decoupled manner. This is actually a good practice to have in some ways, as it reduces coupling between your business application and the source of data for them – which in turn makes testability easier.

### Quick Check

1. Can you use connection strings with Type Providers?
2. Name one benefit of using connection strings with type providers?
3. When should you not use connection strings with type providers?

## 34.2 Manually passing connection strings

So, what other options are there? The second choice is more akin to what we've discussed previously in this unit – using a static, hard-coded connection to a public / local data source for *compile-time* code, and redirecting to a secure connection at *runtime*.



**Figure 34.2 – Using different data sources and connections at compile- and run-time**

In this way, you can control your application at *compile-time* through a well-known sample data set – perhaps just enough rows of data to allow you to compile the application and for the type provider to infer schema etc. Then, at run-time, you (optionally) redirect to another data source – this might be a test database, or integration server, or private feed of data that contains secure information. This is passed in to the type provider as an *override* (so in the context of e.g. `FSharp.Data`, using `Load()` rather than `GetSample()`; in the case of `SQL Client`, it means passing in a connection string when calling `Create()`).

### Now you try

Let's now see how we can adapt our code to work with a supplied override connection string sourced from a configuration file.

1. Replace `Program.fs` with the following code:

#### **Listing 34.2 – Separating retrieval of live connection string from application code**

```

open System.Configuration

[<EntryPoint>]
let main _ =
 let runtimeConnectionString =

```

```

ConfigurationManager
 .ConnectionString
 .["AdventureWorks"]
 .ConnectionString #A
CustomerRepository.printCustomers(runtimeConnectionString) #B
0

#A Retrieving a connection string from the configuration file manually
#B Supplying that connection string to the data access layer

```

`CustomerRepository` is a module that contains the same code that we wrote before in `Program.fs`, except it now expects a connection string to be supplied to it for use at runtime; in the case of our Console application, we'll retrieve this from the `app.config` file using the standard .NET Configuration Manager.

2. Open the `DataAccessThroughScript.fsx` file and execute the code in it.
3. Observe it does the same thing as `Program.fs`, except rather than read from a configuration file, we've hard-coded a connection string in the script and supplied it in – but this could conceivably come from anywhere e.g. a text file, config file or a web service etc.

The main takeaway here is that we've decoupled our data access code from the retrieval of the connection string to the data source. Our console application uses the `app.config` file to retrieve it, but our script wasn't forced to use that as well. This sort of decoupling is crucial when developing larger applications, because being able to quickly and easily "jump in" to a specific, arbitrary source file through a script and test it out quickly and easily is a key benefit of working with scripts and a REPL.

### Quick Check

4. Why is it good practice to decouple your data access code from a connection string?

## 34.3 Continuous Integration with Type Providers

Let's now talk a little about working with continuous integration (CI) and deployment (CD) processes. Both terms essentially relate to the automation of your application being compiled, tested and optionally deployed whenever you make changes to the source code. You may already be using systems such as Team City, Jenkins, AppVeyor or Visual Studio Team Services for this. Even if you aren't using them today, they're growing in popularity all the time and it's worth knowing about how type providers work with them.

### 34.3.1 Data as part of the CI process

We already know that type providers generate types at *compile-time* not through a custom tool (as in the case of T4 templates etc.) but actually through the F# compiler. This means that whenever we perform a build of code, the type provider kicks in, accesses the data source from which it generates types that are then used later on in the compilation process.

Of course, CI servers build your code too – not just Visual Studio! In other words – *your build server will need access to a valid data source in order to compile your code*. For some data sources, this won't be a problem e.g. CSV or JSON data – simply include a static sample file in your solution and compile off of that. But what about other data sources, such as SQL Server or Azure Storage – what then? The answer is, as usual – it depends. Some data sources have emulators that can run (Azure Storage) on a build server, or may already host lightweight processes that can be used for compilation (SQL Server has LocalDB). Other systems may not have such a rich tooling environment, and you might need to have a “real” data source service available for your CI server to use in order to build code.

Creating a build process with a SQL Type ProviderFigure 34.3 contains an example of how to achieve this sort of approach when using a SQL type provider. In essence, the key is to be able to quickly and easily create an *isolated* database on the CI server itself (or at least, on a database server that the CI server can easily reach). This is relatively easy to achieve using SQL Server LocalDB (the same database service we used earlier in this unit) and Microsoft's DACPAC technology – and many CI servers support LocalDB out of the box, including AppVeyor and Microsoft's own Visual Studio Team Services.

Once the “CI” database is built, the application points to it for compilation (and potentially unit testing / integration testing). Then, once the build and test phases are completed, a separate configuration value is used to direct the *runtime* connection string which is set to point to the live database server. The build database is then destroyed; when a new commit into source control occurs, a clean database using the latest database schema is created and the whole process starts again.

Using this approach, it's extremely difficult to get into a situation where your source code and the latest database schema get out of date – if you make changes to the database and commit it without testing against the latest source code, the *compile* will fail if the schema changes are incompatible with your F#. You won't even need to run integration tests!

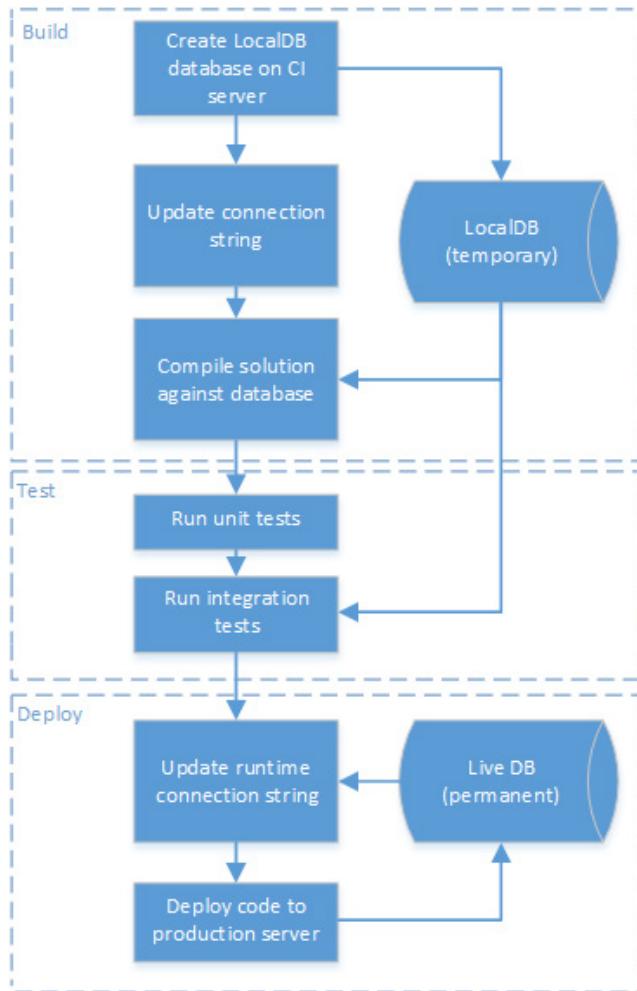


Figure 34.3 – Building a solution using a SQL Type Provider through a CI server

### Quick Check

5. Why is creating a CI process sometimes more work when using type providers?

## 34.4 Best Practices

We've seen a number of alternative mechanisms for working with configuration for type providers in this lesson. It's important to always remember that type providers nearly always

have two modes of operation – *compile-time* and *run-time*, and that you can usually redirect them to a different data source at run-time.

Things get trickier when mixing scripts as a “driver” against production code, especially if you throw configuration files into the mix – so be careful when making choices as to whether to use configuration files or not. My advice can be summarised as follows: -

| Compile-time   | Run-time          | Effort    | Best for                                    |
|----------------|-------------------|-----------|---------------------------------------------|
| Literal values | Literal values    | Very easy | Simple systems, scripts, fixed data sources |
| app.config     | app.config        | Easy      | Simple redirection, improved security       |
| Literal values | Function argument | Medium    | Script drivers, large teams, full control   |

I would advise you to start with using simpler configuration mechanisms, as you may be fine using e.g. literal values for many use cases. However, for larger teams – and where you want to be able to hook into your F# code base through arbitrary scripts – manually passing connection strings in your application gives you the most control, at the cost of greater effort.

## 34.5 Summary

In this lesson, we saw: -

- Some of the different ways to replace type provider configuration data to allow working with “private” connection strings
- Pros and cons of different approaches to working with connection strings
- Working with type providers within the context of a CI process

That’s the end of the Working with Data unit! You’ve now seen how easy it is to explore data within the context of F#, using the REPL in conjunction with Type Providers to rapidly investigate data sources, perform operations and then visualise them. You’ve also seen some tips and exercises for how to incorporate type providers within a standalone .NET application as well as some of the concerns for working with connection strings in a secure fashion.

### Try This

Perform a build using your CI tool of choice e.g. Team City etc. – build an F# application using a type provider with a local data source for compiling, but a remote one for execution.

### Quick Check Answers

1. Yes.
2. Ease of use.
3. When you want to decouple code from data source, especially with scripts.
4. Scripts can more easily access production code against any data source.

5. You'll need to ensure data sources required by type providers are available as part of the CI process in order to perform a build of your application.

# 35

## *Capstone 6*

**Before we leave the world of data, let's try to apply some of what we learned in this unit to the Bank Accounts solution we've been working on. In this lesson, we'll plug in a SQL Database layer to the application instead of the file-based repository that we've been using up until now.**

### **35.1 Defining the problem**

This capstone is essentially an exercise in hooking in a different data source to an existing codebase, but it also explores some challenges we might face with configuration and state in modules, particularly from an interop perspective – working with multi-language solutions with type providers. We'll work on performing SQL queries and commands, configuration and finally making a pluggable repository layer.

#### **35.1.1 Solution Overview**

`src/lesson-35` has both a starting solution and a completed version in the sample-solution folder. There are three projects – the two projects from the previous capstone (slightly modified), plus a new SQL database project - we'll be deploying this locally before connecting the application up to it. I've also added a binding redirect to the WPF application's `app.config` file to ensure that we always look for F#4's `FSharp.Core` (4.4.0.0) rather than the F#3.0 version (4.3.0.0).

### **35.2 Hooking up a SQL database**

The first thing we'll be doing is creating our SQL database. Look in the `BankAccountDb` project - there's already a `BankAccountDb.publish.xml` file that you can double-click from within Visual Studio to get to a pre-filled dialog – hit Publish to deploy the database.

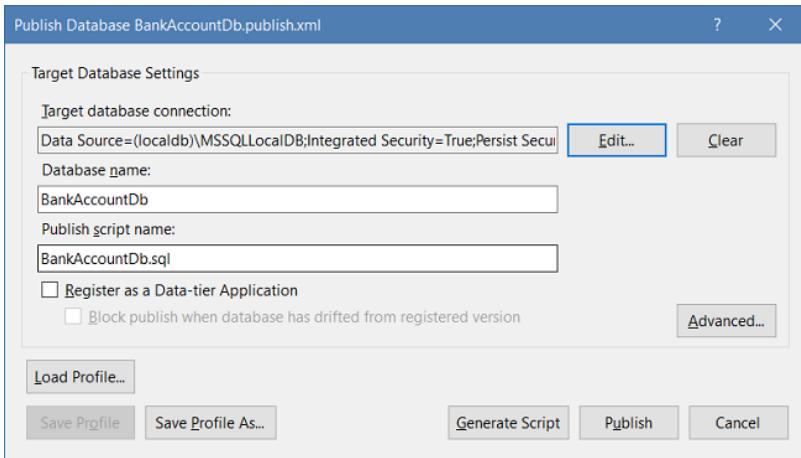


Figure 35.1 – The Publish Database dialog in Visual Studio for a Database Project

Note that I've already done a couple of things to save you some time for working with SQL: -

- The `FSharp.Data.SqlClient` package to handle data access is already installed.
- The `app.config` file is already configured to use a SQL connection string.

The database itself contains three tables: -

- A table for Account information (the account id and owner), `dbo.Account`.
- A table that contains details of every transaction that occurred, `dbo.AccountTransaction`.
- A lookup table for the two different operation types (Withdraw or Deposit), `dbo.Operation`. This is commonly known as reference data – static (or slowly changing) data that acts as a lookup for other tables.

Once you've deployed the database, you can compile the solution to ensure that it builds – if it doesn't, it's because the database you deployed to does not match the connection string in the configuration file – update it if needed by republishing the DB if required.

## 35.3 Creating a SQL data access layer

Now that the database is deployed, we can look at the code needed to interact with the database. For this exercise, we'll be using the `FSharp.Data.SqlClient` package.

### 35.3.1 The SQL Repository

Once again, don't worry if you're not a SQL guru - the solution already contains a `SqlRepository` module to get you going. This module simply contains two functions that will replace the calls to the existing `FileRepository` functions: -

- `getAccountAndTransactions` – tries to find the account and transaction history of a customer (replaces `tryFindTransactionsOnDisk`)
- `writeTransaction` – saves a single transaction to the database (replaces the same function in `FileRepository`)

If you look at both the SQL and File repositories, you'll notice that both sides (for read and write) have the same signatures – so all that's needed for you to do is to implement those two functions – you'll then be in a position to port across from the file system to SQL.

### 35.3.2 Working with SQL to retrieve account history

Reading the account history (`getAccountAndTransactions`) has the following signature: -

```
owner:string -> (Guid * seq<Transaction> option)
```

In other words – given an owner, this function should *optionally* return back the Account Id (the Guid) and a sequence of transactions for the account. You'll notice three pre-written SQL queries in the `Queries` sub-module – these are there to perform all the heavy lifting, leaving you to compose them together: -

- **GetAccountId**: Returns the Account Id for a given owner (if they exist)
- **FindTransactions**: Returns the list of all transactions for a given AccountId
- **FindTransactionsByOwner**: Returns the list of all transactions for a given Owner, along with the account id for each row.

We can approach this in one of two ways: -

1. Call both `GetAccountId` and `FindTransactions` as two separate queries; if the first returns some Account Id. If it returned None, this is a new account – just return None. Otherwise, find any transactions for that Account Id and return them together with the Account Id.

| Account History |                         |             |        |
|-----------------|-------------------------|-------------|--------|
|                 | Timestamp               | OperationId | Amount |
| 1               | 2016-12-30 17:02:53.757 | 2           | 10     |
| 2               | 2016-12-30 17:02:54.337 | 2           | 10     |
| 3               | 2016-12-30 17:02:54.510 | 2           | 10     |
| 4               | 2016-12-30 17:02:57.203 | 2           | 50     |
| 5               | 2016-12-30 17:02:57.377 | 2           | 50     |
| 6               | 2016-12-30 17:02:59.487 | 1           | 100    |
| 7               | 2016-12-30 17:03:02.533 | 2           | 24     |
| 8               | 2016-12-30 17:03:04.220 | 1           | 1      |

Figure 35.2 – The result of calling two SQL queries to locate Account and Transaction history

Observe that calling `GetAccountId` returns an `Guid option` – and *not* an array of `Guid`. This is because we've supplied the `SingleRow = true` argument to the type definition in F#.

### **Listing 35.1 – Calling GetAccountId to try to retrieve account details**

```
type GetAccountId = SqlCommandProvider<"SELECT TOP 1 AccountId FROM dbo.Account WHERE Owner = @owner", Conn, SingleRow = true> #A
let accountId : Guid option =
 GetAccountId.Create(connection).Execute("tony") #B

#A Defining a query to retrieve the Account ID for an owner
#B Executing the query, passing in a specific owner name
```

However, the “problem” with this is that we perform *two* SQL queries here. A more efficient way would be to call a single query that returns all the data in one go, and that's what `FindTransactionsByOwner` does.

2. `FindTransactionsByOwner` performs a join across both tables and returns a single result set, with `AccountId` replicated across all rows. As you can see, although this works, the “shape” of our data has changed – now, we have a single result set, with `AccountId` replicated across all rows.

| AccountId                            | Timestamp               | OperationId | Amount |
|--------------------------------------|-------------------------|-------------|--------|
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:02:53.757 | 2           | 10     |
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:02:54.337 | 2           | 10     |
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:02:54.510 | 2           | 10     |
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:02:57.203 | 2           | 50     |
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:02:57.377 | 2           | 50     |
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:02:59.487 | 1           | 100    |
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:03:02.533 | 2           | 24     |
| 8E7A7909-5667-4CFB-8726-AC3C083EA621 | 2016-12-30 17:03:04.220 | 1           | 1      |

**Figure 35.3 – The result of a single query which joins across both Account and Transaction tables**

There are actually three possibilities here: -

- There is no existing account for the owner, in which case we should get back no rows at all.
- There's an existing account for the owner, but no transactions, in which case we should get back a single row with only the `AccountId` populated.

| AccountId                            | Timestamp | OperationId | Amount |
|--------------------------------------|-----------|-------------|--------|
| 8E7A7909-5667-4CFB-8726-AC3C083EA6D1 | NULL      | NULL        | NULL   |

**Figure 35.4 – The result of a query for an account holder with no transactions**

- There's an existing account for the owner with some transactions, in which case we

should get back at least one row with all fields populated (as per Figure 35.2).

The `SqlClient` type provider is smart enough to figure out that the join might not be successful, and so the provided type generated from the query is a mandatory `Account Id`, but the remaining three fields are *optional* – what you have to do is map this into a `<Guid * Transaction seq>` option. This is a classic *impedance mismatch* between a relational database model (rows and columns) and a type system like F# which allows complex types and non-two dimensional models.

### Now you Try

1. Implement the `getAccountAndTransactions` function using one of the two approaches above.
2. If you elected the more efficient second option, you could use a *pattern match* over the records (once converted to a list): –
  - o If it's an **empty list**, there's no account holder
  - o If it's a **single item list** and **the three columns are blank**, it's an account holder with no transactions.
  - o **Otherwise**, it's an account with a proper transaction history.
3. As you construct the `Transaction` records, you'll need to set the `Operation` field by mapping from the SQL field `OperationId` (an `int`) to a `BankOperation` (an F# Discriminated Union). For now, just hard-code it to `Deposit`.

### 35.3.3 Working with reference data

Setting the `Operation` field of the `Transaction` needs a simple mapping from the `OperationId` on the database to the discriminated union `BankOperation`. The usual process for doing this would be to manually create an enum in code that “happens to match” the database structure, and map between them. Now the type provider can't do the actual mapping for us, but it can at least remove the need for us to manually create and maintain an enum. We can use the following code to easily map from one type to another without any hard coding: –

#### **Listing 35.2 – Using a provided enum to safely map into an F# domain model**

```
let toBankOperation operationId =
 match operationId with
 | DbOperations.Deposit -> Deposit
 | DbOperations.Withdraw -> Withdraw
 | _ -> failwith "Unknown DB Operation case!"
```

The `DbOperations` type is an enum type that's generated by the type provider, and contains both of bank operation cases (based on the contents of the `Operations` table in SQL). This provides us with a fairly strong guarantee that we can safely create Bank Operation DUs (although I've still kept an explicit failure handler just in case), without the need for magic numbers etc.

### Using provided types in a domain model

You'll notice that we're having to map from the generated provided types into a "pure" F# domain model. Partly this is because our solution involves C#, and provided types don't necessarily play well in a non-F# environment. But there's also a question of whether you would want to tightly-couple your domain model with data that is directly generated from SQL. I certainly wouldn't rule out the possibility of doing that – indeed, it's an extremely rapid way to get up and running – but in a larger-scale application it wouldn't be uncommon to decouple your domain model from the data store completely.

#### 35.3.4 Inserting data into SQL

When it comes to implementing the `writeTransaction` function, there are a couple of options you have. Either you can manually create an `INSERT` SQL statement, or you can use the Data Table support that the type provider offers - I've opted to go with the latter, but feel free to write your own insert statement if you want.

Either way, you'll have one issue to contend with – when we insert a Transaction, we also need to insert the Owner / Account information – but only if this is a *new* account. You could decide to make this a decision supplied to the repository – that callers have to know whether this is a new account or not – but a simpler option is simply to either: -

- Check on the database if there's already a record for this account; if there isn't, insert a new record.
- Always try to insert the account / owner record, and if it fails, swallow the exception that's raised as a result.

You can write code as follows to create a data table, insert a row into it, and then persist it to the database: -

#### **Listing 35.3 – Creating a data table with a row for insertion into SQL**

```
use accounts = new AccountsDb.dbo.Tables.Account() #A
accounts.AddRow(owner, accountId) #B
accounts.Update() #C
```

```
#A Creating an in-memory datatable
#B Adding a row to the datatable
#C Updating the database with the new row
```

You can use the existing `GetAccountId` query to check if there's an existing account id for the supplied owner and match on the result to test whether to create an account or not. Alternatively, you could elect to always try to perform the insert, catch the exception if it's a `SqlException` where the text contains "Violation of PRIMARY KEY constraint" e.g.

#### **Listing 35.4 – Pattern matching over an exception**

```
try codeThatMightThrow() #A
with
| :? SqlException as ex when ex.Message.Contains "Violation of PRIMARY KEY constraint" -> ()
```

```

#B
| _ -> reraise() #C

#A Executing some code in a try / with block
#B Checking against a type in with the :? operator conjunction with a when clause
#C Rethrow the current exception

```

The `:?` operator above is analogous to a safe form of “type cast” – if the exception is a `SqlException`, we bind it to the symbol `ex` and can then use it within the `when` clause. This sort of functionality is being brought into C#7 as a limited form of pattern matching.

It’s worth bearing in mind that the `writeTransaction` function is a function in our system that returns unit – it’s doing a write to a database and gives back nothing. This makes it somewhat difficult to “reason about” – effectively, we’ve encoded the error into an exception. A more functional approach might be to return a `Result<unit>` – in other words, at least explicitly state that the save was successful, even if we don’t have any payload with that success.

When performing the save, you’ll be mapping backwards, and taking the fields from a `Transaction` record into a function call that takes primitives (strings, ints etc.), including the integer for the `OperationId` on the database. You can use another pattern match to map from the `BankOperation` to the `DbOperations` enum value, essentially inverting Listing 35.2 (a clean approach might be to make a dedicated helper submodule to store both conversion functions).

Now that we have our SQL data access layer written, the only thing left to do is plug it in. The easiest thing to do is simply replace the calls in the `Api` to any functions in the `FileRepository` module with those in the `SqlRepository` module. They should be drop-in replacements as the function signatures match exactly!

## 35.4 Making a pluggable data access layer

This section will look at possible ways to improve the architecture of the application.

### 35.4.1 Pluggable data access repositories

Coming from an OO background, your instinct at this point might be to wonder why we’re tightly coupling the “service” layer i.e. the orchestration logic in API etc. with the “data” layer i.e. the SQL Repository. Can’t we decouple the two of them – perhaps using some form of dependency injection with an IoC container? You bet!

It’s beyond the scope of this lesson to do that in detail, but in the sample solution you’ll see how it’s done. You can actually do it yourself without too much difficulty: -

#### **Now you Try**

1. Inside `Api.fs`, everywhere you see a hard reference to `SqlRepository`, simply remove that and pass in the function (`writeTransaction` or `getAccountAndTransactions`) as a higher-order function.

2. Note that since both `Withdraw` and `Deposit` both load up the account from the DB before performing the write, they have dependencies on both read *and* write SQL functions!
3. Now the caller to the API needs to pass in the SQL-dependent functions – in other words, the view model code in the client. Compiling the app will show you exactly where you need to pass in the functions – you'll also have to make the SQL Repository code public again, since the view model will need to inject it into the API calls.

### 35.4.2 “Reusing” dependencies across functions

The problem with this solution is that we now have to repeatedly pass in the SQL dependencies for every function call, every time we call an API function – not great. In a purely functional solution, we'd simply partially apply those functions with the relevant dependencies within a bootstrapper of some sort (I believe the current trendy term to use these days is composition root) and then pass the partially applied versions into the app.

However, it's sometimes useful to “group up” a list of functions into a single, logical bundle. In this case, it's especially useful because all of the functions (read and write to database) use the same dependencies i.e. the calls to the read and write functions. In F#, it's not possible to parameterise a *module* with arguments, so in this case we can fall back to using an *interface* that represents our API:

#### **Listing 35.4 – Representing functions through an interface rather than a module**

```
/// Represents the gateway to perform bank operations.
type IBankApi = #A
 abstract member LoadAccount : customer:Customer -> RatedAccount #B
 abstract member Deposit : amount:Decimal -> customer:Customer -> RatedAccount
 abstract member Withdraw : amount:Decimal -> customer:Customer -> RatedAccount
 abstract member LoadTransactionHistory : customer:Customer -> Transaction seq

#A Name of the interface
#B Members of the interface
```

We can now create an *instance* of this interface which takes in the dependencies as *function arguments*: -

#### **Listing 35.5 – Creating a factory function for the Bank API**

```
let buildApi readData writeData = #A
 { new IBankApi with #B
 member this.LoadAccount(customer) = // code elided
 member this.Deposit amount customer = // code elided
 member this.LoadTransactionHistory(customer) = // code elided
 member this.Withdraw amount customer = // code elided }
```

#A Factory function taking in varying dependencies as function arguments

#B Implementations with access to dependencies

Using this, you can easily make a SQL or File bank API. There's a fully working example in the sample solution, but a few things to note here: -

- We pass in the read and write functions as function arguments
- In the function body, we create an *instance* of `IBankApi` (F# allows you to create “anonymous objects” from interfaces – there's no need to formally declare a type)
- The implementations of the member functions can all access the read and write dependencies. In effect, we're partially applying all the member functions with both dependencies simultaneously!

This approach to viewing interfaces is an interesting one. In the OO world, we tend to think of interfaces as objects that have a list of member methods on them to fulfill the interface; in the FP world, you can flip this on its head and simply think of interfaces as a group of functions that can be *looked up* by name. In that sense, rather than using an interface, we could just as easily have used an F# *record* of functions: -

#### **Listing 35.6 – Using a Record instead of an Interface**

```
type BankApi = #A
 { LoadAccount : Customer -> RatedAccount
 Deposit : Decimal -> Customer -> RatedAccount
 Withdraw : Decimal -> Customer -> RatedAccount
 LoadTransactionHistory : Customer -> Transaction seq }

let buildApi readData writeData = #B
 { LoadAccount = fun customer -> ()//..
 Deposit = fun amount customer -> ()//..
 Withdraw = fun amount customer -> ()//..
 LoadTransactionHistory = fun customer -> Seq.empty }
```

#A Defining a record of functions  
#B Creating an implementation of the record

I personally prefer to use interfaces in such situations – it's a well-known pattern – although using Records in this way does offer some advantages, such as better support for type inference.

#### **35.4.3 Handling SQL connection strings directly**

As we've observed earlier in this unit, using application configuration files is a quick way to get up and running with SQL but moving to an explicit connection string gives us more flexibility and control, plus it makes it much easier to work with SQL in scripts. The final thing you'll want to do is migrate from the “implicit” connection strings that using a configuration file offers, to working explicitly with a connection string.

### Now you Try

1. For all functions in the SQL Repository module, take in an extra string argument called `connectionString` and pass that into all calls that connect to the DB e.g. `GetAccountId.Create()` for reads.
2. Do the same for the `Update()` calls, except you'll need to manually create a `SqlConnection` object first (and remember to `Open()` it!)

#### **Listing 35.7 – Creating a SQL Connection object**

```
use conn = new System.Data.SqlClient.SqlConnection(connectionString) #A
conn.Open() #B

#A Creating a SQL connection object in a using block
#B Opening the connection before using it
```

3. Where the `IBankApi` is created, you'll need to pass a `connectionString` into both the higher-order function calls. You can opt to get a handle to the connection string here, or simply take the connection string in as an argument.
4. Either way, either in the `ViewModel` or the factory function, you'll need to get a handle to the connection string. Firstly, add a reference to the **System.Configuration** assembly.
5. You can then use the  `ConfigurationManager` static class to pull out the connection string.

#### **Listing 35.8 – Retrieving a connection string from the configuration system**

```
open System.Configuration
ConfigurationManager.ConnectionStrings["AccountsDb"].ConnectionString #A

#A Using the ConfigurationManager to retrieve the AccountsDb connection string
```

Now that you've done this you've completely decoupled the API layer from the data access layer, whilst you've also removed the dependency on the application configuration from the SQL layer.

## 35.5 Summary

That's another capstone done! You've successfully integrated a SQL data access layer into the application using a type provider to quickly and easily give you strongly-typed access to a database. We also had a look at a practical example of how we can create a pluggable interface from F# in tandem with higher order functions in order to decouple an api from a data access layer.

# Unit 8

## *Web Programming*

You can't have any programming language and platform these days without having a decent story for web programming! Luckily, F# is a great fit for working on the web, with a great set of libraries and frameworks that it can use. We'll be covering both *creating* and *consuming* web-based resources in this unit, using a set of technologies that exist in the general .NET ecosystem – ones that you'll probably already be familiar with - and some cool F#-specific technologies as well.

# 36

## Asynchronous Workflows

**For the last few units, we've been focusing almost exclusively on libraries and frameworks that work with F#. In the first lesson of this unit, we're going to briefly hop back to the language side of things and introduce an important language feature in F# called *asynchronous workflows*, which allow you to orchestrate asynchronous and multi-threaded code in a manageable way.**

We'll see: -

- Why asynchronous programming is important
- What async workflows are
- Comparing async programming in F# and C#
- Look at computation expressions in general

### 36.1 Comparison of Synchronous and Asynchronous models

Whilst there's a decent chance that you're already aware of multi-threading and asynchronous programming (especially if you've been working with C#5), it's worth quickly covering what these terms are, and why they're important for the web. I'm going to slightly simplify things here by simply talking about "synchronous" and "asynchronous" work. The former represents work that happens in a single, sequential flow of execution – essentially, all the code we've written thus far. Asynchronous work represents the notion of doing work "in the background", and when it completes, receiving some notification that the background work has finished, before consuming the output and continuing.

Why might we want to perform "background work"? The most obvious reason is performance, because doing work in the background allows us to perform multiple tasks at the *same time* – you can "kick off" several tasks "in the background", each working with their own pieces of data, and then when they're all done, carry on with the main program. There are efficiency concerns as well – particularly when it comes to communicating with "external" systems such as databases or other web servers etc., as using asynchronous APIs can "free

up” threads in the application whilst they “wait” for the external resource to return, so that the web application can effectively handle more requests from more users at the same time.

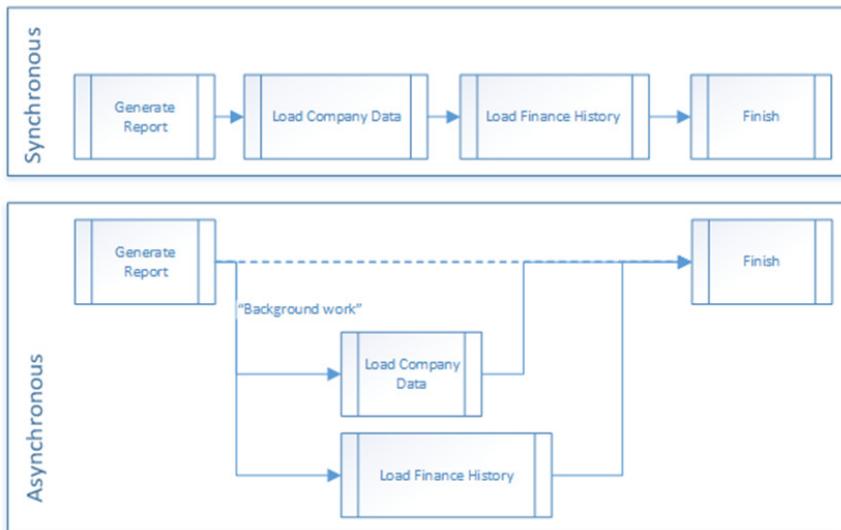


Figure 36.1 – An example of performing the same work both in synchronous and asynchronous fashion.

### 36.1.1 Threads, Tasks, and I/O bound workloads

There are several (perhaps too many!) ways of performing “background work” in .NET. I’m not going to devote too much time to this, but rather give you a brief overview – there are many great resources that you can read up on this elsewhere in painful detail.

#### **THREADS**

Essentially, the lowest “primitive” for allocating background work is the thread. A .NET application has a “thread pool” with a finite number of threads; when you execute work on a background thread, the thread pool assigns a thread to carry out the work. When the work is finished, the thread pool reclaims the thread, ready for the next piece of background work.

#### **TASKS**

Introduced in .NET 4, Tasks are a higher-level abstraction over threads. They’re much easier to work with and reason about, with good support for cancellation and parallelism, and so effectively become the de-facto type in C# and VB .NET to use for performing background work.

## I/O BOUND WORKLOADS

Both Threads and Tasks can both be thought of as supporting *CPU-bound* workloads – work that is carried out in the current process. *I/O-bound* workloads are background tasks that you wish to execute which *don't* need a thread to run on. These are typically used when communicating and waiting for work from an *external* system to complete. Instead of utilising a thread from the pool (which both the Thread and Task models do), the operating system provides a low-level callback that .NET monitors; when the external system returns with data, .NET picks up the response and resumes work. These sorts of methods are *truly asynchronous* – they don't block any threads whilst running.

**Table 36.1 - Examples of I/O- and CPU-bound workloads**

| Type | Example                                           |
|------|---------------------------------------------------|
| CPU  | Calculating the average of a large set of numbers |
| CPU  | Running a set of rules over a loan application    |
| I/O  | Downloading data from a remote web server         |
| I/O  | Loading a file from disk                          |
| I/O  | Executing a long-running query on SQL             |

Why is asynchronous code important with regards to web applications in particular? The main reason is simply *throughput*. A web application may be receiving hundreds, or even thousands of requests from different users per second – and processing those requests as quickly as possible is usually extremely important. As a .NET process only has a finite number of threads, it's important to utilise those threads as effectively as possible – and this means eliminating times where a thread is blocked “idling” for e.g. waiting for SQL to return some data to us. That's why, where possible, we should try to work with *asynchronous* code.

### 36.1.2 Problems with asynchronous models

From a programming perspective, there are a few difficulties when working asynchronous code. In essence, reasoning about several “background items” working concurrently (at the same time) is *hard*, particularly with things like synchronization of multiple work items or exception handling etc.

In the spirit of this book, I'm not going to go over the same ground that's been covered in depth many times before (particularly since C#5 came into being) – suffice it to say that writing truly asynchronous code using what's called “continuation passing style” is *very hard* to get right. So, in order to solve this the F# team came up with a brilliant solution known as *asynchronous workflows*.

### What about `async / await`?

If you've used a relatively recent version of C# (basically C#5 onwards), you'll almost certainly be aware of the `async / await` pattern. Great! Then a lot of this lesson will feel very natural to you. That's because the `async / await` pattern is actually based on F#'s asynchronous workflows – although as we'll see, `async / await` isn't quite as flexible.

### Quick Check

1. What is the preferred type in .NET to use when executing work in the background?
2. What is the difference between CPU and I/O bound workloads in terms of their activity on a thread?

## 36.2 Introducing Asynchronous Workflows

Whilst Threads and Tasks are *library* features to schedule background work, the `async` workflow is a *language* level feature to achieve the same thing.

### 36.2.1 Async workflows basics

The general gist of things is quite easy – simply wrap any code block that you want to execute in the background in an `async { }` block. As we'll see, this can be something as simple as a single value to doing a complex set of calculations. Let's look first at how we might schedule some work "in the background" by comparing carrying out the same piece of work synchronously and asynchronously – execute both blocks separately in a script to observe the differences in behaviour.

#### **Listing 36.1 – Scheduling work with `async` blocks in F#**

```
printfn "Loading data!" #A
System.Threading.Thread.Sleep(5000) #A
printfn "Loaded Data!" #A
printfn "My name is Simon." #A

async { #B
 printfn "Loading data!"
 System.Threading.Thread.Sleep(5000)
 printfn "Loaded Data!" }
|> Async.Start #C

printfn "My name is Simon."
```

#A A conventional, synchronous sequential set of instructions

#B Wrapping a portion of code in an `async` block

#C Starting the `async` block in the background

The first time, it executes a set of code instructions that block the thread for 5 seconds (simulating loading some data from an external system) before printing someone's name. The problem is that we can't print the final line until the first three are completed (and indeed, FSI will be completely blocked for you whilst it executes).

The second version "wraps" the long-running portion of code into an `async { }` block, and fires it off in the background using the `Async.Start` method. The difference is that now, the

person's name is printed immediately, whilst the asynchronous block executes in a background thread.

Let's now look at another example – this time, unlike the previous example which was a "fire-and-forget" one, this time we'll see how to asynchronously execute some code that *returns a value*.

#### **Listing 36.2 – Returning the result from an async block**

```
let asyncHello : Async<string> = async { return "Hello" } #A
let length = asyncHello.Length #B
let text = asyncHello |> Async.RunSynchronously #C
let lengthTwo = text.Length

#A Returning a value from an async block
#B Compiler error when trying to access a property of an async value
#C Executing and unwrapping an asynchronous block on the current thread
```

Try this out yourself, executing *one line at a time*. As you can see, by "wrapping" the text "Hello" in an `async` block, rather than getting back a string, we get back an `async` string. This essentially means "when you start this `async` workflow, it will at some point in the future return a string". You *can't* "dot into" an `async` workflow to e.g. get the length of a string – you first need to "unwrap" the value. A couple of other points worth noting here: -

1. Unlike regular expressions, the result of an `async` expression must be prefixed with the `return` keyword.
2. We can "unwrap" an `Async<_>` value by calling `Async.RunSynchronously`. This is roughly equivalent to `Task.Result` – it blocks the current thread until the workflow is completed.

One other very important distinction to make here is that creating an `async` block does *not* automatically start the work in the block – you have to *explicitly* start it. One way is to use one of the methods on the `Async` class e.g. `RunSynchronously` or `Start` (see 36.5.3 for a full list of useful methods). Also – unlike `Task.Result`, if you repeatedly called `RunSynchronously` on an `async` block, it will re-execute the code *every time*.

#### **36.2.2 More complex async workflows**

You can do more than simple one liners in an `async` block – if you want to delegate some work that does more than just hello world, that's no problem – you can wrap entire function calls and blocks of code within them. Here's an example of a more complex `async` workflow that calls a nested function.

#### **Listing 36.3 – Larger async blocks in F#**

```
open System.Threading

let printThread text = printfn "THREAD %d: %s" Thread.CurrentThread.ManagedThreadId text
```

```

let doWork() = #A
 printThread "Starting long running work!"
 Thread.Sleep 5000
 "HELLO"

let asyncLength : Async<int> =
 printThread "Creating async block"
 let asyncBlock =
 async {
 printThread "In block!" #B
 let text = doWork()
 return (text + " WORLD").Length } #C
 printThread "Created async block"
 asyncBlock

let length = asyncLength |> Async.RunSynchronously #D

```

#A A standard function that simulates a long running piece of work  
#B Printing to console within an async block  
#C Returning a number from within the block  
#D Unwrapping the number

If you execute this code in one chunk, you'll see the following output: -

```

THREAD 1: Creating async block
THREAD 1: Created async block
THREAD 5: In block!
THREAD 5: Starting long running work!

```

It's important to realise that no work actually occurs until we execute the *final line* in the script – everything up until that point simply compiles the code but does *not* start the background work. You can create async blocks easily, “pass them around” your application without problems, and then execute them at a time of your choosing by calling `Async.RunSynchronously`.

### Quick Check

3. What extra keyword must you use in Async blocks to return a value?
4. Do async workflows immediately execute on creation?

## 36.3 Composing asynchronous values

Async blocks like we've seen so far are already quite useful in their own right – you can easily reason about what you want to run asynchronously, and then pass that code around as a simple value. However, we've seen that to “unwrap” an asynchronous value, you need to call `Async.RunSynchronously`. This *blocks the current thread* until the async workflow has executed – which is a real shame! What's the point in passing around code that can run in the background if we need to block the current thread to get at the result? Luckily, F# has a built-in way to “continue” when a background workflow completes, called `let!`.

**Listing 36.5 – Creating a continuation using let!**

```
let printHelloWorld =
 async {
 let! text : string = getTextAsync #A
 return printf "%s WORLD" text } #B

printHelloWorld |> Async.Start #C

#A Using the let! keyword to asynchronously “unwrap” the result
#B Continuing work with the unwrapped string
#C Starting the entire workflow in the background
```

There are a few things happening here – so let’s take it step by step. First, we create an `async` block. Inside that block, we “execute” the `getTextAsync` computation and *wait* for the string result by using the `let!` keyword (if you’ve used C#5’s `async / await` before, think of this as loosely equivalent to `await`). This keyword is *only valid* when inside the `async` block – you can’t use it outside. Now, notice that the value `text` is a type `string` - *not* an `async string`! Essentially, `let!` waits for `asyncWork` to complete *in the background* (it doesn’t block a thread), “unwraps” the value for us and then continues on. Try replacing the `let!` with just `let` and see what happens. Finally, we close the block and then start this “composed” `async` workflow *in the background* using `Async.Start`. `Async.Start` is perfect if you want to “kick off” a workflow that has no specific end result, as in this case where the workflow simply prints out something to the console.

Of course, `async` blocks also allow you to perform `try / with` blocks around a `let!` computation; you can nest multiple computations together and use .NET `IDisposable`s without a problem.

**Quick Check**

5. What is the purpose of the `let!` keyword?
6. When do you need to use the `return` keyword in F#?

## 36.4 Fork / Join

One thing that’s extremely easy to achieve with `async` blocks is to perform what’s known as a *fork / join* – the ability to launch several `async` workflows in the background, wait until *all of them* are completed, and then continue on with all the results combined. In F#, we use `Async.Parallel` to collate a *collection* of `async` workflows into a *single*, combined workflow. Note that it doesn’t start the new workflow – it simply creates a *new workflow* that represents the result of *all the individual workflows* collated together.

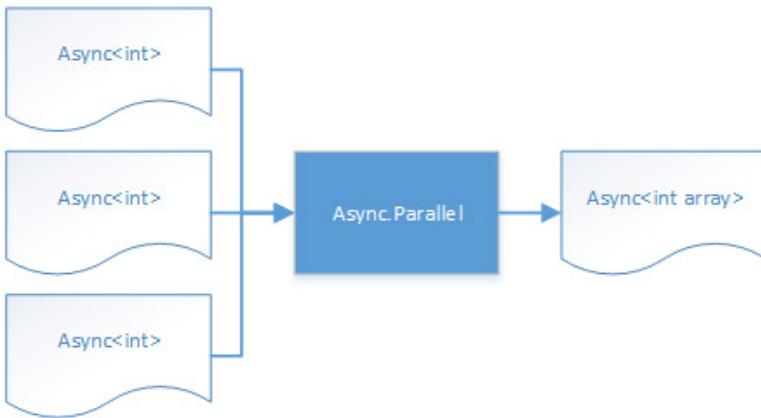


Figure 36.2 – Async Parallel collates a number of async workflows of the same type into a single async

Here's how we might asynchronously generate and work with 50 random numbers: -

#### **Listing 36.6 – Looking at Fork / Join with Async.Parallel**

```

let random = System.Random()
let pickANumberAsync = async { return random.Next(10) }
let createFiftyNumbers =
 let workflows = [for i in 1 .. 50 -> pickANumberAsync()] #A
 async {
 let! numbers = workflows |> Async.Parallel #B
 printfn "Total is %d" (numbers |> Array.sum)
 }
createFiftyNumbers |> Async.Start

#A Creating 50 asynchronous computations
#B Executing all computations in parallel and unwrapping the collated results

```

Again, try this out yourself – the important thing to note here is the use of `Async.Parallel`. Remember that this handy function goes from `Array<Async<T>>` to `Async<Array<T>>` – see Figure 36.2. In our case, this means going from an `Array<Async<int>>` to an `Async<Array<int>>` which can be awaited (incidentally, this is very similar to `Task.WhenAll`).

#### **Now you try**

Let's move try downloading data from HTTP resources – using a BCL method that supports F# `async`'s feature *natively*. This means that it'll use .NET's native `async` support so that we can download data much more efficiently.

1. Write a function, `downloadData`, which takes in a *single* string `url` and *asynchronously* returns the number of bytes in the contents. It should have a signature of `string -> Async<int>`.
  - o You can use the standard `System.Net.WebClient` object to perform the download.

- o There's a handy method on the `WebClient` designed to work specifically with `Async` workflows called `AsyncDownloadData` that you can use (you'll have to create a `System.Uri` from the string to work with this function).
2. You can use `let!` to "unwrap" the `async<byte[]>` into a `byte []`.
  3. You can then return the `Length` of the byte.
  4. Within your script, create an array of three URLs
    - o <http://www.fsharp.org>
    - o <http://microsoft.com>
    - o <http://fsharpforfunandprofit.com>
  5. Use standard `Array.map` in conjunction with your `downloadData` function to map the array of `string` into an array of `Async<int>`
  6. Use `Async.Parallel` to execute the workflows in parallel and return all the results as one.
  7. Use `Async.RunSynchronously` to block until you have the results.
  8. Sum the results using standard `Array.sum` to get the total number of bytes downloaded.

Your code should like something like this: -

#### **Listing 36.7 – Asynchronously downloading data over HTTP in parallel**

```
let downloadData url = async {
 use wc = new System.Net.WebClient()
 printfn "Downloading data on thread %d" CurrentThread.ManagedThreadId
 let! data = wc.AsyncDownloadData(System.Uri url)
 return data.Length }

let downloadedBytes =
 urls
 |> Array.map downloadData
 |> Async.Parallel
 |> Async.RunSynchronously

printfn "You downloaded %d characters" (Array.sum downloadedBytes)
```

#### **Quick Check**

7. What does `Async.Parallel` do?

## **36.5 Tasks and Async Workflows**

Interoperating with TasksThe `Async` type isn't something pervasive in .NET – instead, there's a good chance that any libraries you use work with the `Task` type. Luckily, F# has a couple of handy combinators (or "transformation functions") that allow us to go between `Task` and `Async`, similar to how `Option` has combinators for `Nullables`: -

- `Async.AwaitTask` – converts a `Task` into an `Async` workflow
- `Async.StartAsTask` – converts an `Async` workflow into a `Task`.

## Now you Try

Let's change the above code so that it uses the *Task* version of `DownloadDataTaskAsync`, rather than the F#-specific `AsyncDownloadData` version.

1. Starting from Listing 36.7 replace the call to `AsyncDownloadData` with one to `DownloadDataTaskAsync`.
2. Your code will not compile – you can only “unwrap” an `Async<_>` inside an `async` block, but we have a `Task` here. So, pipe the `Task` into the `Async.AwaitTask` function to convert the `Task` into an `Async`. Your code will compile again.
3. Replace the call to `Async.RunSynchronously` with a call to `Async.StartAsTask`. Observe that `downloadedBytes` is no longer an `int[]` but a `Task<int[]>`.
4. In the call to the `printfn` expression, rather than printing out `downloadedBytes`, print out `downloadedBytes.Result`.

### **Listing 36.8 – Replacing calls to and from Async with Task-bound methods**

```
let downloadData url = async {
 let! data =
 wc.DownloadDataTaskAsync(System.Uri url) |> Async.AwaitTask #A
 return data.Length }

let downloadedBytes =
 urls
 |> Array.map downloadData
 |> Async.Parallel
 |> Async.StartAsTask #B

printfn "You downloaded %d characters" (Array.sum downloadedBytes.Result)

#A Using the AwaitTask combinator to convert from Tasks to Async
#B Using the StartAsTask combinator to convert from Async to Task
```

#### **36.5.1 Comparing Tasks and Async**

Let's take a quick look at some of the distinctions between .NET Task and F#'s `Async` types, as well as the `async / await` pattern.

**Table 36.1 – Tasks and Async compared**

|                         | Task & Async Await                                         | F# Async Workflows                    |
|-------------------------|------------------------------------------------------------|---------------------------------------|
| Native support in F#    | Via <code>Async</code> combinators                         | Yes                                   |
| Allows status reporting | Yes                                                        | No                                    |
| Clarity                 | Hard to know where <code>async</code> “starts” and “stops” | Very Clear                            |
| Unification             | <code>Task</code> and <code>Task&lt;T&gt;</code> types     | Unified - <code>Async&lt;T&gt;</code> |
| Statefulness            | <code>Task</code> Result only evaluated once               | Infinite                              |

There's a great post by the venerable Tomas Petricek on `async / await` vs `async` workflows with several excellent examples of where `async / await` breaks down, so I'd recommend you have a read of this in your own time: <http://tomasp.net/blog/csharp-async-gotchas.aspx/> - particularly around the notion of where `async` "starts" and "stops".

In general, internally in your F# code I would recommend using `Async` workflows wherever possible. There are times where you might want to use `Tasks` instead, but it's pretty unusual. One case is if for interop purposes; another is for extremely large number of CPU-bound items where `Task` is more efficient – `Async` was originally designed for asynchronous work rather than, for example, huge numbers of tiny CPU-bound work items.

Of course, it's not uncommon to rely on libraries that themselves expose `Tasks` (including many within the BCL) - but generally you'll immediately convert them to `Asyncs` so that you can use `let!` on them within an `Async` block.

### 36.5.2 Useful `Async` keywords

Here's a quick list of most of the common `Async` keywords and functions that you'll be using.

**Table 36.2 – Common `Async` commands**

| Command                             | Usage                                                                                                                           |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <code>let!</code>                   | Used within an <code>async</code> block to unwrap an <code>Async&lt;T&gt;</code> value to <code>T</code>                        |
| <code>do!</code>                    | Used within an <code>async</code> block to wait for an <code>Async&lt;unit&gt;</code> to complete                               |
| <code>return!</code>                | Used within an <code>async</code> block as a shorthand for <code>let!</code> and <code>return</code>                            |
| <code>Async.AwaitTask</code>        | Converts a <code>Task&lt;T&gt;</code> to <code>Async&lt;T&gt;</code> , or a <code>Task</code> to <code>Async&lt;unit&gt;</code> |
| <code>Async.StartAsTask</code>      | Converts an <code>Async&lt;T&gt;</code> to a <code>Task&lt;T&gt;</code>                                                         |
| <code>Async.RunSynchronously</code> | Synchronously unwraps an <code>Async&lt;T&gt;</code> to <code>&lt;T&gt;</code>                                                  |
| <code>Async.Start</code>            | Starts an <code>Async&lt;unit&gt;</code> computation in the background, i.e fire-and-forget                                     |
| <code>Async.Ignore</code>           | Converts an <code>Async&lt;T&gt;</code> to <code>Async&lt;unit&gt;</code>                                                       |
| <code>Async.Parallel</code>         | Converts an <code>Async&lt;T&gt;</code> array to an <code>Async&lt;T&gt;</code> array                                           |
| <code>Async.Catch</code>            | Converts an <code>Async&lt;T&gt;</code> into a two-case DU of <code>T</code> or <code>Exception</code>                          |

Some of these you'll have not seen yet, but it's good to know that they are there – experiment with them in a scratchpad in order to see yourself how they work.

#### Quick Check

8. How do we convert from `Async` to `Task`?
- 9 Name one benefit that `Task` offers over `Async`.

## 36.6 Summary

You've now completed the Async lesson. We saw: -

- An introduction to what the async expression is
- How we can use it to easily compose multiple async blocks together
- How to execute multiple async workloads in parallel

### **Try This**

- Write an application to demonstrate the difference in terms of performance and threads between synchronous, multi-threaded and asynchronous parallel downloading of 10 HTTP resources.
- Then, try using the Async methods included in FSharp.Data for downloading JSON data from a remote resource.
- Finally, try to handle an exception raised in an async block using the `Async.Catch` method.

### **Quick Check Answers**

1. `System.Threading.Tasks.Task`
2. CPU bound workloads consume a thread whilst working; I/O ones should not.
3. `return`
4. No. You should start them explicitly using `Async.Start` or `RunSynchronously`.
5. To asynchronously wait for another async computation to complete
6. At the end of an async block to return a result.
7. `Async.Parallel` allows fork / joins of multiple Async values.
8. Using `Async.StartAsTask`.
9. Tasks allow us to report on the progress of a work item. Tasks are also more efficient when working with very large groups of work items.

# 37

## *Exposing data over HTTP*

In this lesson, we're going to look at ways in which to create HTTP-enabled APIs using F# on the .NET platform. Firstly, we'll do this using the well-known Microsoft ASP .NET framework, using the Web API component; we'll then move on to looking at an alternative web technology, Suave.

We'll see: -

- Working with ASP .NET Web API and F#
- How to reason about HTTP response codes in F#
- Working with Async and ASP .NET
- Using Suave, an F#-first web application model

### 37.1 Getting up and running with ASP .NET Web API

Let's jump straight into this lesson and dispel any fears that your ASP .NET applications are somehow not compatible with F# by creating an ASP .NET application, in F#! We'll look at two different hosting mechanisms – web projects and console apps.

#### 37.1.1 Web Projects with F#

Exposing data over HTTP in .NET is most commonly done using Microsoft's ASP .NET framework; if you've done any form of web programming on .NET, it's a pretty safe bet that you've used it in some form, be it using the old-school Web Forms, its replacement MVC, and its API-focused sibling Web API. The most common and popular way to host an ASP .NET application in .NET is as a *web project*. These projects bootstrap into IIS (or its little sibling, IIS Express), with a `web.config` file providing configuration information to .NET (rather than an `app.config`).

Here's the bad news out of the way: Within the F# world, out of the box, there is no tooling support *in Visual Studio* for web projects. Now here's the good news: The F#

community has fixed this itself with some excellent third-party templates that *are* available within Visual Studio.

### Problems with F# and Web Projects

The basic lack of support for web projects boils down to the fact that Visual Studio doesn't understand how to resolve a web project with F# tooling. You can manually "create" a web project by creating a standard F# Console Application and changing the project GUID in the `.fsproj`, but that leaves you with another problem: the Add New Item dialog won't work, so you'll never be able to add a new file to the project. Thankfully, there's a registry fix for this that can be applied, and will automatically be done when you install the F# MVC template.

### Now you Try

Let's create a basic ASP .NET Web API 2 application with F#.

1. Open Visual Studio and choose the standard **New Project**.
2. In the New Project dialog, select **Online > Templates > Visual F# > F# MVC 5**.
3. From the next dialog, choose Web API 2.2 and Katana 3.0 (Empty). You'll now have an empty Web API project with no controllers. Whilst there are larger, ready-made templates here with ready-made controllers etc., we'll start with an empty one so that you can see the basics first (you'll also notice a non-Katana Web API project, which uses the older `global.asax`-based web projects). Just in case you're unaware of Katana – it's essentially a middleware layer that allows you to plug lightweight OWIN-compliant web applications into IIS.
4. Build the solution to pull down any NuGet packages required.
5. Run the solution. You'll see that a web site is created on IIS Express, and a browser page opens. Of course, there's nothing to see yet!

At this point, let's take a look at what we have in the solution – a single F# file, `Startup.fs`. This file acts as the "bootstrapper" of your application and is very, very similar to what you would have with the equivalent empty project in C#. There's essentially a `Startup` class that contains two methods: one to configure the ASP .NET app builder, and another to configure Web API.

6. Delete the two serialization lines (we'll come back to them shortly).
7. Add a new file to the project, `Controllers.fs`, and enter the following code into the file. This file will implement a simple ASP .NET controller class that can respond to HTTP GET requests to the `api/animals` route.: -

#### **Listing 37.1 – Our first Web API controller**

```
namespace Controllers
open System.Web.Http

type Animal = { Name : string; Species : string } #A
```

```
[<RoutePrefix("api")>] #B
type AnimalsController() =
 inherit ApiController()

 [<Route("animals")>] #C
 member __.Get() = #D
 [{ Name = "Fido"; Species = "Dog" }
 { Name = "Felix"; Species = "Cat" }]

#A Creating a type from which to expose data.
#B Setting the Web API route prefix
#C Setting the Web API route
#D Creating a GET handler
```

This should be somewhat similar to what you've done in the past with Web API in C# - in fact this is a standard .NET class, although the syntax looks a little different to C#. If you've not used Web API before, the `RoutePrefix` + `Route` attributes identify the path that the HTTP request should come from e.g. `api/animals`. I've elected to use explicit Attribute routing here rather than set up conventions in the bootstrapper, but you could have just as easily done that, too. Notice that we've defined a standard F# record here and are exposing an F# List of Animals – nothing special there!

- Run the application again and navigate to `api/animals`. Depending on what browser you're using, the result will either be opened directly in the browser or downloaded. Using Chrome or Firefox, you'll see an output that probably looks something like this: -



The screenshot shows a browser window with the address bar containing `localhost:48213/api/animals`. The page content is an XML document:

```
<FSharpListOfAnimalvuv8Py07 xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
 <head xmlns:d2p1="http://schemas.datacontract.org/2004/07/Controllers">
 <d2p1:Name_x0040>Fido</d2p1:Name_x0040>
 <d2p1:Species_x0040>Dog</d2p1:Species_x0040>
 </head>
 <tail>
 <head xmlns:d3p1="http://schemas.datacontract.org/2004/07/Controllers">
 <d3p1:Name_x0040>Felix</d3p1:Name_x0040>
 <d3p1:Species_x0040>Cat</d3p1:Species_x0040>
 </head>
 <tail>
 <head xmlns:d4p1="http://schemas.datacontract.org/2004/07/Controllers" i:nil="true"/>
 <tail i:nil="true"/>
 </tail>
 </tail>
</FSharpListOfAnimalvuv8Py07>
```

**Figure 37.1 – Default XML output when using ASP .NET**

- Urgh – XML! That's not what we want – we'd like some JSON data, right? So, let's turn off the XML formatter. In the `RegisterWebApi` method, add the following line: -

```
config.Formatters.Remove(config.Formatters.XmlFormatter) |> ignore
```

10. Re-run the application and refresh the browser.

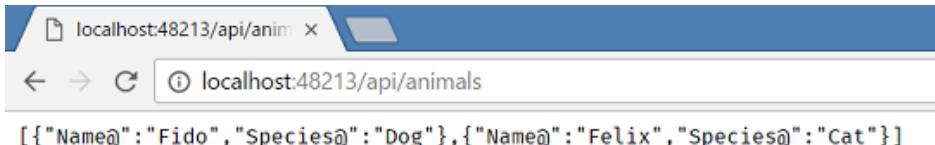


Figure 37.2 – F# records serialized using the standard JSON formatter.

11. OK – this is looking better, as we’re now getting back JSON – except all of the fields have @ post-fixed on them. This is a side-effect of how F#’s fields are compiled, and one we can easily fix by replacing ASP .NET’s JSON *formatter* with one that comes with JSON.Net. Back in RegisterWebApi, add the following line: -

```
config.Formatters.JsonFormatter.SerializerSettings.ContractResolver <-
 Newtonsoft.Json.Serialization.DefaultContractResolver()
```

12. Re-run the application and refresh the page.

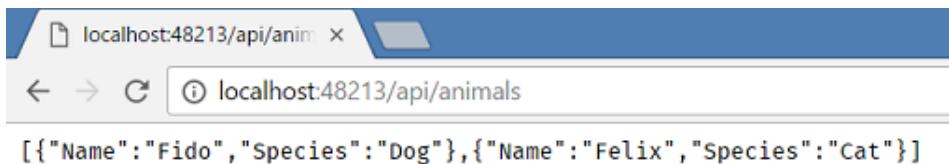


Figure 37.3 – F# records are correctly serialized using the Newtonsoft.JSON formatter

Job done! You’ve now created an ASP .NET Web API controller that uses the standard .NET web project. Remember that Web API – is not tied to C# or VB .NET, so any other feature in ASP .NET such as Filters can be used (and implemented) in F#, as well as the other features of Web API’s routing engine etc.

---

#### **CLIMutable for F# and JSON**

A common misconception in the F# community is that you must use the [`<CLIMutable>`] attribute on F# records in order to make them compatible with JSON serialization. This isn’t true, and hasn’t been for some time now - just use the `DefaultContractResolver`.

---

### **37.1.2 Using Owin Host with F#**

An alternative mechanism for working with Web API is to use the Owin Host for Web API – a lightweight host for ASP .NET that allows you to create a web host as part of any application e.g. a console application, windows service or WPF application. One of the other benefits of this approach is that you remove the reliance on custom tooling i.e. Visual Studio project

templates – there's just a standard .NET console / service and a couple of library calls, and you can more easily see what is actually happening – there's no "magic" happening behind the scenes.

### **Now you Try**

Let's now try to create an F# console app that uses Owin Host to host our Web API app.

1. Create a standard F# console application.
2. From the project properties, set the target framework to .NET 4.5.2.
3. Install the `Microsoft.AspNet.WebApi.OwinSelfHost` NuGet package.
4. Copy across the `Controllers` and `Startup` files from the web project.
5. Replace the entry point of `Program.fs` with the following. It starts up a listener on your machine, using the `Startup` class for configuration: -I

#### **Listing 37.2 – Using the Owin Host to run a web application from a Console**

```
open Microsoft.Owin.Hosting

[<EntryPoint>]
let main _ =
 use app = WebApp.Start<Startup>(url = "http://localhost:9000/") #A
 printfn "Listening on localhost:9000!"
 Console.ReadLine() |> ignore #B

0 // return an integer exit code

#A Launching the web app host within the console application
#B Blocking the console from quitting
```

6. Try hitting `localhost:9000/api/animals`. The result should be the same as before.

### **Quick Check**

1. Can you create F# web projects in Visual Studio?
2. Do you need to use `CLIMutable` to serialize records in JSON for ASP .NET?

## **37.2 Abstracting Web API from F#**

One thing about Web API is that we have to expose controllers as classes. This isn't necessarily a terrible thing, but it does go against the grain of everything we've done thus far i.e. using modules as the primary method for grouping functions together etc. So, in my experience it's not necessarily uncommon to separate out the implementation of a controller to a module, and have the controller as a simple "mapper" that is responsible for marshalling data in and out of the web app. This way, we can write our application code without worrying about HTTP response codes etc. In some ways, this is not so different when working with mixed C# / F# applications – but here, we're talking about JSON and HTTP Responses as the "output", rather than C# classes etc.

### 37.2.1 Abstracting HTTP codes from F#

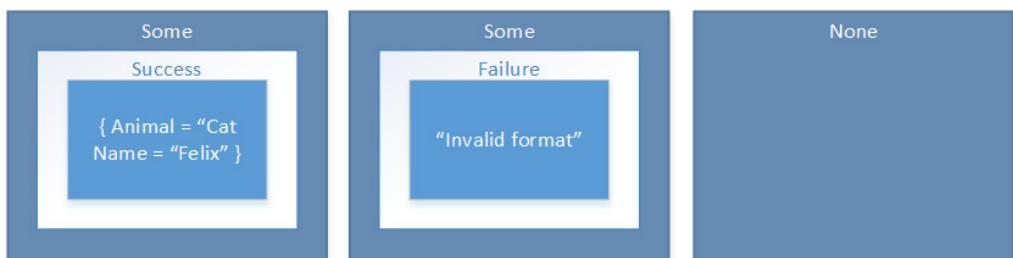
Let's take a simplistic example and consider some basic HTTP results to be one of a few different values: -

- Success with some payload (HTTP 202)
- Invalid (HTTP 400)
- Not found (HTTP 404)
- Internal error (HTTP 500)

One thing we wouldn't want to do is force our internal F# code to be polluted with HTTP codes. A much better way is to abstract this away from our internal domain – here's a sample mapping from HTTP return codes to a simple F# domain model that models the first three cases using a nested discriminated union: -

**Table 37.1 – Mapping HTTP return codes from F# types**

HTTP Code	F# Type	Example
202 (Accepted)	Some Success	Some(Success { Animal = "Cat"; Name = "Felix" })
400 (Bad Request)	Some Failure	Some(Failure "You must provide a valid name")
404 (Not Found)	None	None
500 (Internal Error)	Exception	SQL Connection Exception etc. etc.



**Figure 37.4 – Modelling both Success / Failure and “Absense of value” through with DUs**

Let's see how this maps to our real code. We'll start by abstracting away our implementation from the controller into a standalone module (I've also added a second function to try to find a specific animal by name).

#### **Listing 37.3 – Moving Web API logic to a standalone module**

```
module AnimalsRepository =
 let all =
 [{ Name = "Fido"; Species = "Dog" }
```

```

 { Name = "Felix"; Species = "Cat" }]
let getAll() = all
let getAnimal name = all |> List.tryFind(fun r -> r.Name = name)

```

The `getAnimal` function returns an `Option<Animal>`. Let's write a simple mapper function to go from this to an `HttpResponseCode` object that ASP .NET understands, and update the actual controller class as well.

#### **Listing 37.4 – Mapping between F# and HTTP domains**

```

[<AutoOpen>]
module Helpers =
 let asResponse (request:HttpRequestMessage) result = #A
 match result with
 | Some result -> request.CreateReponse(HttpStatusCode.OK, result)
 | None -> request.CreateReponse(HttpStatusCode.NotFound)

[<RoutePrefix("api")>]
type AnimalsController() =
 inherit ApiController()

 [<Route("animals/{name}")>]
 member this.Get(name) =
 AnimalsRepository.getAnimal name |> (asResponse this.Request) #B

```

#A Helper function to map from Option to HttpResponseMessage  
#B Mapping from Option<Animal> to an HttpResponseMessage

We now have a function that can take in any “result” – in our case, either `Some result` or `None`, and map it to an appropriate `HttpResponseMessage`. Notice that rather than manually creating a `HttpResponseMessage`, we’re using the `CreateResponse` method on the `Request` object that we get “for free” when inheriting from `ApiController` and passing that in (this is all created for us by the ASP .NET Controller Factory). Also, notice the explicit type annotation on the `createResponse` argument – as `HttpRequestMessage` isn’t an F# type, type inference won’t be able to figure it out for us as usual.

#### **Now you Try**

Let’s now enhance the application to handle success / failure cases more cleanly.

1. Run the application, hitting the route `api/animals/Felix` and observe that the route returns an HTTP 202 with the correct result.
2. Hit the route `api/animals/Toby` – the result will be 404 with no response payload.

So far, we’ve accounted for two of the four cases from Table 37.1. Let’s now account for the next one, HTTP 400, which is often used for e.g. validation errors (invalid request).

3. Create a new discriminated union type, `Result`. This type can store one of two cases: Either a successful payload, or some failure with a string explaining the error.

```
type Result<'T> = Success of 'T | Failure of error:string
```

4. Update the implementation of `getAnimal` so that if the name that is supplied contains any non-letters, some failure case is raised with an appropriate error. Refer back to Table 37.1 if you need help with the syntax to create a nested discriminated union i.e. `option<Result<Animal>>`.
5. You'll also have to "lift" the result of the existing call `tryFind` into a `Success` case so that the types of both cases match.
6. Update the `asReponse` function so that it now caters for `Some Failure` and `Some Success` cases instead of just `Some`.
7. Run the application and test that all three cases are working.

Since Web API automatically cascades exceptions to response codes of 500, we don't actually have to do anything there. But you can test it out by adding logic to your `getAnimal` function so that if it takes in some special string e.g. "FAIL", it raises an exception using the `failwith` function.

### **Error Handling in F#**

Just to reiterate what was mentioned earlier in the book – it's not always considered best practice to include exceptions as a "standard" part of your application. Instead, use something like `Result` to raise failure cases that you can account for – this way, they're included in the type system and you can more easily reason about them through pattern matching etc. Again, Scott Wlaschin's excellent series on Railway Oriented Programming is an in-depth look at how to model failure cases, as well as how to reason about them succinctly.

### **Quick Check**

3. Why would you create a layer of abstraction from your application code and HTTP request codes?
4. How might you model the no result (404) within an F# application?

## **37.3 Working with Async**

Web API has native support for working with Tasks. This means that your controllers can return data *asynchronously* (and therefore wrapped as `Task<T>`) and Web API will happily "unwrap" this automatically in the background for you. Let's just now show how we can work with asynchronous data in F# using F#'s `Async` type, and yet still interoperate with Web API's support for Tasks.

1. Wrap the implementation of `getAnimal` in an `async { }` block (remember to explicitly `return` the result of the match expression!). Obviously, in this example there is no real asynchrony occurring – we're just pushing this work to a background thread – but the principle is the same.

2. The controller method will break, because we're trying to push an `Async<Option<Result<Animal>>>` into `asResponse`, which expects an `Option<Result<Animal>>` - we need to "unwrap" the async somehow. The simple thing to do here would be to call `Async.RunSynchronously` - but this defeats the whole purpose of using asynchronous code within a web app. Instead, we can make the controller method itself async: -

#### Listing 37.4 – Creating an asynchronous Web API controller method

```
[<Route("animals/{name}")>]
member this.Get(name) =
 async { #A
 let! result = AnimalsRepository.getAnimal name #B
 return result |> asResponse this.Request.CreateResponse #C
 }

#A Creating an async block
#B Asynchronously “awaiting” the result of getAnimal using let!
#C Returning the result.
```

3. Notice that the result of the controller method is now an `Async<HttpResponseMessage>`.
4. We're still not there - Web API doesn't know how to work with F# Async blocks – only Tasks. Run the application and try to access the route – you'll always get back an HTTP 200 with an empty payload.
5. However, Web API *can* natively “unwrap” a `Task<T>`. So, convert the `async` to a `Task` by pipelining the `async` workflow that's returned by the `Get(name)` method into `Async.StartAsTask`. The result of the controller method will now be `Task<HttpResponseMessage>`.
6. Re-run the application and observe that the results are once again correct.



Figure 37.5 – Moving from an `F# Async<T>` to `Task<T>` as needed by ASP .NET

Your code now will run entirely asynchronously, and yet still play nicely with the Web API framework's support for Task.

#### **Quick Check**

5. How do we return F# results wrapped in async blocks over ASP .NET?

## 37.4 Introducing Suave

Suave is an F#-first web library designed to allow us to model our web applications using an entirely functional-first model. It's very lightweight, and once you get your head around the concepts it introduces (which, admittedly, are different from those you'll know from ASP .NET) you can rapidly create powerful applications in a very lightweight fashion.

### 37.4.1 Modelling web requests as functions

- One thing about ASP .NET is that it (unsurprisingly) pushes us into an object-oriented model – we have controllers that inherit from base controllers; controllers are objects with methods on them etc. etc. In reality, web applications are a great fit for functional programming and F#: -
- Web apps are by their very nature nearly always stateless – we take an HTTP request in, and give back an HTTP response.
- Web apps often need to make use of asynchronous programming, which F# has excellent support for.
- Web apps – particularly the back end – are very often data-centric; another great fit for F#.

Suave takes a very different approach to ASP .NET – every request is handled by a "web part", which itself is built up of other, smaller web parts. A web part is a function with a simple signature: -

```
HttpContext -> Async<HttpContext option>
```

In other words, given an http context (essentially, details on the request / response), a web part must *asynchronously* return either: -

- *Some* context – typically the context that was passed in, perhaps along with an updated response payload
- *Nothing*

A web part can be composed together from other, smaller web-parts, each working together to create a pipeline that builds up to a final response. In a way, you can think of these as somewhat similar to filters, or visitors. For example, one web part may convert an object to JSON. Another might check if the supplied route is a GET or POST request etc. Here's a simple Suave pipeline that mirrors our existing Web API controller: -

#### **Listing 37.5 – A simple Suave pipeline**

```
GET >=> #A
choose [#B
 path "/api/animals" >=> (AnimalsRepository.getAll |> asJson) #C
 pathScan "/api/animals/%s" getAnimal] #D
```

#A Only match requests that are of type GET

#B Choose the first matching handler in this list

#C If the path matches, get all animals and return as JSON  
#D If the path matches, try to get the animal, supplying the animal name taken from the route

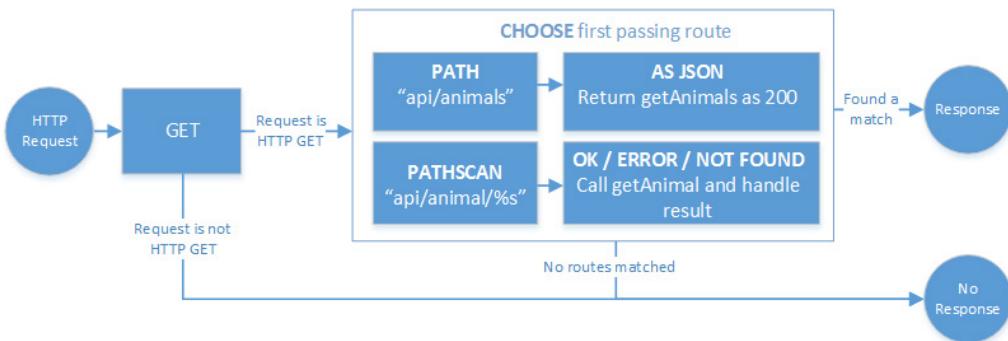


Figure 37.6 – A simple Suave pipeline

Table 37.1 – Common Suave Web Parts

Web Part	Description
<b>OK</b>	Returns a string response as HTTP 200
<b>GET</b>	Checks the incoming request. If it's a GET, it passes the context along, otherwise returns nothing.
<b>CHOOSE</b>	Takes in several web parts, trying each of them sequentially. The first one to return a valid response is returned, otherwise returns nothing.
<b>PATH</b>	Scans the request URI. If it matches the specified URL, it passes the context along, otherwise returns nothing.

This model of composing web-parts is very powerful, as you can rapidly build up arbitrary pipelines based on small, reusable functions. However, the syntax and model is very different to what you're probably used to – for example, there's a custom `>=` operator that's used to “connect” to web parts together, as well as the underlying nature of Suave – basically, a purely functional pipeline.

What is very nice about Suave is that it's extremely lightweight – there's no code-gen needed, nor is there any requirement for custom projects. It can live as a console application or service and is easy to start – just a single line of code, in fact: `startWebServer defaultConfig app`, where `app` is your overall composed webpart. There's a full example in the code sample that shows how you might adapt the existing code into Suave. If you like what you see, it's worth checking out some of the excellent resources (both online and paper-based) to learn more about it.

### **Quick Check**

6. What is a web part in Suave?
7. What does Suave's GET web part do?

## **37.5 Summary**

That's a wrap for creating web applications in F#! We saw: -

- How to create ASP .NET Web API applications in F# and Visual Studio
- Getting F# to work smoothly with ASP .NET
- Marshalling data between F# and ASP .NET domains
- The Suave F# web programming library

### **Try This**

Enhance the sample application so that it handles POST requests as well as just GETs. Alternatively, create a web application in F# using ASP .NET that serves us data sourced from the `FootballResults.csv` file in the `data` folder from the source code repository. Try to use the CSV Type Provider as the data access layer!

Enhance the sample application so that it handles POST requests as well as just GETs

### **Quick Check Answers**

1. Yes, but only via the third-party F# Web templates.
2. Not when using Newtonsoft's standard JSON serializer.
3. To protect your business logic code from HTTP results; making use of richer F# types e.g. Option, Result.
4. Through the Option type i.e. Some / None.
5. Convert from Async to Task via the `Async.StartAsStart` combinator.
6. A function which takes in an HTTP context and may return a new context, asynchronously.
7. If the request is an HTTP GET, passes the context back out, otherwise returns nothing.

# 38

## *Consuming HTTP data*

**Having seen how to serve HTTP data up using ASP .NET in F#, we're now going to look at the other side of the fence – consuming HTTP data quickly and easily using a number of F# libraries.fs. We'll see: -**

- How you might access HTTP endpoints today
- Using FSharp.Data to work with HTTP endpoints
- HTTP.fs, a lightweight F# library for HTTP access
- The Swagger Type Provider

Let's start by quickly considering a few common situations you might encounter today.

You've written an ASP .NET Web API application and deploy it to an environment ready for testing. Quickly you get back some feedback from the test team with some issues – some data doesn't match as expected. And under some circumstances, the APIs don't seem to respond at all. You need to set up some way of easily exploring these cases as they come – possibly even in a repeatable way.

Here's another one. You need to consume an HTTP endpoint from an external supplier. The problem is that there's no SDK to consume their API – you need to do it all yourself. The API is complex – you need to first authenticate with a time-limited token, before exploring the API (we won't cover this in detail in this lesson, but there's a sample of how to do this in the code samples for this lesson!).

What options do you have for these sorts of situations? One is to use a tool like Fiddler or Postman. These are either dedicated Windows or web applications that allow you to test out endpoints by sending example request payloads to an endpoint, getting the response back in the browser and analysing the responses. However, this isn't necessarily the most effective tool – you have to leave your development IDE and context switch to another tool. And, if you want to "do" anything with the response, you need to copy the information manually into an IDE to experiment with the data. To get around this, another option is to go straight to the

`System.Net.WebClient` (or related classes) to start hitting the endpoints directly. Doing this is often painful to do – particularly with a console application etc.

F# provides a number of libraries for working with HTTP data quickly and easily – in this lesson, we'll explore three different options, each with different benefits.

## 38.1 Using FSharp.Data to work with HTTP endpoints

FSharp.Data's JSON type provider works very well at consuming HTTP endpoints that expose JSON data quickly and easily.

### Now you Try

Let's test this out using the HTTP API that we created in the previous lesson.

1. Open the Web API project from the previous lesson and start the Web API endpoint up. Note that the following code samples will assume a port of 8080, but you can find the correct port from the Web tab of the project's properties pane.

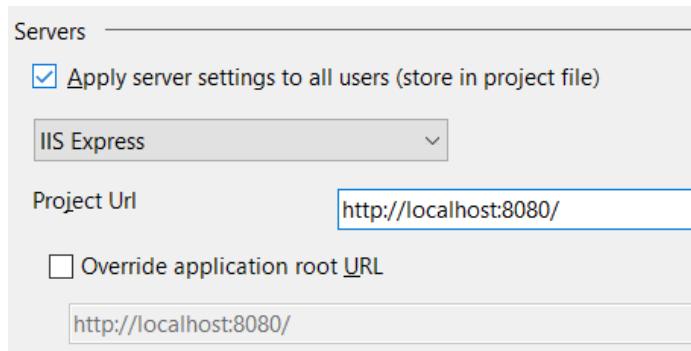


Figure 38.1 – Identifying the host uri of IIS Express for Web API in Visual Studio 2015

2. Create a new fsx script and reference the FSharp.Data nuget package.
3. Enter the following code to retrieve the names of all animals.

### Listing 38.1 – Using FSharp.Data to access a HTTP endpoint

```
#I @"..\..\..\packages"
#r @"FSharp.Data\lib\net40\FSharp.Data.dll"
open FSharp.Data
type AllAnimalsResponse =
 JsonProvider<"http://localhost:8080/api/animals"> #A
let names =
 AllAnimalsResponse.GetSamples() #B
 |> Seq.map(fun a -> a.Name) #C
 |> Seq.toArray
```

#A Creating a type that matches the api/animals route

```
#B Retrieving all animals
#C Accessing the JSON in a strongly-typed fashion
```

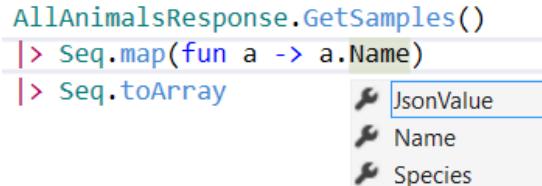


Figure 38.2 – Intellisense from FSharp.Data and an HTTP route.

As you can see, we're able to get intellisense over the payload of a route as it's simply JSON (this is actually very similar to what we did in the Data unit when accessing the NuGet API). We can also easily make a parameterizable function to get a specific animal.

4. Enter the following code

#### **Listing 38.2 – Creating a parameterizable function to access a route**

```
type GetAnimalResponse =
 JsonProvider<"http://localhost:8080/api/animals/Felix"> #A
let getAnimal =
 sprintf "http://localhost:8080/api/animals/%s"
 >> GetAnimalResponse.Load #B
getAnimal "Felix" #C

#A Creating a type that matches the parameterised Animals route
#B Creating a simple function to call the API
#C Calling the function for a specific animal
```

Unfortunately, using FSharp.Data does have a few restrictions – firstly, it doesn't give you *total* control over things like HTTP headers and the like, and secondly, it only works with JSON data (obviously). Lastly, it doesn't provide any in-built way of handling different response codes e.g. 400, 404 or 500 - you'll need to write some code that can wrap any call within a try/with block and convert into a discriminated union e.g. `Result<'T>` e.g.

#### **Listing 38.3 – Writing a simple try / with converter to Result**

```
type Result<'TSuccess> = Success of 'TSuccess | Failure of exn #A
let offFunc code = #B
 try code() |> Success
 with | ex -> Failure ex
let getAnimalSafe animal =
 (fun () ->
 sprintf "http://localhost:8080/api/animals/%s" animal
 |> GetAnimalResponse.Load)
```

```
|> offunc #C
let frodo = getAnimalSafe "frodo" #D

#A A Result discriminated union
#B Combinator function to wrap any code in a try / with block and convert to Result
#C Wrapping the GetAnimalResponse in Result
#D Safely calling the route with a non-existent animal
```

FSharp.Data is a lightweight, quick and easy way to start consuming JSON data enabled over HTTP. However, it doesn't contain any mechanisms for discovering routes – you need to know them already – and you need to create a separate type for each type of data exposed.

### Quick Check

1. Why might you use FSharp.Data to consume a JSON resource over HTTP?
2. What limitations does FSharp.Data have with working with HTTP resources?

## 38.2 HTTP.fs

HTTP.fs is a small, general-purpose nuget packages for working with data over HTTP, and allows us close control over both creating requests and processing responses.

---

### Http.fs versions

Newer versions of HTTP.fs have slightly increased the complexity of the library by introducing a dependency on other third-party libraries. In the interests of keeping things simple, we'll be using a slightly older version of the library (1.5.1) which has a smaller API surface area and is therefore – in my opinion – easier to work with.

---

### 38.2.1 Building requests as a pipeline

Http.fs follows a standard pattern we've already seen when working with immutable data and composable functions – we build a request by chaining together small functions which modify a request in some way: -

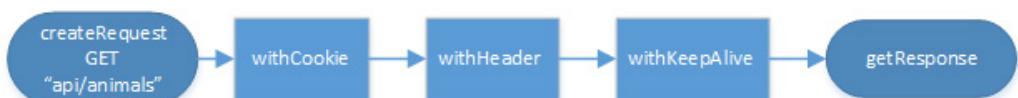


Figure 38.3 – Creating an HTTP request by chaining small functions together

In the fictitious Figure 38.3, we're creating a small pipeline to build an HTTP request – first, we create a basic GET request, before modifying the request in several ways before calling `getResponse`, which actually fires off the request and returns the data back. Here's how that looks in code: -

**Listing 38.4 – Mapping an HTTP composition pipeline in HTTP.fs**

```
createRequest Get "http://host/api/animals"
|> withCookie { name = "Foo"; value = "Bar" }
|> withHeader (ContentType "test/json")
|> withKeepAlive true
|> getResponse
```

As you can see, the code maps very closely to the diagram. We could use this mechanism to construct any sort of HTTP message – with security tokens as headers, or custom cookies etc. Unlike FSharp.Data, Http.fs returns back the *optional* response (as raw text) as part of an object that also contains any headers, cookies, the content length and the response code.

**Now You Try**

Let's test out Http.fs against our running web api.

1. Add a reference to Http.fs 1.5.1 (if you're using the packages included in the learn-fsharp github repository, it'll already be in the packages folder) and open the HttpClient namespace.
2. Create a request to the Animals endpoints: -

**Listing 38.5 – Creating our first request with Http.fs**

```
#r @"Http.fs\lib\net40\HttpClient.dll"
open HttpClient
let request = createRequest Get "http://localhost:8080/api/animals"
let response = request |> getResponse
```

The request object contains all the details needed for an HTTP request – the URI to hit, any cookies, headers, a body for POSTs, the type of request etc. etc. When we send this to the web server using `getResponse`, we get back the following: -

**Listing 38.6 – Getting a response from ASP .NET Web API with Http.fs**

```
{StatusCode = 200; #A
EntityBody = #B
Some "[{"Name":"Fido","Species":"Dog"}, {"Name":"Felix","Species":"Cat"}]";
ContentLength = 66L;
Cookies = map [];
Headers =
Map #C
[((ContentTypeResponse, "application/json; charset=utf-8");
 (DateResponse, "Mon, 16 Jan 2017 15:34:02 GMT");
 (Server, "Microsoft-IIS/10.0"); (NonStandard "X-Powered-By", "ASP.NET");
 (NonStandard "X-SourceFiles", <elided>)]);}

#A Status code of 200
#B Response body is a string option
#C Any headers from the server are provided
```

Notice that the `EntityBody` property is an `Option<string>` to cater for when there is no valid payload returned. Let's continue by working to calling the "named animal" route. Remember from the previous lesson that this endpoint might return a valid animal if one was found, or none, or an HTTP 400 if the name contains numbers.

3. Now, add a reference to `Newtonsoft.Json` and create some helper functions.

```
open Newtonsoft.Json
let buildRoute = sprintf "http://localhost:8080/api/%s" #A
let httpGetResponse = buildRoute >> createRequest Get >> getResponse #B

#A Helper function to build a route to our API
#B Creating a composed function to go from a string to an Http.fs response
```

Now we have that helper function, we can easily write some API wrappers. Here's a simple one that gets the response, and if there's a payload, converts it to an F# record.

#### **Listing 38.7 – Creating a wrapper API function with `Http.fs` and `Newtonsoft JSON`**

```
type Animal = { Name : string; Species : string }
let tryGetAnimal animal =
 let response = sprintf "animals/%s" animal |> httpGetResponse #A
 response.EntityBody #B
 |> Option.map(fun body -> JsonConvert.DeserializeObject<Animal>(body)) #C

#A Getting the response from Web API via Http.fs
#B Checking the entity body has a value
#C Converting it with Newtonsoft.JSON
```

It's very easy to build a simple DSL around your own custom routes to allow you to create scripts (or even full APIs) around them. And because the `Http.fs` is a simple library, and not a framework, you're free to use it however you want. For example, we could easily use the provided type that we generated in Listing 38.1 to handle deserialization rather than an explicit F# record and `Newtonsoft.Json`. However, the cost of this flexibility is that you have to usually create some wrapper façade around `Http.fs` – you wouldn't want to expose the outputs from a low-level library like this to callers. There's also no provided types here – you'll have to either create your own types, or share types across both server and client – or use something like the JSON type provider to give you types based on sample JSON.

#### **Quick Check**

3. What property is exposed by `Http.fs` that contains the response body?
4. How are errors exposed by `Http.fs`?

### **38.3 The Swagger Type Provider**

Swagger is rapidly becoming a standard for providing schematised metadata HTTP APIs (as well as an interactive browser application). It provides a web-front end as well as a

programmatic API for working with web apis – In other words, it's a service to provide information *about other services!* Most languages and runtimes provide a Swagger generator, and .NET is no different, with the Swashbuckle package. Here's what a Swagger endpoint looks like for our Animal web service: -

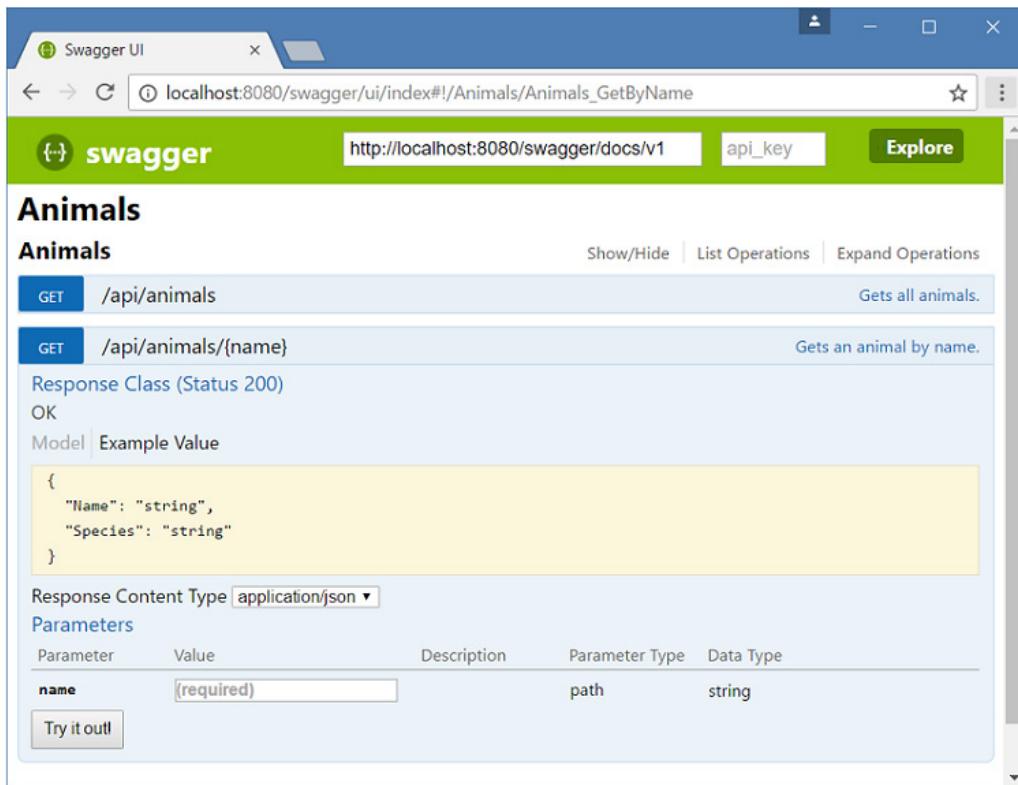


Figure 38.4 – Viewing the Swagger-generated API documentation.

### 38.3.1 Adding Swagger to ASP .NET Web API 2

To save you some time, I've made some small modifications to the Web API project from Lesson 37 – you'll find the modified version in the `lesson-38` folder. In essence, I've added the Swashbuckle NuGet package to the project, fixed the binding redirects that NuGet forgot to do, and then made several small changes to the codebase. Firstly, I've added Swagger support to the ASP .NET pipeline so that it runs on startup: -

**Listing 38.8 – Activating Swagger within ASP .NET Web API**

```
open Swashbuckle.Application
config
 .EnableSwagger(fun config -> #A
 let path = sprintf "%s\bin\FSharpWeb.XML"
 System.AppDomain.CurrentDomain.BaseDirectory
 config.IncludeXmlComments path #B
 config.SingleApiVersion("v1", "Animals") |> ignore)
 .EnableSwaggerUi() |> ignore #C
```

#A Enabling Swagger documentation generation  
#B Using XML code comments to generate API docs  
#C Turning on the Swagger GUI

Having turned on XML comments support, I applied some comments and attributes to the HTTP routes so that they document what they do: -

```
[<Route("animals/{name}")>]
[<ResponseType(typeof<Animal>)>]
/// Gets an animal by name.
member this.GetByName(name) =
```

Figure 38.5 – Decorating a route with the [`<ResponseType>`] attribute and XML triple-slash comments

Now that that's all working, you can launch the application and navigate to the Swagger docs endpoint

(e.g. `http://localhost:8080/swagger`) You'll now have a full web UI showing us the different API methods we've surfaced, along with our XML comments on the right-hand-side as well as an example response document. Neat!

### 38.3.2 Consuming Swagger APIs in F#

#### **Now You Try**

Swagger on its own is great, but even better is that there's a Swagger Type Provider for F# that can generate a *full api* from the Swagger endpoint - let's try it out.

1. In your script, add `#r` references to `YamlDotNet` and both `SwaggerProvider` and `SwaggerProvider.Runtime` assemblies in `SwaggerProvider`.
2. Enter the following code to create a typed instance of the Swagger provider: -

**Listing 38.9 – Hooking up the Swagger Type Provider to a Swagger endpoint**

```
#I @"..\..\..\packages"
#r "YamlDotNet/lib/net35/YamlDotNet.dll"
#r "SwaggerProvider/lib/net45/SwaggerProvider.dll"
#r "SwaggerProvider/lib/net45/SwaggerProvider.Runtime.dll"
```

```

open SwaggerProvider
type SwaggerAnimals =
 SwaggerProvider<"http://localhost:8080/swagger/docs/v1"> #A
let animalsApi = SwaggerAnimals() #B

#A Generating an API from the Swagger endpoint
#B Creating an instance of our API

```

Now, we can simply dot into `animalsApi` and get access to all the methods we've exposed. And not only that, but the full response types will also be generated for us: -

The screenshot shows an F# code editor with the following code:

```

animalsApi.AnimalsGetByName("Felix")

```

A tooltip for the `AnimalsGetByName` method is displayed, containing the following information:

`SwaggerAnimals.AnimalsGetByName(name: string) : SwaggerAnimals.Animal`

Gets an animal by name.

Figure 38.6 – Consuming a Swagger endpoint using the Swagger Type Provider

As you can see here, we also get generated comments based on the source XML comments – even at the field level: -

The screenshot shows an F# code editor with the following code:

```

animal.

```

A tooltip for the `Name` field is displayed, containing the following information:

`property SwaggerAnimals.Animal.Name: string`

The name of the animal e.g. Felix, Fido.

Figure 38.7 – XML comments are generated by the Swagger type provider for fields as well as methods.

This automatic API and type generation makes it incredibly easy to start consuming web-enabled APIs in F# - you get automatic route discovery and full type generation. However, this puts a requirement on the caller to expose accurate API schema information via Swagger.

### Quick Check

5. What is Swagger?
6. What NuGet package is used in order to create Swagger documentation on .NET?

## 38.4 Summary

That concludes the “consuming” side of the HTTP unit. Let’s quickly review what we covered this lesson.

- We saw how to use `FSharp.Data` to quickly and easily consume JSON-ready HTTP

endpoints.

- We then worked with the `Http.fs` package, a low-level package that gives you full control over sending HTTP requests to web servers.
- Lastly, we looked at the Swagger Type Provider, which provides you with full access to a generated HTTP client based on Swagger metadata.

### **Try This**

Enhance the error handler in 38.1 so that failures have specific cases instead of just an exception e.g. `PageNotFound` | `InternalServerError` of `exn` | `BadRequest` etc.

Then, enhance the `tryGetAnimal` function in 38.7 to check the `StatusCode` and emit logging if an HTTP 400 or 500 response is received.

Finally, try to replace the explicit F# record using in 38.7 with the provided type created in 38.1.

### **Quick Check Answers**

1. `FSharp.Data` is extremely quick and easy to use, and works with all JSON web services.
2. Only works with JSON services; no error handling built-in.
3. `EntityBody`.
4. The `StatusCodes` property – no exceptions are for server-generated errors.
5. A standard for exposing schematized metadata on a web API.
6. `Swashbuckle`.

# 39

## *Capstone 7*

**The last lesson of the Web Programming unit will apply the lessons we've learned here to our bank account solution that we've been working on.**

### **39.1 Defining the problem**

In this capstone, we'll first make our application web-enabled: we'll add a Web API layer on top, before consuming it via a script over HTTP. We'll also discuss how you could make your WPF application work over HTTP as well. Lastly, we'll make our bank account use asynchronous data access.

#### **39.1.1 Solution Overview**

At the end of the last capstone, we had replaced our file system with a SQL database for storing transactions, along with a WPF application for the GUI. In this lesson, I've removed the GUI element completely (don't worry, you could plug it back in again without too much difficulty) as well as the database project (the scheme hasn't changed at all). What I've introduced instead is an ASP .NET Web API project which we'll use to provide an API for the bank application. I've done all of the work to add the correct NuGet packages and set the binding redirects up already, so you can focus on the application coding.

### **39.2 Adding Web API support to our application**

We'll start by making a controller for our bank account application. It should have endpoints reflecting the different functions exposed by the BankAPI interface – getting details on an account, getting the transaction history, withdrawing funds and depositing funds.

#### **39.2.1 Our first endpoint**

Let's start by creating a controller in the `Controllers.fs` file: -

### Listing 39.1 – A basic Web API controller for the Bank API

```
[<RoutePrefix("api")>]
type BankAccountController() =
 inherit ApiController() #A
 let bankApi =
 let conn = ConfigurationManager.ConnectionStrings.["AccountsDb"]
 CreateSqlApi(conn.ConnectionString) #B
 []
 member __.GetAccount(name) = #C
 let account = bankApi.LoadAccount { Name = name }
 match account with
 | InCredit (CreditAccount account) -> account
 | Overdrawn account -> account

#A Creating a basic ASP .NET controller
#B Creating an instance of our SQL-enabled Bank API
#C Our first route to retrieve basic account details
```

That's all that's needed! Notice how we're "unwrapping" the result of the load account so that we simply return the "raw" account object rather than the full discriminated union – this keeps things a little simpler in terms of the JSON that we expose (by the way – I'm assuming here that you still have the SQL database from the previous capstone – if not, go back and create it, or replace the call to create the SQL layer with the `FileApi`. However, doing this will stop you doing some of the later exercises in this lesson).

#### Dependency injection with F#

If you've been using ASP .NET for a while now, you'll probably have immediately spotted that we've tightly coupled the controller to the SQL API implementation – shouldn't we be injecting this dependency into the controller based on the interface instead? Well, you can definitely do that – remember, F# can create interfaces just like C#. You'll need to do create an implementation of an `IDependencyResolver` and assign it to the `DependencyResolver` property on the `HttpConfiguration` object.

You should now be able to run the web app, open a browser and navigate to a URL similar to `http://localhost:8080/api/accounts/isaac`. You'll get back a response such as this:

```
{"AccountId": "8e7a7909-5667-4cfb-8726-ac3c083ea621", "Owner": {"Name": "isaac"}, "Balance": -3.0}
```

That was easy! You should now move along and create the transaction history route, `GetHistory` (I've used the route `transactions/{name}`). Again, test it out in a browser – notice that transaction is serialized as follows: –

```
{"Timestamp": "2016-12-30T17:02:53.757", "Operation": {"Case": "Deposit"}, "Amount": 10.0}
```

Observe how the `Operation`, rather than being simply "Deposit", is actually a full JSON object with a property called `Case`. In such situations, you can either look to replace the JSON serializers with another one, or create a "JSON friendly" type – a simple F# record with no discriminated unions – and map from the complex F# type to the dumber JSON-friendly

record. This has the added benefit that you de-couple your internal domain with your public, user-facing contracts.

### Discriminated Unions over JSON

Newtonsoft JSON will happily serialize and deserialize F# discriminated unions for free. However, the JSON that emits is not the most idiomatic out there, as you've just seen. There are a couple of alternative serializers (such as FifteenBelow.Json) that can plug into Newtonsoft to override the serialization for F# types, and they do a much better job of creating idiomatic JSON from F# types.

### 39.2.2 Posting data to Web API

Let's now create handlers for Deposit and Withdrawals. You'll have noticed that we've prefixed both routes so far with the word Get. This convention tells ASP .NET that those methods should be bound to HTTP GET requests. Similarly, prefixing a method with Post tells ASP .NET to bind a method to HTTP POST requests (You can explicitly state this with the [`<HTTPPost>`] attribute as well). As Deposit and Withdrawal are functions that save new data (rather than just *reading* data), we'll mark them as Post methods. Here's an example to get you started: -

#### **Listing 39.2 – Example routes for POSTing a Deposit request**

```
type TransactionRequest = { Amount : decimal } #A
[<Route("transactions/deposit/{name}")>] #B
member __.PostDeposit(name, request : TransactionRequest) = #C

#A Creating a type to hold the POST request payload
#B Creating a custom route for deposit
#C Binding routing and POST payload data to a controller method.
```

Make sure that you also remember to "unwrap" the `RatedAccount` to a raw `Account`, as we did with the initial controller method (you may want to create a reusable function that does this and call it from all three controller methods).

It's not so easy to test out these POST methods in a browser – so let's expose this API over Swagger and then write a script to access the API programmatically.

## 39.3 Consuming data with Swagger

There are a couple of things we'll need to do to expose the data over Swagger. Firstly, we need to "turn on" Swagger in the Web API pipeline, and then we'll add some metadata to make Swagger a little easier to work with.

### 39.3.1 Activating Swagger

To turn on Swagger, we simply need to hook into the configuration phase of Web API, just as we did earlier in this unit in the `Configuration` method of the `Startup` class.

### Listing 39.3 – Activating Swagger in ASP .NET Web API

```
config
 .EnableSwagger(fun config =>
 let path =
 sprintf @"%s\bin\Web.XML"
 System.AppDomain.CurrentDomain.BaseDirectory
 config.IncludeXmlComments path
 config.SingleApiVersion("v1", "Bank Accounts") |> ignore)
 .EnableSwaggerUi() |> ignore #A
```

#A Turning on Swagger with support for XML comments

Re-run the application; navigating to <http://localhost:8080/swagger/ui/index#/> should now present you with a documented API as follows: -

The screenshot shows the Swagger UI interface for a BankAccount API. At the top, it says "Swagger UI" and has a URL bar with "localhost:8080/swagger/ui/index#/!/\_BankAccount/". Below that, there's a title "BankAccount" followed by four API endpoints listed in a table:

<b>GET</b>	/api/accounts/{name}
<b>GET</b>	/api/transactions/{name}
<b>POST</b>	/api/transactions/deposit/{name}
<b>POST</b>	/api/transactions/withdraw/{name}

Below the table, there's a section titled "Response Class (Status 200)" which contains "OK". At the bottom left, there are tabs for "Model" and "Example Value".

Figure 39.1 – A swagger API for our Bank Account API

#### 39.3.2 Applying metadata

Now, let's improve the API a little by adding the following: -

- XML comments.** Apply triple-slash XML comments over any methods or data that are publicly exposed. This includes controller methods and any types e.g. Transaction Request, Account etc. etc. (you can comment on both F# records and their fields).
- Response Types.** Place the [`<ResponseType>`] attribute on all controller methods, stating the type that they return e.g `[<ResponseType(typeof<Account>)>]`.
- Mandatory metadata.** Swashbuckle generates Swagger metadata where most fields are marked as optional by default. The problem is that the Swagger Type Provider will

process this metadata and generate optional types for us. This is a pity - the default for Swashbuckle should really be that fields are marked as mandatory by default. Nonetheless, we can override this by placing the [`<System.ComponentModel.DataAnnotations.Required>`] attribute on any fields that are mandatory. For now, just place it on the `TransportRequest`'s `Amount` field, but you might want to also place it on decimal fields that are exposed by the core Bank domain (or alternatively, to avoid polluting the internal domain, create "public" types that live exclusively in your Web API domain and map across to them).

### 39.3.3 Consuming our API

Now we're ready to consume our API in F#. Open up a new script and add references to Swagger as follows: -

#### **Listing 39.4 – Consuming the Bank Accounts API via the Swagger Type Provider**

```
#I @"..\..\..\packages"
#r "YamlDotNet/lib/net35/YamlDotNet.dll"
#r "SwaggerProvider/lib/net45/SwaggerProvider.dll"
#r "SwaggerProvider/lib/net45/SwaggerProvider.Runtime.dll" #A

open SwaggerProvider
type BankApi = SwaggerProvider<"http://localhost:8080/swagger/docs/v1"> #B
let bankApi = BankApi() #C

#A Adding references to required assemblies
#B Connecting to the Swagger endpoint
#C Creating an instance of the Swagger client.
```

Now we're good to go! You can now call methods on the provided client to call methods in our HTTP API: -

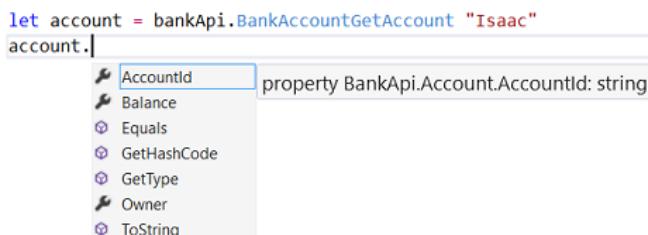


Figure 39.2 – Strongly-typed access to an ASP .NET API

Think about that for a moment. You're calling an HTTP API, with a full strongly-typed domain, in six lines of code (and no code generation required). That API is itself performing some strongly-typed business logic, which itself calls a pluggable data access layer (which in this

case goes to a SQL database). In all, this needed perhaps just a few hundred lines of code – not bad at all!

## 39.4 Enriching the API

Let's look at two final areas in this capstone – working with results, and async data.

### 39.4.1 Working with Results

The eagle-eyed among you will have spotted that for the Withdraw method, we return a successful message even if the withdrawal fails i.e. the user was already overdrawn. The problem is that the Bank API doesn't tell us if a Withdrawal was a success or not – it simply returns back either the new account if the withdrawal succeeded, or the original account if it was rejected.

#### **Listing 39.5 – The existing Withdraw function**

```
member this.Withdraw amount customer : RatedAccount =
 let account = this.LoadAccount customer
 match account with
 | InCredit (CreditAccount account as creditAccount) -> #A
 auditAs Withdraw saveTransaction withdraw amount creditAccount account.AccountId
 account.Owner
 | Overdrawn _ -> account } #B

#A An InCredit account is allowed to withdraw funds. New account state is returned.
#B An Overdrawn account cannot withdraw funds. The existing account state is returned.
```

Notice that the return type is simply a RatedAccount – here's our problem. Let's change it so that the return type of `Result<RatedAccount>` - we'll return Success with the account when the account was in credit, and a failure with an error message if it was overdrawn. Create a Result type, and then change the logic so that we either return wrap the result of `auditAs` in a Success, or a Failure with the message "Account is overdrawn – withdrawal rejected!". Here's a Result type definition to get you going: -

```
type Result<'T> = Success of 'T | Failure of string
```

In the controller, we now expose either an OK or Failure for Success or Failure: -

#### **Listing 39.6 – Surfacing errors from our Domain as HTTP codes**

```
[<Route("transactions/withdraw/{name}")>]
member this.PostWithdraw(name, request : TransactionRequest) =
 let customer = { Name = name }
 match bankApi.Withdraw request.Amount customer with
 | Success account -> this.Ok(account) :> IHttpActionResult #A
 | Failure message -> this.BadRequest(message) :> IHttpActionResult #B
```

#A Returning OK for successful withdrawals

#B Returns BadRequest for failed withdrawals

The only slight fly in the ointment is that we need to “safe upcast” from `OK` and `BadRequest` to `IHttpActionResult` so that both branches return the same type – F# won’t implicitly upcast for us here. Quick tip – once you’ve done the explicit upcast for the first branch, you can simplify the second from `:> IHttpActionResult` to just `:> _`.

### 39.4.2 Making the API asynchronous

Let’s finally spend a little bit of time looking at what’s involved in making the application asynchronous. The best place to start is at the bottom of the stack, and work upwards – in our case, this is in the SQL data layer. The only method we can truly make `async` is `getAccountAndTransactions` – the `writeTransactions` function uses `DataTable.Update` which doesn’t support `async`. You could of course wrap it in an `async` block anyway, but you’ll just push the work onto another thread.

You’ll first need to replace the `Execute()` calls to the database with `AsyncExecute()` calls instead. This will then cause *everything* to break, because these calls now return `Async<T>` instead of just `T`. To fix it, you’ll need to “unwrap” from `Async<T>` to `T` using the `let!` keyword, and to do that, you need to wrap your code in an `async { }` block. Also, remember to explicitly `return` values e.g. `return Some(accountId, transactions)`.

Once you have that fixed, you’ll see a “cascading” effect up the stack – all the calls will also need to be made “`async`” friendly, just like in C# with `async / await`. So, the `IBankApi` will need to be updated so that all the methods return `Async` data, and the implementations of those methods will also need to be made `Async` aware so that you can `let!` them as needed.

Note that the file system-based API won’t be compatible with the `BankApi` anymore – you’ll need to “lift” those functions so that they’re also `async`. This is easily accomplished: -

```
FileRepository.tryFindTransactionsOnDisk >> async.Return
```

`async.Return` simply takes an value and wraps it in an `Async`. In other words, whatever the result of the function on the left is, push it into `async.Return` and give that result back out.

There’s one, final area to fix – the controllers themselves. Be careful that you don’t accidentally `return Async<T>` values out – you’ll need to convert them to `Task<T>` using `Async.StartAsTask`. You’ll also find one last problem which is a bit of an F# oddity – you can’t call protected class members (such as `this.OK` or `this.BadRequest`) directly within lambdas or `async` blocks. So, some of your code will probably break with a lengthy error message, the key part of which is this: -

```
Protected members may only be accessed from an extending type and cannot be accessed from inner lambda expressions.
```

You might ask, where’s the lambda expression? The answer is that `async` blocks (indeed, all computation expressions) are just syntactic sugar to rewrite code as continuations, which are effectively lambda expressions. It turns out that there’s a simple, rote solution to this, which is simply to make a member that *explicitly* calls the protected member manually, and then call that member instead: -

**Listing 39.7 – Working around F# restrictions with protected members**

```

member __.AsOk(account) = base.Ok(account)
member __.AsBadRequest(message:string) = base.BadRequest(message) #A

member this.PostWithdrawal(name, request : TransactionRequest) =
 async {
 let customer = { Name = name }
 let! result = bankApi.Withdraw request.Amount customer
 match result with
 | Success account ->
 return this.AsOk(account |> getAccount) :> IHttpActionResult
 | Failure message ->
 return this.AsBadRequest(message) :> _ #B
 } |> Async.StartAsTask

```

#A Manually calling a protected base class member via a top-level member method

#B Calling the delegating member method

Don't worry if you're feeling confused now - I know that this workaround feels completely ridiculous. Hopefully, a future release of F# will automatically do this boilerplate for us, but for now this is one of the edge-cases where OO features of the CLR and F# language features don't mesh together that nicely.

## 39.5 Summary

We're done! You've now written an end-to-end application that's backed by SQL server with a WPF front-end as well as a web-enabled, fully asynchronous API service as well. Not bad! Whilst the tools and techniques you've seen so far whilst building the application are not the only ways to write F# applications, it's worth remember some things that you might not have thought would be possible when you started creating the app: -

- You've written an app that connects to SQL Server and can perform standard CRUD operations in a type-safe manner, quickly and easily.
- You've implemented some business rules and seen how to model a domain with using F# types.
- You hooked up a WPF application running in C# from an F# back-end.
- You also exposed your data over an ASP .NET Web API app written entirely in F#, using asynchronous code, using Swagger to generate documentation, and then consumed it from a script within just a few lines of code.
- The application makes no use of mutable data, and you didn't need to resort to classes or inheritance. There are some impure functions – notably ones that write data to SQL etc. – but the core app adheres to most of the core FP behaviours, such as separation of code and data and higher order functions.

Here are some ideas for further areas to enhance the application: -

- Reincorporate the WPF front-end, but instead of connecting "in proc" to the API, use

the web API that you just created. Use either Swagger or another option for the HTTP client façade.

- Create a dedicated web domain model instead of directly exposing the types from the internal domain over HTTP.
- Write a full web front-end that uses the web api application as a data source.

# Unit 9

## *Unit Testing*

We touched on the concept of testing early on in the book, but haven't looked at it at all since then. This unit will discuss when and where you might want to unit test when working in F#, as well as some different unit testing libraries. As per the other units, I'll show you how to get up and running with popular .NET libraries you might already know, and then we'll "go pro" and use some libraries designed specifically with F# in mind that are seriously cool (or at least, as cool as you can get when it comes to unit testing!).

# 40

## *Unit Testing in F#*

**Let's start this unit with a quick review of basic unit testing tools and how they relate to F#. We'll see: -**

- How we approach unit testing in F#
- How to write unit tests with F# and Visual Studio
- Some F# DSLs for popular unit test libraries

### 40.1 When to Unit Test in F#

Earlier in this book, I touched very briefly on unit testing and how in F# you might tend to not need as much unit testing as you might have done previously. This section will provide an overview of what I consider different “levels” of unit testing, and how and where they’re appropriate in F#. We’ll also discuss different forms of unit testing practices, including test driven development (TDD).

#### 40.1.1 Unit testing complexity

Let's start by stating plain and simple that, yes, there is still a place for unit testing in F#. Whilst its type system allows us to implement some kinds of business rules in code so that illegal states are unrepresentable (and this is a worthy goal), there are many rules that are not easily possible to encode within F#'s type system. Let's partition tests into three groups – basic type system tests, simple rules and complex rules – and see in which languages you might more commonly write these sorts automated unit tests for them.

**Table 40.1 Types of Unit Testing**

Type of Test	Example	Typical Languages
Simple Type	Is the value of the Age property an integer?	Javascript

Complex Type	Is the value of the Postcode field a Postcode?	Javascript, C#
Simple Rule	Only an in-credit customer can withdraw funds.	Javascript, C#
Complex Rule	Complex rules engine with multiple compound rules.	JS, C#, F#

The point here is that the *stronger* the type system, the *fewer* tests you should need. Consider a language such as Javascript – at compile time there's no real type checking, and even at runtime you can assign a number to a property meant to store a string, whilst accidentally assigning a value to a misspelled property (Javascript will merrily carry on in such a situation - hence why languages such as Typescript are becoming popular) – which explains why unit testing is so important in such a language. In effect, you're writing a custom compiler for each of your programs! Languages such as C# eliminate the need for such rudimentary tests, yet even in C# anything more than the most simplistic rules can often lead to the need for unit tests in order to maintain confidence that your application is really doing what it's meant to do. Lastly, we have F#. There are many cases where I would suggest that unit testing doesn't make sense, but you might still want unit tests for complex rules or situations where the type system doesn't protect you. Here are some examples: -

- **Complex business rules, particularly with conditionals** – For more complex rules, or nested rules that combine together to perform some overall feature, you'll still probably want some form of unit testing.
- **Complex parsing** – Parsing code can be tricky, and you might want some form of unit testing to ensure regressions don't occur.
- **A list that must have a certain number of elements in it** - some programming languages (such as Idris) *do* allow you to encode this within the type system . In other words, you can specify that a function takes in an argument that is a list of 5 elements, *at compile time!* These languages are known as *dependently typed* languages; F# is not such a language.

Conversely, here are several cases in which you can often avoid the need for unit testing because the compiler gives us a greater degree of confidence that the code is doing the correct thing: -

- **Expressions** – Probably the most important thing in F# to help us is that we tend to write code as expressions, with immutable values. This alone helps prevent many types of bugs that we'd otherwise need to resort to unit testing for: functions that simply take in a value and return another one are much simpler to reason about and test than those that require complex setup, with state based on previous method calls etc. etc.
- **Exhaustive Pattern Matching** – The F# compiler will tell us that we've missed out cases for conditional logic. This is particularly useful and powerful when pattern matching over tupled values because you can perform truth-table style logic and be confident that you've dealt with every case.
- **Single Case Discriminated Unions** – These provide us with confidence that we've

not accidentally mixed up fields of the same type e.g. Customer Name and Address Line 1 by providing a “type of type” e.g. Name of string, Address of string etc. which prevents this sort of error.

- **Option types** – Not having null in the type system for F# values means that we generally do not need to worry about nulls when working within an F# domain. Instead, we have to deal with the possibility of “absence-of-value” when it’s a real possibility.

### 40.1.2 Test Driven Development or Regression Testing?

We’ve discussed some high-level situations where you might write unit tests, but we’ve not said *when* to write them i.e. before writing production code – test-driven development – or after the fact? I personally can talk from some experience of writing production systems in F# and say that, as someone that was a complete TDD zealot in C#, I don’t perform TDD any more. Not because I can’t be bothered any more, or because it’s not possible in F# - I’ve tried it. It’s simply that the language, when combined with the REPL, means that I don’t really feel the need for it. My productivity feels much higher in F# without TDD - and this includes bug fixing – than in C# with TDD.

Instead, I *do* write unit tests for low-level code that’s either fiddly and complex, or at a reasonably high level – perhaps something that can be matched to a part of a specification – but generally only after I’ve experimented with the code in the REPL, written the basic functionality and made sure it works nicely within the rest of the codebase.

---

#### The F# equivalent of TDD

Mark Seeman (as far as I’m aware) coined the phrase *Type Driven Development*, which has become known as a kind of F# version of *Test Driven Development*. This refers to the idea that you use your types to encode business rules and make illegal states unrepresentable – thus driving development and rules through the type system, rather than through unit tests.

---

Of course, your mileage may vary, and I don’t want to sound dogmatic here. I recommend that you at least *try* by starting writing F# *without* unit tests, and see how you get on. Alternatively, write the tests first - you’ll most likely find that you didn’t really need them (particularly with the REPL). Follow the sorts of rules and practices that you’ve covered in this book, and you’ll probably find that for many cases where you might have resorted to unit testing or even TDD, you won’t need any longer. The first time you perform a compound pattern match and the compiler tells *you* about a case that you hadn’t thought of yourself – that’s when it hits you that, yes, a compiler *can* replace many unit tests.

#### Quick Check

1. What is the relationship between a type system and unit tests?
2. Name two features of the F# language that reduce the need for unit testing.

## 40.2 Basic Unit Testing in F#

Okay, that's enough theory – let's get on with the practical stuff, and write some unit tests. I'm going to use the popular Xunit test framework here, but you can also happily use NUnit or MSTest as well – they all essentially work in the same way.

### NUnit or XUnit?

Both NUnit and XUnit test frameworks are very popular, and both work seamlessly with F#, so there's no reason to move from one to another just for F#. However, there's also a new F#-specific unit testing library that you might want to try out, called *Expecto*. It's very different – rather than a test *framework* with attributes, it's a flexible runner that can make tests out of any *function*. It's beyond the scope of this lesson to show it, but you should definitely check it out.

### 40.2.1 Writing our first unit tests

#### Now You Try

Let's write a set of unit tests for some arbitrary (simple) code in F# using Xunit.

1. Create a new solution in Visual Studio and create a single F# class library. Normally you'd probably create a separate test project, but it's not needed for this example.
2. Add the **XUnit** and **XUnit.Runner.VisualStudio** nuget packages to the project.
3. Create a new file, `BusinessLogic.fs` which will contain the logic that we'll test out: -

#### Listing 40.1 – Business Logic that can be tested

```
module BusinessLogic

type Employee = { Name : string; Age : int } #A
type Department = { Name : string; Team : Employee list }

let isLargeDepartment department = department.Team.Length > 10 #B
let isLessThanTwenty person = person.Age < 20
let isLargeAndYoungTeam department =
 department |> isLargeDepartment
 && department.Team |> List.forall isLessThanTwenty

#A A simple domain
#B Some simple functions on our domain
```

4. Let's now write our first team. Start by creating a new file, `BusinessLogicTests.fs` (remember to ensure it lives *underneath* `BusinessLogic.fs` in Solution Explorer).
5. Enter the following code in the new file: -

#### Listing 40.2 – XUnit Tests in F#

```
module BusinessLogicTests

open BusinessLogic
```

```
open Xunit

[<Fact>] #A
let isLargeAndYoungTeam_TeamIsLargeAndYoung_ReturnsTrue() =
 let department =
 { Name = "Super Team"
 Team = [for i in 1 .. 15 -> { Name = sprintf "Person %d" i; Age = 19 }] }
 Assert.True(department |> isLargeAndYoungTeam)
```

#A A Standard Xunit test using the [<Fact>] attribute and Assert class

6. Rebuild the project. You should see the test show up in Test Explorer in Visual Studio; run it and the test will go green.

Since a module in F# compiles down to a static class in .NET, and let-bound functions in F# compile down to static methods, everything *just works*. You can use all the extra features in XUnit and NUnit as well without a problem e.g. theories, parameterized tests – they all work.

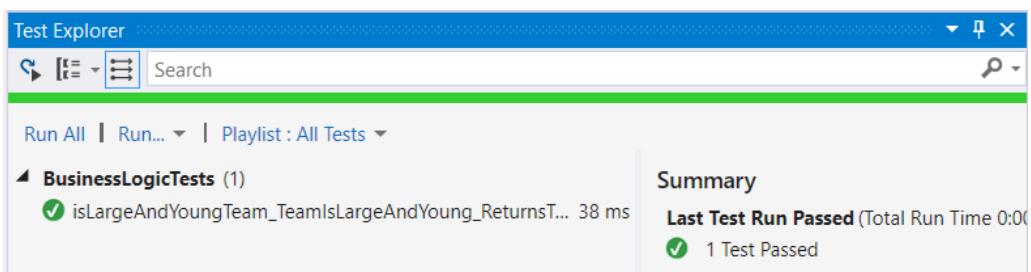


Figure 40.1 – F# tests show up in the Visual Studio Test Explorer as you would expect.

### Removing Class Names from Tests

In Figure 40.1, you'll notice that the test name is *not* prefixed with the class name. To achieve this, you need to add an app setting key "xunit.methodDisplay" to the app.config file of the test assembly, and set its value to "method".

#### 40.2.2 Naming Tests in F#

There are entire blogs (and probably books) on how to name unit tests. This can be quite an emotive subject in many organisations – not to mention difficult to keep consistent. I've seen many different naming "standards", from conventions such as Given\_When\_Then (which is a popular one for people following Behaviour Driven Development) to ones such as Method\_Scenario\_Expected (recommended by Roy Oshrove). There's nothing to stop you following those standards (as I've done in the listing above) but thanks to F#'s backtick methods, you can eliminate this debate completely and simply name the method based on exactly what it's testing: -

```
let ``Large, young teams are correctly identified``() =
```

Believe it or not, not only does this work, but it works beautifully. Try it – rename the test by starting and stopping the name with double back-ticks, recompile and then view Test Explorer again – much nicer! But don't stop there – you can go one step further by renaming the test module name as well, so now the test explorer looks as follows: -

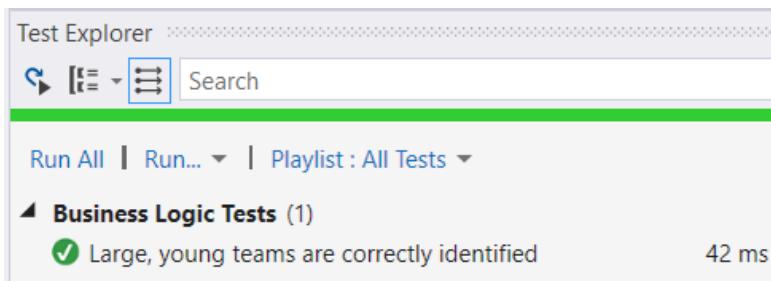


Figure 40.2 – Using F# backtick methods can aid readability of unit tests.

Much nicer, isn't it? Aside from this, unit testing in F# acts pretty much exactly as you would expect in C#.

### What about BDD?

I touched on BDD just before. You can certainly use frameworks such as SpecFlow to write BDD tests, and benefit from the extra readability that backtick members give you, no problem. There's another option, though – *TickSpec*. This is a lightweight F# library that works with Cucumber format tests, but is extremely lightweight and automatically binds up tests to features based on naming convention etc. – all very cool and definitely worth looking at.

### Quick Check

3. What are backtick methods?
4. How is XUnit able to work natively with F# modules and let-bound functions?

## 40.3 Testing DSLs in F#

You can easily create your own domain-specific language (DSL) on top of test libraries so that your tests take advantage of F#'s language features to make tests quicker and easier to read and write. For example, we can use the pipeline and even replace the `Assert.True` static method from Xunit with a helper function that can improve our test readability: -

### Listing 40.3 – Creating a simple F# DSL wrapper over XUnit

```
let isTrue (b:bool) = Assert.True b #B
[<Fact>]
let ``Large, young teams are correctly identified``() =
```

```
// existing code elided...

department |> isLargeAndYoungTeam |> Assert.True #A
department |> isLargeAndYoungTeam |> isTrue #C

#A Using the pipeline with the native Xunit assertion library
#B Creating a simple wrapper around Assert.True
#C Using the wrapper function instead of Assert.True
```

Note that we can immediately use the pipeline with Xunit's existing Assertion library, but by making a simple wrapper function we can make our unit test even more succinct and readable.

To make our lives easier, F# also has a number of ready-made DSL wrapper libraries around the various test frameworks that take advantage of F#'s lightweight syntax to go even further. Let's take a look at a couple of them: -

### 40.3.1 FSUnit

FSUnit is a NuGet package that takes the above approach for a DSL so that you can easily make fluent pipelines of conditions as tests. There are wrappers for both NUnit and XUnit (via the FSUnit.XUnit package). Here's how you might write a couple of tests using FSUnit: -

#### **Listing 40.4 – Using FSUnit to create human-readable tests**

```
open FsUnit.Xunit
[<Fact>]
let ``FSUnit makes nice DSLs!``() =
 department
 |> isLargeAndYoungTeam
 |> should equal true #A

 department.Team.Length
 |> should be (greaterThan 10) #B
```

#A FSUnit's custom language functions for equality checking  
#B Custom checks for "greater than".

FSUnit has a rich language, including functions for string comparisons e.g. "isaac" |> should startWith "isa" and collection tests e.g. [ 1 .. 5 ] |> should contain 3. If you like this style of unit testing, FSUnit is a great place to start.

---

#### **Binding Redirects with FSUnit**

Note that FSUnit was compiled against F#3.0, yet Visual Studio 2015 will by default set F# projects to build against F#4.0. So, you'll need to add a binding redirect to the app.config file of the project (if NuGet doesn't create it). Place this in the <runtime> node:-

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
 <dependentAssembly>
```

```

<assemblyIdentity name="FSharp.Core" publicKeyToken="b03f5f7f11d50a3a"
culture="neutral"/>
<bindingRedirect oldVersion="0.0.0.0-65535.65535.65535.65535"
newVersion="4.4.0.0"/>
</dependentAssembly>
</assemblyBinding>
```

---

### 40.3.2 Unquote

Unquote is a test framework wrapper with a difference. It, too, works with both XUnit and NUnit, but unlike FSUnit, it simply provides a way to easily assert whether the result of a comparison is *true or false*. In other words, it lets you test whether two values are equal to each other. At its most basic, Unquote gives a simple custom operator that can compare two values. Here's how it looks: -

#### **Listing 40.5 – Using Unquote's custom comparison operator**

```

open Swensen.Unquote
[<Fact>]
let ``Unquote has a simple custom operator for equality``() =
 department |> isLargeAndYoungTeam =! true #A
```

#A The custom `=!` operator fails if the values on both sides do not equal

Of course, you can use this operator for more than comparing booleans – for example, comparing whether two lists are equal, or two records are the same (remember that F# types implement equality already!).

But Unquote goes one step further than this. Unquote, as its name suggests, takes advantage of F#'s *quotations* language feature. This allows Unquote to explain *why* two values don't equal one another.

---

### F# Quotations

F# quotations are beyond the scope of this book. Suffice it to say that they are roughly equivalent to C#'s expression trees – in other words, by enclosing a code block within an F# quotation, you can get back a typed (or untyped) abstract syntax tree to perform analysis on the code itself i.e. treating *code* as *data*. Clear as mud? Don't worry – you don't need to understand them to take advantage of Unquote.

Here's an example of how you'd use Unquote's quotations support for a simple test: -

#### **Listing 40.6 – Evaluating a quotation with Unquote**

```

[<Fact>]
let ``Unquote can parse quotations for excellent diagnostics``() =
 let emptyTeam = { Name = "Super Team"; Team = [] }
```

```
test <@ emptyTeam.Name.StartsWith "D" @> #A
#A Wrapping a condition within a quotation block.
```

A quotation is easy to create – you simply wrap around the condition to test with `<@ ... @>` (Visual F# Power Tools will also identify the quotation and colour it differently for you): -

---

```
[<Fact>]
let ``Unquote can parse quotations for excellent diagnostics``() =
 let emptyTeam = { Name = "Super Team"; Team = [] }
 test <@ emptyTeam.Name.StartsWith "D" @>
```

Figure 40.3 – VFPT highlights code quotation blocks for you

If the result of the expression inside the block returns true, the test passes. What's more interesting is if it fails (as per the example above). Let's look at a slightly simplified version of the error output from Visual Studio's Test Runner: -

```
emptyTeam.Name.StartsWith("D")
{ Name = "Super Team"; Team = [] }.Name.StartsWith("D")
"Super Team".StartsWith("D")
false
```

The first three lines represent a *step-by-step guide* of how the test failed: -

1. Line 1 represents the original test code.
2. Line 2 examples the test code by replacing `emptyTeam` with the actual record that it represents.
3. Line 3 simplifies this to just the value of the `Name` property that was being compared against.
4. We're finally left with just a simple comparison which returns `false`, so the test fails. But we now can see exactly how unquote reached the value "`false`".

This is a relatively simple example, but it's actually fantastically powerful – imagine your test code calls a function of production code which itself calls two other functions that return some data etc. etc. etc. – it quickly becomes very difficult to understand *why* a test failed. Unquote sheds light on this by allowing you to understand the code without needing to resort to a debugger.

### Quick Check

5. What is FSUnit?
6. What is a code quotation?

## 40.4 Summary

That's the end of basic unit testing in F#! We covered a few points here: -

- We discussed when and where unit testing might and might not be appropriate in F#.
- We looked at basic unit testing integration in F# with VS2015 and XUnit.
- We saw a couple of custom F# libraries that can make unit testing even better.

### ***Try This***

Try porting some unit tests in your own code to an F# library. Experiment with your own custom DSL functions, FSUnit and Unquote; try performing a set of nested function calls that return a value, and test calling this through Unquote.

### ***Quick Check Answers***

1. Generally, the stronger the type system, the fewer tests that are needed.
2. Option Types, Expressions, Exhaustive Pattern Matching, Discriminated Unions.
3. Methods surrounded with double backticks can include spaces and other characters.
4. Modules and let-bound functions compile down to static classes and methods, which XUnit supports natively.
5. A DSL wrapper around XUnit and NUnit to allow fluent, human-readable tests to be written.
6. A block of code in which code is treated as data which can be programmed against.

# 41

## *Property Based Testing in F#*

**In this lesson, we'll look at a different type of automated testing that one can do in F#, called Property Based Testing (PBT). We'll see: -**

- What PBT is
- Why you might want to use it
- The FsCheck library for .NET
- How to integrate FsCheck with popular test runners

### **PRIMING EXERCISE – LIMITATIONS OF UNIT TESTS**

There are times when, even if we've achieved "full test coverage" – that is, our tests are covering every branch of code in the system – we still might not be satisfied with our tests. Here are a few examples: –

One common problem is when your code misses out "edge cases" that you've not considered – this is common when working with strings or other "unbounded" inputs. How many times have you written code that "expects" a string to be a certain minimum length but, somehow, an empty string creeps into your code? Was that something that you should have realistically expected? Or an empty array, and you try to index into it?

Another example is where your unit tests appear to be fragile because they only test arbitrary cases. Imagine a situation where you have a method, `half()`, being tested which takes in a number and returns half of it. To test this out, you write a single test case that proves that when calling `half()` with 10, you get back 5. Why did we pick 10 as the "test data"? Is it easy to understand what the test is really proving? Have you tested all valid cases?

The last example I'll give are when a system under test is too complex to cover (or even identify) all the possible permutations with specific test cases. This is particularly common – but not unique – to integration tests. Normally we unit test lower level components very well, but tend not to have great integration tests – they're too expensive to write to cover all combinations. But this often misses out bugs where there are unintended interactions between

two components when you finally bring them together. A good real-world example of this can be seen with the Mars Orbiter in 1999, which failed because one team wrote a module using English units of measurement, whilst unknowingly, another team used the metric system! Both components worked well in isolation, but when plugged together, the system failed because of the differences in units. End result: a costly failed space craft (let's leave aside the fact that F# has a feature known as Units of Measure which would have prevented this...).

There are all sorts of occasions where conventional unit tests feel "fake" or simply unsatisfactory – it's hard to put your finger on why, but something doesn't "feel" right! This is often a sign that another form of testing might be worthwhile.

## 41.1 What is Property-Based testing?

Regular unit tests work on a simple basic process (I'm ignoring the whole areas of mocking frameworks etc. here!): -

1. We manually create some test data.
2. We push that test data through some production code.
3. We confirm that the outputs of that code are as expected.

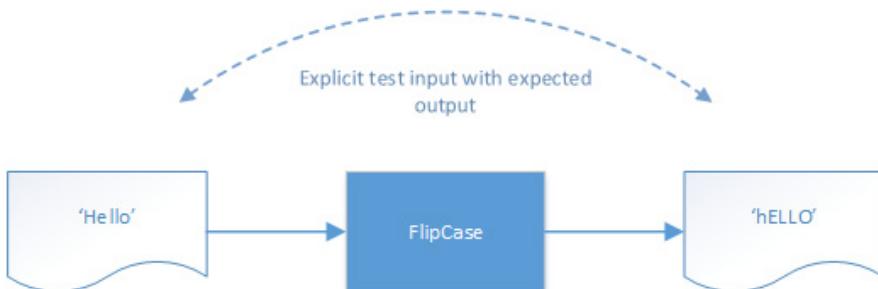


Figure 41.1 – A function with an explicit test input and expected output

Figure 41.1 shows a sample function, `FlipCase`, which takes in a string and flips the case of all the characters. We might normally test this with a single word as a unit test e.g. "Hello" becomes "hELLO". The idea behind property-based testing is that instead of testing code with arbitrary data that we create ourselves, we instead allow the *machine* to create test data *for us*, based on some guidelines that we provide. Then, we test behaviours, or *properties* of the system that should hold true for *any* input values. Write enough properties, and you prove the functionality of the function as a whole.

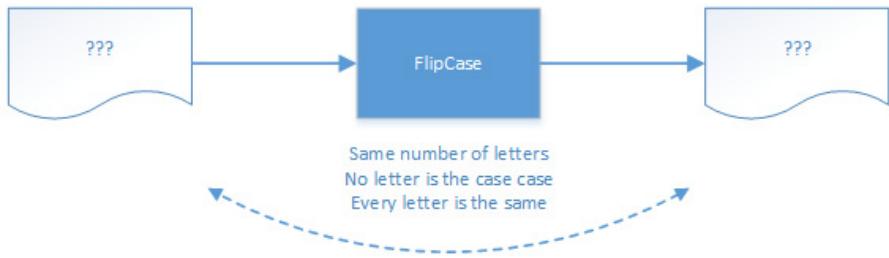


Figure 41.2 – A function with sample properties that should hold true for all possible input values

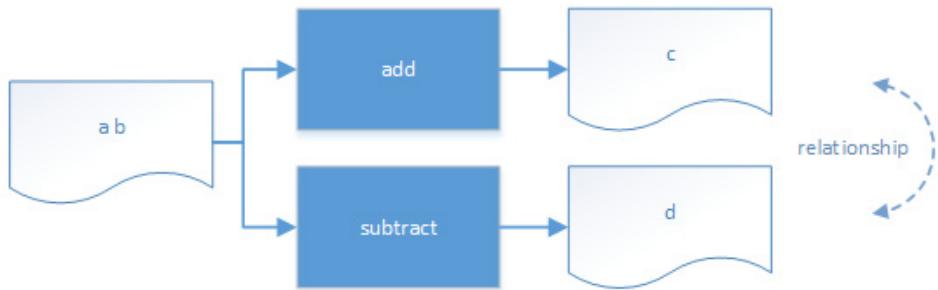
Property-based testing requires a different approach to thinking about tests – unlike conventional unit tests, the system generates test values *for us*. Therefore, we can't hard-code the “expected” result of a test. This is where the notion of *properties* comes in. A property is some kind of *relationship* that you can test on the output of your production code, without knowing the actual *value* of it. In Figure 41.2, we've specified three properties that should all hold true for our FlipCase function. We don't need to know the actual inputs or outputs – we just need to prove that those behaviours hold true against a large enough sample dataset to prove that the code works.

Property-based testing can help with many kinds of tests, and can help identify specific corner cases (from which you might well want to write specific unit tests for).

### 41.1.1 How to identify properties

Identifying properties is one of the hardest parts of property based testing, and it's not something that (in my experience) you can easily illustrate in a few pages – you could probably write a whole book on the subject! So whilst I'm going to give you some examples here of how you *might* identify properties in your production code, think of this as a “flavour” into what property testing is about, rather than a detailed look into every aspect of it: -

- **Identify specific properties about the behavior of the inputs and outputs.** This depends very much on the code being implemented and functionality you're trying to implement. For example, in Figure 41.2, I identified three specific properties of the FlipCase function.
- **Identify a relationship between two functions.** You can pass the *same input data* into two *different* functions and see if there's a *relationship* between the outputs of those two functions. For example, there's a relationship between the two functions “add two numbers together” and “subtract two numbers together” that you could define as a property - in Figure 41.3, it's that  $(c - d = b * 2)$ . A property-based test would generate many “pairs of numbers” that you would pipe into that equation and “prove” if it's true or not through brute force.



**Figure 41.3 – Using a secondary function is often a useful way to identify a property test**

- **Compare another implementation of the same function.** If you have an alternate function that does the same thing and is known to work correctly, you can compare the results to ensure that they are always the same. This is useful when refactoring a function e.g. to optimize performance – you can help ensure that the behaviours are the same for many arbitrary test cases. This is actually just a specialized form of the previous item – in this case, the relationship is that the values should always be equal!

The main point is that *all* of these sorts of tests can be achieved in a generalized way i.e. without having to *know* a specific input or output value.

Property-Based testing in the real world

A few examples of how property-based testing has been used to identify issues and prove the validity of a system include the Riak database system (which had several serious bugs that could result in loss of data, until they ran it through a property-based test system) and Dropbox. Another example is the Paket nuget package manager, which used FsCheck to create different types of package dependency graphs to ensure that the package resolution algorithm always picked an optimum set of versions for the packages specified.

## **Quick Check**

1. Name the main difference between conventional unit tests and property based tests.
  2. Name three types of ways to identify properties within a system under test.

## 41.2 Introducing FSCheck

That's enough theory for now – let's move onto the practical side. FsCheck is a property testing library that is effectively a port of Haskell's QuickCheck library. FsCheck allows us to provide it with a *parameterised* function that performs some business logic, and then test the result of that logic. FSCheck then calls the test function many times, each one with slightly different input values. In some ways, this is not so different from NUnit or XUnit's `TestCase` or `Theories` features, except here you don't specify the data to test against - FsCheck does.

### 41.2.1 Running Tests with FsCheck

#### Now you Try

Before we jump into the thick of things with full-blown property based tests, let's just start with a simple test that uses FSCheck to get a feel for the library itself.

1. Create a new F# class library project.
2. Download the **FsCheck.Xunit** nuget package (this will also download the core FsCheck package as well as XUnit).
3. Create a standard F# module and enter the following code. It creates a simple function that manually adds the numbers in a list together, followed by a test that compares the values with a "known good" function, `List.sum`, that we use as a basis: -

#### **Listing 41.1 – Our first FsCheck test**

```
open FsCheck.Xunit

let sumsNumbers numbers = #A
 numbers |> List.fold (+) 0

[<Property(Verbose = true)>] #B
let ``Correctly adds numbers`` numbers = #C
 let actual = sumsNumbers numbers
 actual = List.sum numbers #D

#A Code under test.
#B Writing a parameterized unit test
#C A parameterized unit test
#D Comparing expected and actual
```

4. Now run the test; you'll see that this actually runs, and also turns green.

Let's review. The test defined in Listing 41.1 is a *parameterised* test – it takes in a list of numbers which we run against our code, and compare against the `List.sum` function. But what list of numbers?

5. To understand this, open the test explorer window and navigate to the appropriate test before clicking the Output link. You'll see something like this: -

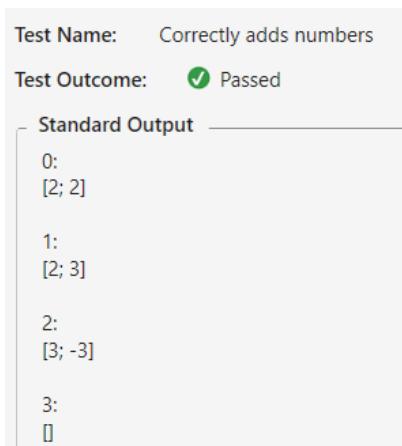


Figure 41.4 – The output of an FsCheck test

If you scroll down the pane, you'll see a whole host of entries. What this means is that FsCheck has seen the type of data that is needed by the test – a list of integers – and *generated* a set of samples for it – in fact, it creates and runs *100 test cases* by default. The data isn't completely random either – it will intentionally start with simple cases and then expand to more complex ones. So, in this example, you can see that it tries a simple list with 2 items, empty lists, different values etc. etc. Even better – FsCheck can generate entire object graphs – full F# records (or lists of records) each with their properties populated.

6. Place a breakpoint on the first line of the unit test.
7. Re-run the test but ensure that you choose Debug Selected Test in the test explorer when you right-click the test.
8. Observe that the breakpoint is repeatedly hit, once for each test run, with different data.

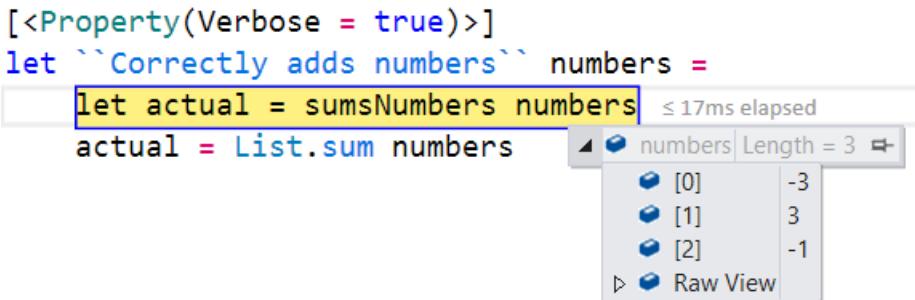


Figure 41.5 – Setting a breakpoint within a unit test that is called by FsCheck

## FsCheck and Microsoft Pex

Some years ago, Microsoft Research came up with a testing framework named Pex. Pex would analyse your code and generate test cases based on that. FsCheck doesn't do anything like that – it simply generates random test data across a known distribution that you can control with the intent of proving or disproving your test cases.

### Failing tests and Shrinking

#### Now you Try

Now let's make our production code fail and see what happens.

1. Change the implementation of sumsNumbers so that if the list of numbers contains 5, always return -1 (otherwise continue with normal logic).
2. Re-run the tests. You'll see that the test fails: -

```
FsCheck.Xunit.PropertyFailedException :
Falsifiable, after 5 tests (3 shrinks) (StdGen (382638564,296259880)):
Original:
[5; 5; -5; -4]
Shrunk:
[5]
```

Figure 41.6 – A failing test with a shrunk test case from FsCheck.

The key bits to notice here are two-fold: In Figure 41.6, you can see two items: -

- **Original** – the initial data that failed the test, randomly generated by FsCheck. Notice that the list contains four values in it.
- **Shrunk** – a *simplified* dataset that fails the test based on the original data. Notice now that the dataset only contains a single element – 5! FsCheck has realized that the other values have no impact on the test, and so strips them out leaving us with the *simplest possible failure case*.

"Shrinking" is the process by which FsCheck "reduces", or simplifies a failing test data set to be the *simplest* possible failure case that it can be. This happens automatically, and massively helps us to identify how and why a test failed.

#### Quick Check

3. What attribute do you place on a test to allow FsCheck to provide test data?
4. What is Shrinking?

## 41.3 Controlling Data Generation

There are times when you might want to control up-front the data that is generated by FsCheck. For instance, let's take our fictional `FlipCase` function and write one of the property tests for it: -

### **Listing 41.2 A function to flip case of all letters with a single property test**

```
Open System
let flipCase (text:string) =
 text.ToCharArray()
 |> Array.map(fun c ->
 if Char.IsUpper c then Char.ToLower c
 else Char.ToUpper c)
 |> String

[<Property>]
let ``Always has same number of letters`` (input:string) =
 let output = input |> flipCase
 input.Length = output.Length
```

The one problem with this is that this function is only designed to work with *letters* – no numbers or special characters are allowed, and no null values either. If you run this test now, there's a good chance that it will fail – because FsCheck will pick random strings, *including* those with non-letters. Now, you could make the `flipCase` function “throw out” such values, but in our case, we want the test to tell FsCheck to only supply valid data to the test; we need a way to tell FsCheck to generate string with *only letters*.

### 41.3.1 Guard Clauses

Luckily, FsCheck offers a simple way to help us using a “guard” clause in our test function: -

### **Listing 41.3 Providing a guard clause for FsCheck**

```
[<Property>]
let ``Always has same number of letters`` (input:string) =
 input <> null ==> lazy #A
 let output = input |> flipCase
 input.Length = output.Length
```

#A Adding a guard clause to an FsCheck property.

Here, we've added a guard clause that says to “prematurely exit” any test where `input` is null. (FsCheck comes with the custom `==>` operator, which says “only run the code on the right if the guard clause on the left passes”). Also, notice the `lazy` keyword – this is a language alias for `System.Lazy` and ensures that FsCheck only actually runs the test code when the guard check has passed.

### 41.3.2 Generators and Arbitrary Values

The problem with guard clauses is if you need a specific clause. For example, if you try to implement either of the other two properties, you'll need to enhance the guard clause so that it not only ignores null strings, but also empty strings or those with non-letters in them. In such a case, FsCheck will give up and complain that it couldn't generate enough valid data with a message such as `PropertyFailedException: Arguments exhausted after 3 tests.`. In other words, FsCheck couldn't randomly generate enough data to "pass" the guard clause for 100 tests.

When guard clauses aren't sufficient for our needs, you'll need to create a custom *generator*. A generator allows you to specify a certain type of data, such as chars or strings, which then map into *Arbitrary* values that FsCheck can run in its tests. FsCheck comes with many built-in arbitrary values in the `FsCheck.Arb.Default` module, such as the usual lists, sets, numbers etc. and even things like non-empty strings, IP Addresses and so on. Let's create our own one that forces all characters that are generated to only be letters. In conjunction with the built-in `NonEmptyString`, we can achieve exactly what we need: -

#### **Listing 41.4 Creating a letters-only generator for FSCheck**

```
Open FsCheck
type LettersOnlyGen() =
 static member Letters() =
 Arb.Default.Char() |> Arb.filter Char.IsLetter #A

[<Property(Arbitrary = [| typeof<LettersOnlyGen> |])>] #B
let ``Always has same number of letters`` (NonEmptyString input) = #C
 let output = input |> flipCase
 input.Length = output.Length

#A Creating a class that contains arbitrary generators
#B Creating a generator that creates a stream of letters
#C Attaching the generator to the property test
```

The trickiest bit to understand here is the `LettersOnlyGen` type. This class has a single static method on it, `Letters`, that returns a type `Arbitrary<Char>` - the type that controls what set of data FsCheck can use to "pick" from randomly. You can create multiple generator methods with any name – there's no conventions, but there must be only one method per type that you wish to generate e.g. strings, integers, customers etc. In this case, we're creating the set of all characters filtered by whether the character is a letter. This is essentially the same as doing a `Seq.filter` or similar, except instead of operating on a real list of items, we're supply the logic to FsCheck to tell it how to generate data.

Next, we apply that generator to the test by setting the `Arbitrary` value of the `Property` attribute. This ensures that all characters generated for this test are letters. And we can replace the null check guard by using the `NonEmptyString` discriminated union that was previously mentioned, which FsCheck guarantees will never be null. Now if you rerun the test

(and turn on `Verbose` mode by passing `Verbose = true` to the `Property` attribute), you can observe the test cases will all be valid test data e.g.

```
NonEmptyString "cAv"
NonEmptyString "S"
NonEmptyString "Yo"
NonEmptyString "UkUr"
NonEmptyString "NBwhlF"
NonEmptyString "el"
NonEmptyString "UI"
NonEmptyString "CxyP"
NonEmptyString "xeuZepy1"
NonEmptyString "bUIx1"
NonEmptyString "Pf"
```

Creating your own generators isn't always trivial to do, but the time and effort spent doing it is definitely worthwhile. I've only used simple types here in this sample, but imagine creating a generator that provides customers of a certain type for you to test, or user requests meeting a certain specification. This can be absolutely invaluable in rapidly creating test data for multiple tests, quickly and easily.

### **Quick Check**

5. What are guard clauses?
6. When should you not use a guard clause and prefer a full Arbitrary generator?

## **41.4 Summary**

Property Based Testing is a very different way to approach automated testing. However, it's also a very useful tool to have in your arsenal, particularly for certain types of problems that are very difficult to "white box" test (i.e. test the internals), and where it's easier to simply "black box" test (i.e. test the external behaviours). This lesson has only scratched the surface of PBTs, and you should seriously consider checking it out in more detail in your own time.

### **Try This**

Try implementing tests the other two properties for the `FlipCase` function. Then, write some example functions and associated PBTs for the data model in the previous lesson, before writing a set of property based tests against a method in a class in the BCL e.g. `System.Collections.Generic.Queue.Enqueue`.

### **Quick Check Answers**

1. We supply explicit inputs and outputs to unit tests. PBTs are supplied random generated data many times.
2. Identify arbitrary properties, compare a relationship between two functions, compare to a "known good" function.
3. The `[<Property>]` attribute.

4. The process of simplifying a failure case to it's most basic form.
5. Simple predicates that tell FsCheck that data is invalid for a given PBT.
6. Guard Clauses are only useful for simple filters. If the filters are too constraining, FsCheck will not be able to generate sufficient test data that meets the requirements. In these situations, Arbitrary generators should be used.

# 42

## Web Testing

**This short lesson will round off our trip around testing in F#. We'll see: -**

- What web testing is, and where it fits in within the testing landscape
- What Selenium is
- What Canopy is
- How we can use Canopy for web automation

### **PRIMING EXERCISE – INTEGRATION TESTING YOUR WEB APPLICATIONS**

Oftentimes, we find that unit testing – even property based tests – aren't enough to test an entire system from end-to-end. Perhaps you've written code that has excellent test coverage against individual components such a class that calculates the interest on a loan payment or perhaps across two or three layers – but then tested the “real” application and found that the interest calculate that was saved to the database was incorrect. We need something that can ideally cover the *entire spectrum* of tiers within our application, yet we would like this to run in a repeatable way.

In my experience, there are two popular ways of doing this. One is through an acceptance testing framework like *Fitness* – a system that allows business users to specify tests through *tables* in what is essentially a Wiki website, which then calls production code behind the scenes. It's very powerful, but also requires a reasonable learning curve. That's fair enough – after all, F# also has a learning curve associated with it – but it's more than just the investment in learning it. It's a full application, so you need to manage it within your application tier somehow, as well as consider how it works within the context of a continuous integration environment etc. And lastly, the .NET SDK for it is somewhat “unusual” – certainly it's not what I'd call idiomatic.

An alternative to this when working with a website is to write web-based tests. These are often step-by-step tests that people can follow, and then record the results in some bug-tracking system during testing. This process is so popular that some versions of Visual Studio

(and it's sibling, Team Foundation Services) even has a "test recorder" that allows you to record a video of your test along with the results of the test, plus capture information such as call stack etc. to more easily enable error reproductions. That's nice, but it's ultimately a manual system – the more tests you have, the longer (and more expensive) it becomes to run through the test suite.

## 42.1 Web Automation with Canopy

The alternative that we'll be looking at is how to run web-based tests in an automated fashion, by writing simple, *declarative* programs in F# that can state how a web application should behave, by filling in fields on a web page, clicking buttons etc. – essentially, interacting with a web site to simulate a real person – and then testing the results e.g. what is the value of a field? What URL are we now on? etc. etc. But before looking at the testing side of things, let's start with the basics – controlling a browser through code.

---

### Selenium

Selenium is a freely distributable WebDriver plug-in to your web browser, plus an SDK and IDE, which allows you to perform what is known as *web automation*. In other words, you can programmatically automate the control of your web browser. It can be run as a standalone tool and generate scripts via an IDE, but as we'll see, there's a much nicer way of doing this in F#.

---

### 42.1.1 What is Canopy?

Canopy is an open source, free to use library which wraps around the Selenium plug-in. It provides an extremely simple, easy to use DSL for writing simple scripts that can control your web browser. As it's just a wrapper on top of Selenium, any browser that Selenium supports, is also supported by Canopy (this includes IE, Chrome and Firefox etc.). We can use Canopy to perform all manner of tasks, such as filling in fields, clicking buttons, dragging elements from one to another etc. etc. This can be exceptionally useful for automating processes that must be done through a web-based UI – I've seen individuals use this for anything from uploading information from a spreadsheet into a web portal, to filling in timesheets at the end of every week!

### 42.1.2 Creating our first Canopy script

#### **Now you Try**

Let's take Canopy for a quick spin. For this exercise, we'll use the Google Chrome browser – the majority of this will be identical for other browsers, but in the interest of keeping things simple, I'm going to just focus on Chrome (the Canopy site documents any differences). In this exercise, we'll open up Chrome, navigate to the Manning web site and fill out a form to

buy another copy of *Get Programming with F#* (don't worry, we won't really complete the checkout!) - two copies are better than one, right?

1. The first thing you'll need to do is download the latest version of the Chrome Web Driver from <https://sites.google.com/a/chromium.org/chromedriver/>. This driver allows us to communicate with Chrome programmatically, so copy it to the drivers/ folder in the learnfsharp repository.
2. Create a new F# script. As usual, I recommend to either put it in the src/code-listings folder, or at least one that's parallel to it, so that the package references in the code listings in this lesson work straight away.
3. Enter the following code into your script. This simply references the core Selenium and Canopy assemblies and then sets the path to where the chromium web driver is located (chromeDir is a global mutable variable exposed by Canopy!)

```
#r @"..\..\packages\Selenium.WebDriver\lib\net40\WebDriver.dll"
#r @"..\..\packages\canopy\lib\canopy.dll"
open canopy
chromeDir <- "drivers"
```

4. Let's now start a new instance of chrome under Canopy's control.

```
start chrome
```

You should see a similar output to the following in the FSI window: -

```
Starting ChromeDriver 2.27.440174 (...) on port 3732
Only local connections are allowed.
```

You'll notice that Chrome opens up by default tiled to the right of the desktop – if you tile VS to the left, you can send commands to Chrome and watch them take effect.

5. Navigate to the Manning site: url <https://forums.manning.com/forums/get-programming-with-f-sharp>. Observe how the DSL makes commands extremely easy to read e.g. start chrome, url <url> etc. No brackets or curly braces, and we can execute arbitrary commands from the script.
6. There are two versions of the book that you can buy – we'll buy the combo version.

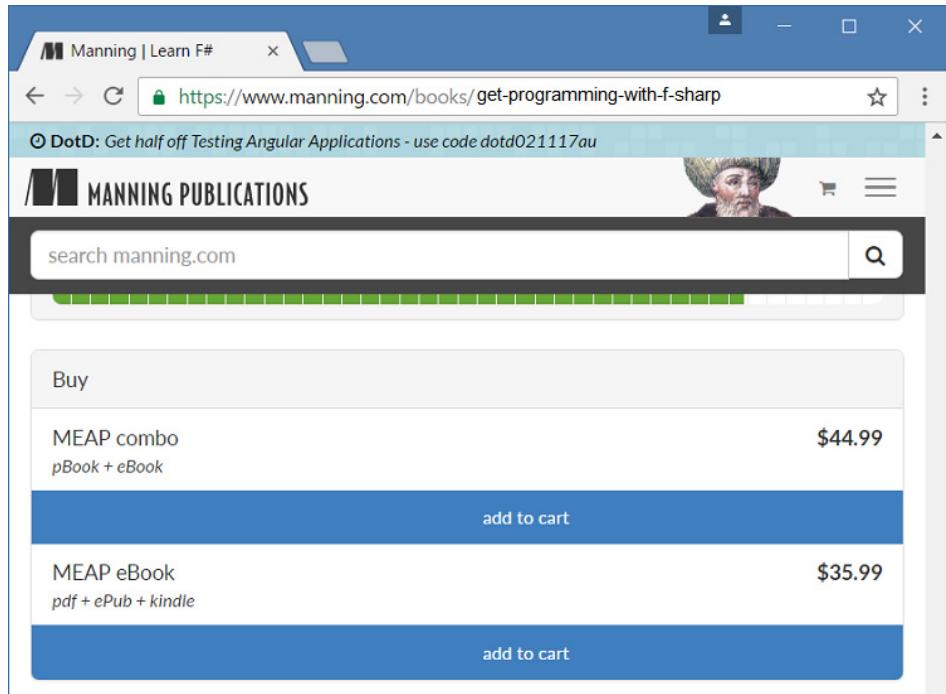


Figure 42.1 – Buying “Get Programming with F#” on the Manning website

Both of these buttons are Submit buttons – so we’ll use Canopy to just get the *first* element that has an id of `submit`, and then click it: -

```
first "#Submit" |> click
```

The identifier for the element can be any valid css selector. You’ll see that we click the button, and receive a notification that the item has been added to the cart. You can also confirm this at the top of the page where the cart icon is.

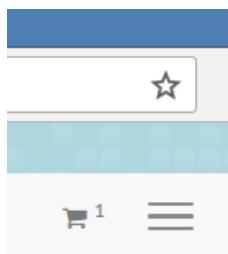


Figure 42.2 – Adding an item to the shopping cart on the Manning website.

## Getting element names for Canopy

You see that above there are a whole set of element ids and classes I've used to identify elements that we can access. I've manually gone through the page in Chrome's developer panel to "pull out" the ids. Another option is to search for elements with specific text using Canopy's `elementWithText` function.

7. Let's now proceed to the checkout. We do this by first going to the cart, and then clicking the checkout button that is on that page: -

```
click ".cart-button"
click ".btn-primary"
```

8. Now, we're asked to either sign in or checkout as a guest. Let's do the second option – let's set the Email address textbox to a random name before clicking the last (i.e. the second) button on the form.

```
"#email" << "Fred.Smith@fakemail.com"
last ".btn-primary" |> click
```

Note the `<<` operator – another feature in the Canopy DSL. This allows you to set the contents of any element on the left-hand side with the contents on the right hand side!

9. We'll now be on the final page which captures details of the guest. Fill in all the form fields as follows: -

```
"#firstName" << "Fred"
"#lastName" << "Smith"
"#company" << "Super F# Developers Ltd."
"#address1" << "23 The Street"
"#address2" << "The Town"
"#city" << "The City"
"#freeformState" << "ABC"
"#zip" << "90210"
"#addressPhone" << "0800 123 456"
```

We'll stop there (you can close the browser using `quit()`). The main point to illustrate is that we can create a repeatable script to easily control a web browser to perform a number of actions. Canopy has a rich set of helper functions that allow us to easily interact with a browser, including reading the contents of an element, double-clicking, switching tabs etc. – a whole host of functions (you can find out more at <http://lefthandedgoat.github.io/canopy/actions.html>)

## Quick Check

1. What must you manually download before using Canopy?
2. Does Canopy only work with Google Chrome?
3. What custom operator do you use in Canopy to assign text to an element?

## 43.2 Web Tests with Canopy

Now that you've seen the basics for how Canopy works, let's briefly look at the second half of Canopy, which is a full test runner designed to create and run automated tests.

### 43.2.1 Hooking into Canopy events

Canopy has a very simple test runner that can launch directly from a script or a console application – there's no need to shoehorn it into a class library that supports the VS test runner if you don't want to. We simply declare events and tests through a set of functions, before starting a test run. Let's first see how we can hook into events for when tests start and finish etc.

#### **Listing 42.1 – Setting up a basic test run suite in Canopy**

```
once (fun _ -> start chrome) #A
before (fun _ -> url "https://www.manning.com/books/get-programming-with-f-sharp") #B
lastly (fun _ -> quit()) #C
```

#A Run once before starting the test run  
#B Run before each test  
#C Run once after the all tests

In other words, before we start our test run, we start chrome. At the start of each test, we'll navigate to the manning website, and after all the tests are complete, we'll close the browser. These functions relate roughly to the [`<SetUp>`] and [`<TearDown>`] etc. attributes in NUnit or XUnit. You can find out more on the different functions and features at <https://lefthandedgoat.github.io/canopy/testing.html>.

### 43.2.2 Creating and running tests

Creating tests with Canopy is just as simple – we call one of two functions to *create* (but not *run!*) a test case. Both are higher order functions. The simplest form creates a case with an auto-generated name of "Test #n" e.g. Test #1, Test #2 etc.

```
test <test function>
```

The test function is simply a function that takes no arguments and returns no arguments – inside, it performs some test code using Canopy's assertion test functions (see below). The alternative form allows you to specify a custom test name:

```
<test name> &&& <test function>
```

Each test case that gets executed adds the test case to a list that Canopy maintains of all test cases. Where you're done, simply calling `run()` will execute all the test cases.

### 42.2.3 Assertions in Canopy

Canopy has a reasonably large set of common test functions that can do the usual comparisons of elements etc. etc., as well as several others. Here I've created three tests within a test context called "Sample Tests": -

#### **Listing 42.2 – Creating a simple test suite in Canopy**

```
context "Sample Tests"

test (fun _ -> "#chapter_id_1" == "LESSON 1 THE VISUAL STUDIO EXPERIENCE") #A
"49 lessons in total" && fun _ -> count ".sect1" 49 #B
"There's a web programming unit" && fun _ -> ".sect0" *= "UNIT 8: WEB PROGRAMMING" #C

run() #D

#A Testing the value of an element
#B Testing the count of elements
#C Testing that a value exists in a collection
#D Actually run the tests
```

Let's take these tests one by one.

- The first test simply checks that the value of a specific element (identified by an id of chapter\_id\_1) is equal to LESSON 1 THE VISUAL STUDIO EXPERIENCE. Note the custom == operator which Canopy uses to perform the equality check.
- The second test is titled "49 lessons in total" and confirms that there are 49 elements that have the sect1 css style.
- Lastly, the final test get all the sect0 elements and checks that at least one of them has the value UNIT 8: WEB PROGRAMMING.

There are more helper functions and operators that you can read up on at the Canopy website here <https://lefthandedgoat.github.io/canopy/assertions.html>.

Once we've compiled our tests, we can simply run them using the run() function. The output should be something like this: -

```
Starting ChromeDriver 2.27.440174 on port 3446

Test: Test #1
Passed
Test: 49 lessons in total
Passed
Test: There's a web programming unit
Passed

0 minutes 10 seconds to execute
3 passed
0 skipped
0 failed
```

## Integration with other libraries

Canopy is an extensible library, so you can choose how you output data – there's a TeamCity plugin which outputs test results in a manner that TeamCity can read, or you can create your own. And remember, there's nothing to stop you using Canopy's assertion libraries etc. but use e.g. NUnit's test runner. Similarly, there's nothing to stop you using FsCheck to generate test data for you that you can then plug into Canopy!

There's even a web page version of the test reporter so that you get an interactive web page showing test results rather than just the console.

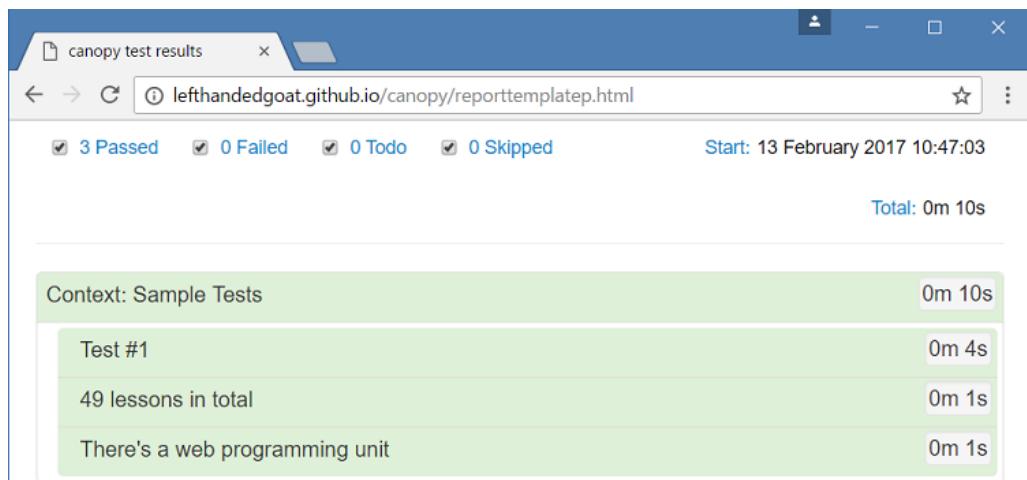


Figure 42.3 – The Canopy UI test runner

You can activate this by adding the following code before you start your test run:

```
open reporters
reporter <- LiveHtmlReporter(BrowserStartTime.Chrome, "drivers") >: IReporter
```

I strongly advise you to try this section out yourself and see it working first-hand – once you realise how easy Canopy is to use, there are endless possibilities for easily creating repeatable tests!

### Quick Check

4. Can you use Canopy with other test runners?
5. What's the difference between `test` and `&&`?
6. What does the `==` operator represent in Canopy?

## 42.3 Summary

That's the last testing lesson. You've touched the edges of Canopy, and seen how you can write automated tests within F#, using a powerful yet simple to use DSL. Although it's not necessarily a "pure functional" library – there's global state behind the scenes with mutable properties etc. - Canopy is an extremely practical library that does a terrific job of making web automation incredibly simple. You saw: -

- How to install Canopy and Selenium.
- How to perform basic web automation with Canopy.
- Writing automated tests in Canopy.

### **Try This**

Create a web automation script to log into Twitter and print out to the console the top five posts in your Twitter feed.

You can also use Canopy to create a bot to play the game "2048" – see <https://github.com/c4fsharp/Dojo-Canopy-2048> for a starting solution.

Finally, write automation code which searches GitHub for all F# repositories, sorting by Most Stars, and prints out the first ten repositories.

### **Quick Check Answers**

1. You must download the appropriate web driver for your browser.
2. No – Canopy works with multiple browsers.
3. The << operator is used to assign text to an element.
4. Yes, you can use Canopy from within e.g. XUnit or NUnit.
5. `test` generates a name for a test case whilst `&&&` allows you to specify a name.
6. The `==` operator performs an equality test equivalent to e.g. `Assert.Equals`.

# 43

## *Capstone 8*

**Phew - this is the very last capstone exercise in this book! We'll be covering unit testing here, and applying some of the techniques and tools that we covered in this unit to the bank accounts system we've been working on throughout the book.**

### 43.1 Defining the problem

In this capstone, we'll add a set of tests to the business logic domain of our application, using a number of different test libraries that we've seen in this unit, from simple unit testing with XUnit, to DSL-based testing with Unquote, before finally writing a couple of property-based tests. We'll look to test a couple of different tiers as well; in this example, we'll not be writing any Canopy-based tests, but we will test out a couple of internal layers within the application – the Web API tier and the internal Bank API: -



Figure 43.1 – The API layers selected for unit tests

#### 43.1.1 Solution Overview

The solution we'll be working with is essentially what we finished up with at the end of the last capstone – a web-enabled version of our Bank API; you'll find it in the lesson-43 folder. The main difference is that I've added a new test project to the solution. Once again, to save you from working with the hassles of NuGet binding redirects, I've done all the hard work for you. But, just so you know what's been done: -

- The project has the following NuGet packages (plus their dependencies): -

- XUnit (and the Visual Studio runner)
- Unquote
- FSCheck (and XUnit integration)
- ASP .NET Web API Core
- The versions of the dependencies have been set to match up with those in the other projects. For example, Newtonsoft.Json is set to 9.0.1.
- There are a set of binding redirects to ensure that everything works nicely.

(Of course, if we used Paket, none of this would be a problem, since packages are automatically kept consistent *across* projects.)

### **100% Test Coverage?**

Don't expect to go through a full set of exhaustive tests in this capstone exercise. As I've mentioned, I'm not a fan of trying to achieve 100% test coverage across your application, particularly with F#. Instead, we'll look at applying some of the techniques and tools we've seen in this unit at a codebase that by now you are (hopefully) familiar with.

## **43.2 Writing API tests**

Let's start by doing a few simple API tests for our core bank API – in-memory tests that can prove some different cases of the API.

### **43.2.1 In Memory Testing**

You'll notice from Figure 43.1 that the "data store" area is greyed out. In other words, we're not going to be doing full end-to-end tests that "hit the database" – we'll need to create an *in-memory version* of the data tier so that you can easily set up different test cases etc. Let's refresh our memory as to what the data tier looked like in our application. First, let's look at save:

```
accountId:Guid * customerName:string * Transaction -> unit
```

This says that given an account Id, a customer name and a transaction, we can perform a side-effectful save for that transaction. Now let's see the signature to load all transactions:

```
customerName:string -> (accountId:Guid * Transaction seq) option async
```

This time, we give a customer name and get back *Some* accountId plus the associated transactions. If no customer exists, we get back *None* instead. And to further complicate matters, this function returns *asynchronously*.

### **Now you Try**

Let's create a function which will create an IBankApi but using a ResizeArray as a backing store – perfect for quick unit tests.

1. Create a new file, `Helpers.fs`, in your test project.
2. Create a function, `createInMemApi()`, which will create a test `IBankApi` whenever called.
3. You should use the existing `buildApi` function in the `Api` module to do the bulk of the work for you. This function takes in both a `load` and a `save` function (with the signatures identified above) and returns an `IBankAPI`.
4. The `buildApi` function is currently private, so won't be accessible to the test project. You can either: -
  - o Make it public so that it's visible to the test project (my preference)
  - o Make it internal and add an `InternalsVisibleTo` attribute to the project
  - o Implement the `createMemApi` function in that module

To get you on your way, here's a stub function for you: -

#### **Listing 43.1 – A stub function to create an in-memory `IBankAPI`**

```
open System
open Capstone8.Api

let createInMemApi() =
 let dataStore = ResizeArray() #A
 let save accountId owner transaction = () #B
 let load (owner:string) = None |> async.Return #B
 buildApi load save #C

#A In-memory backing store for transactions
#B Save and Load function stub implementations
#C Building the BankAPI using these in-memory functions
```

5. The `save` function should be very simple to do – simply add the supplied data as a tuple to the `dataStore`.
6. The `load` one will involve a slightly longer query. You'll need to filter out rows that aren't for the requested owner, and then pull out the account id from the list as well as the transaction elements (one way is to use a `groupBy` over the account id in order to easily pull it out, and take the first group that's returned – there should only be one group per customer).
7. Don't forget to "wrap" the result in an `async`. You can use the shortcut method `async.Return` to do this.

### **43.2.2 Example API tests in XUnit**

I'd advise writing a few different API tests to get your "muscle memory" up and running, such as: -

- **Create an account if none exists** – prove that you get a new account with an empty balance if you call `LoadAccount` for a non-existent customer
- **Multiple Deposits are stored correctly** – Make several deposits to a new account;

provide that the final balance on calling LoadAccount is correct.

- **Cannot withdraw if overdrawn** – If you go into an overdrawn state, trying to withdraw again returns a Failure result.

Here's the first test to get you going: -

#### **Listing 43.2 – Our first API unit test**

```
[<Fact>]
let ``Creates an account if none exists``() =
 let customer = { Name = "Joe" } #A
 let api = createInMemApi() #B
 async {
 let! account = api.LoadAccount customer
 test <@ account.Balance = 0M @> }
|> Async.RunSynchronously #C

#A Creating a sample input customer
#B Creating a clean in-memory api
#C Forcing synchronous evaluation of our test
```

Important things to note about this test are that firstly we create an in-memory api as part of the test – it's stateful, and we don't want the effects of this test shared with another test! Secondly, notice that we explicitly execute this test synchronously. XUnit actually has support for asynchronous tests, so you could also call `Async.StartAsTask` if you wanted (XUnit 2.2 will have native support for F#'s `async { }` blocks but at the time of writing is still in beta).

### **43.3 Testing the Web API tier**

Now that we've written some basic API tests, let's move up a level and test out our Web API controllers. In my view, your Web API controllers should be extremely thin – essentially a “mapping exercise” before your internal and public domains in both directions, and at most some extremely simple orchestration; anything more should probably be moved into your internal domain.

Testing out Web API controllers isn't especially difficult – you create your controller object as you would any other object, passing in any dependencies as needed (in our case, a Bank API which talks to an in-memory list). However, you do have to ensure that the Request and Configuration properties are correctly set on the controller object. Here's a function that will create a test controller with everything needed: -

#### **Listing 43.3 – Creating a Web API controller under test**

```
open Capstone8.Controllers
open System.Web.Http
open System.Net.Http

let createController() =
 let api = Helpers.createInMemApi() #A
 let request = new HttpRequestMessage()
```

```
let config = new HttpConfiguration()
new BankAccountController(api, Request = request, Configuration = config)
```

#A Creating dependencies for the Bank Account Controller class

### Now you Try

Create a set of tests for the HTTP layer. Again, I'd advise you to stick to focusing on the mapping between data from the web layer to the internal Bank domain model and back again. With that in mind, here are some examples you can do: -

- **Successful withdrawal returns OK** – Check that the Withdrawal controller returns `HttpStatusCode.OK` if the withdrawal worked.
- **Unsuccessful withdrawal returns Bad Request** – Check that the Withdrawal controller returns `HttpStatusCode.BadRequest` if the withdrawal was rejected.
- **Returns correct balance** – The balance of an account is correctly entered onto the response message

One thing that is interesting is that Web API methods that return an `IHTTPActionResult` rather than arbitrary objects (e.g. `PostDeposit` compared to `GetHistory`) need to be “unwrapped” to get the actual `HttpResponseMessage`. To save you some time digging around, here's a helper method that will do it for you: -

#### **Listing 43.4 – Unwrapping a Task<IHttpActionResult>**

```
open System.Threading
open System.Web.Http
open System.Threading.Tasks

let executeRequest (request:IHttpActionResult Task) =
 async {
 let! request = request |> Async.AwaitTask
 return! request.ExecuteAsync CancellationToken.None |> Async.AwaitTask
 } |> Async.RunSynchronously
```

### Mocking Libraries

It seems strange that we've gotten all the way here and not touched on mocking frameworks! This is a question that I do see coming up now and again, so it's worth briefly addressing. Firstly, you can absolutely use standard mocking libraries with F#, such as Moq (there's even a library designed specifically with F# in mind called Foq!). However, as you've seen throughout this book, oftentimes we're simply varying code by injecting individual *functions*, rather than entire objects and interfaces – in which case, it's trivial to mock them simply by creating arbitrary test functions by hand and passing them in!

There's also a tendency in F# to lean towards passing simple *data structures* around to *pure functions* rather than objects with behaviour on them – and again, these are generally extremely easy to test.

So, don't get hung up about mocking frameworks – you very, very rarely need them.

## 43.4 Property-based tests

OK! Almost done – there's just one more type of test to "try out"! We'll put in a couple of property-based tests for the Bank API, just to prove that we can.

### 43.4.1 Thinking about Property Based Tests

If you recall from the lesson on PBTs, we identified a few different ways to "discover" tests. I'm going to opt for a couple of simple tests to get you started, which will hopefully give you some good ideas of how to find further properties to test.

- **Going under 0 makes the account overdrawn** – If you make a withdrawal that pushes the account below 0, the account should become overdrawn.
- **Withdrawal fails if the account is overdrawn** – If the account is in an overdrawn state, trying to withdraw funds should fail.

These are actually very similar to the tests we wrote at the start! The difference here is that rather than use hard-coded inputs and outputs, you'll have to write general tests that prove the rule! In these cases, the logic to test should be fairly obvious.

### 43.4.2 Our first Property-based test

I'll get you started on the first test by at least providing the signature for you. It declares a test method that takes in a value that we'll use as the starting balance of the account. From there, it's up to you to write code to force the account to go overdrawn and then prove that the account is correctly identified as such.

#### **Listing 43.5 – A stub property-based test**

```
[<Property(Verbose = true)>]
let ``Going under 0 makes the account overdrawn``(PositiveInt startingBalance) = #A
 let startingBalance = decimal startingBalance #B
 // ... complete the test!
```

#A Using the FsCheck to generate only positive integers

#B Converting the integer to a decimal

The test implementation shouldn't prove too challenging: -

1. Create an instance of the test BankAPI
2. Make a deposit for the starting balance supplied
3. Make a withdrawal of (starting balance + 1)
4. If the returned result was Success (Overdrawn \_) then return true
5. Otherwise, return false (the test fails)

The important bit here is that we make a withdrawal amount *based on the value used as the starting balance*. By always withdrawing one unit more than was deposited, we can ensure that we always go overdrawn (indeed, you could simply start with an empty balance to try to

prove this, but starting from different amounts at least makes the test a little more interesting).

## 43.5 Summary

That's the end of the final capstone! You've written a set of unit tests using a set of different tools and techniques that we covered in this unit, and applied them to a domain that we've worked on throughout the book.

Here are some ideas for further types of tests you could write: -

- Write a set of tests using FSUnit
- Create a mock Bank API for the controller tests, instead of using the real Bank API logic. Use F# Object Initializers to avoid creating a "real" mock type!
- Read up on FSCheck's Model Based Testing feature (also known as Stateful Tests). Create a set of model-based tests that prove the Bank API correctly works with a random sequence of withdrawals and deposits.

# Unit 10

## Where Next?

Congratulations – you've made it! That's effectively the end of the "learn" phase of this book. Remember that this book was split into two distinct halves – the first half covered a *core subset* of the F# language, whilst the second half looked at applying F# in a number of real-world scenarios using different technology stacks.

Having worked diligently through this book, are you now an expert in F#? Probably not. However, what I'm hoping that you do have is the confidence to attack nearly *any* problem that you would have previously by default reached for C# without a second thought. Instead, you now know that you can use F# for just about any use case that you'd have looked at C# for, as well as a number of other areas that you previously might not have even considered .NET a viable option for at all.

Are you a functional programming expert? Again, probably not. Remember, in this book we focused on what I consider to be some of the *fundamentals* of FP – things such as expressions over statements, functions as values over classes and methods, and composition over inheritance. There's a whole world for you to find out more about FP and its use within F# – compare this to learning more design patterns in the OO world to "enrich" your ability to effectively solve problems.

The final unit of this book doesn't contain many code samples, and won't contain any practical exercises. Instead, it consists of a number of small appendices that aim to address questions or issues that you may have: -

- How can I convince my boss to give me permission to use F#?
- What's the best way to start using F# in an existing system and team?
- How can I learn more about F#?
- What features of F# *didn't* you cover?

So, without further ado, let's dive in!

# A

## *The F# Community*

**As of 2015, Microsoft have openly become fans of moving to an open source model. Lots of .NET has now been open sourced, and the impending delivery of CoreCLR – which includes a lighter, cross platform version of the existing CLR – suggests that Microsoft see the long term future of .NET as running on equal footing on Windows, Mac and Linux. There's a long way to go though, in my opinion, until Microsoft truly adopts open source though – there's a world of difference between simply putting a repository on GitHub and embracing a community-led approach to development. Nonetheless, it's something to be applauded.**

If you're not familiar with working with open-source, community-led projects, doing so can be a great learning experience. Not only are you exposed to different coding styles and techniques, but you also learn about a different way of developing software, through features such as a pull requests and (almost certainly) through Git and GitHub.

### A.1 The F# community

F# is way ahead of the curve here in terms of the .NET community – the language and compiler itself was open sourced several years ago, and it has a vibrant, active, passionate (some might say too passionate!) and growing community behind it. There's channels of communication through twitter, hangouts, real world meetups, mailing lists and websites (see Appendix 4). But what does this mean for you as a software developer in terms of day to day development, sourcing reusable libraries and working with others?

#### A.1.1 Microsoft and F# Libraries

Firstly, you'll have to get used to the idea of not relying on Microsoft providing every library, framework or tool to you. Instead, many of the libraries and tools you'll use are open-source, run by the community, for the community. On the one hand, this means that many people do work on these tools in their spare time rather than as a full-time job, and don't have the

obvious financial backing of Microsoft. However, I should point out there are many people that are able to contribute to open source projects as part of their day to day job, simply because their organisations use (or maintain) those tools. On the other hand, you'll have open, direct access to the code base, can make fixes / enhancements yourself, as well as encourage others to chip in, too.

Think of all of the F# libraries we've seen in this book: FSharp.Data, Paket, Suave, XPlot or FSCheck etc. – *all* are open source projects that are owned by individuals within the F# community – not Microsoft. The *community*, not Microsoft, dictates where the tools go – and ultimately the ecosystem of tools that are available. This mentality is a sort of “survival of the fittest” – it means that tooling can rapidly evolve, benefitting everyone. This is very, very different to the typical C# library stack of e.g. Nuget, Entity Framework and ASP .NET etc., all of which are ultimately owned by Microsoft. If Microsoft decides to make a change to this tools, whilst you may be able to comment on the direction it takes, ultimately, you'll have little choice but to accept the decisions that are made inside Redmond.

Note that the F# community *includes* the Microsoft F# team, who contribute to these tools as well – but they do not *own* them. In this way, I think that the F# community stands out a little from the C# and VB .NET communities in that there's a much higher proportion of active involvement from the users of F# in community activities. Part of this is drawn out of necessity – because Microsoft don't invest quite as much in F# as, say C#, the F# community has matured extremely rapidly and is remarkably self-sufficient. Frankly, this attitude is something that the C# and VB .NET communities need to do as well in the future order to ensure that those languages (and associated libraries) evolve in the right way and stay relevant to them, particularly with the move to .NET Core. A few years ago, there was a similar movement within the C# community known as the “alt .net” movement. It encouraged developers to not only look to Microsoft for tooling and libraries but to the community itself. It kind of died off a few years ago, although it's looking like it might resurrect itself now with the coming of .NET Core.

**Impressions of the F# community**

The F# community has become known in some circles within the .NET community – somewhat unfairly in my view – as the “awkward sibling” that works in a different way to the rest of the .NET ecosystem. Part of this is borne of the fact that the F# community is used to simply fixing things that aren't up to scratch in order to improve tooling or process themselves; we don't wait for others to do things for us. The flipside is that the community is often seen to be going “against the grain” of established tools and processes, apparently just for the sake of being different. We'll cover more of this in Appendix 3 where we look at specific libraries and tools that make up a key part of the most common F# tool-chains, but the F# community has alternatives to many of the common tools and libraries that you might use today.

### F# Elitists

There's also a tendency to think of the F# community as somewhat "elitist", looking down upon the "inferior" C# / VB .NET developers who haven't "seen the light". There's unfortunately an element of truth here, as occasionally people in the community have allowed their enthusiasm for F# to bleed over into maligning those languages.

By and large though, whilst many people in the F# community do believe that F# offers a superior way to solve many types of problems than C# or VB .NET, most don't feel the need to disparage them.

In the past, this led to the relationship between the community and Microsoft being somewhat uncomfortable; there were times when Microsoft simply ignored many of the excellent .NET technologies that were created by the F# community simply through lack of awareness or a misconception that F# libraries couldn't be used by .NET in general. Other times, Microsoft teams actively attempted to thwart tools being created within the F# community which ultimately led to public squabbles on social media. Hopefully, these days are now past us!

Within Microsoft, there's a growing acceptance and acknowledgment that F# has a valid place within .NET today in many different spaces, and have a valuable role to play in moving .NET out from the closed world it inhabited for many years into the open.

### The F# Foundation

The F# Software Foundation (<http://foundation.fsharp.org/>) is a not-for-profit organisation whose goal is to "promote, protect, and advance the F# programming language and facilitate the growth of a diverse and international community of F# programmers". The foundation has a board which is voted for by the members of the foundation on an annual basis, whose responsibility is to help shape and promote the evolution of F# as a whole. The Foundation run several working groups and programs, such as a 1:1 mentoring program, core engineering and communications, as well as help connect people looking for speakers and those giving presentations together.

## A.2 Coding in the open source world

As an example of what I mean about community involvement, let's take a simple scenario – you're using some open source library and find a bug with it - or maybe you have a feature request to improve it. How do you report this? It's actually pretty easy – you simply go to the GitHub repository for the library and raise an issue; usually someone that maintains the project will respond within a few hours – sometimes less. Sometimes the feature will already be there. Other times it won't be appropriate, and other times it'll be accepted and placed on the "to-do" list. At this point, rather than simply waiting until one of the maintainers makes the appropriate changes – particularly if the change is relatively small - you might be asked to submit a "pull request" with the changes yourself! The maintainers will usually help you by providing the area of code that needs to change, and then suggesting what you might do for the change. Although it would undoubtedly be quicker for the maintainers to make the change themselves, by getting another contributor involved, there's a reason why they'll be keen to get a new contributor onto the project: This grows the team of people that can help with the

project; the next time you have a feature request or bug fix, you'll already have the source code on your machine, and it'll be much quicker for you to make a change.

### A.3 A real-world example of open source contributions

Here's a real-world illustration of three people on Twitter discussing a feature request for the FSReveal project: -

29<sup>th</sup> January 2016

**08:39 [PI]:** Hello #fsharp nation! Is there any way to do links between slides in #fsreveal?

**10:03 [IA]:** Can't they just point to the uri (every slide has a #page bit in the URI)?

**10:04 [PI]:** Yes, but I'd like the URI to be dynamically generated. Define an anchor in a slide and reference it from another one ☺

**10:05 [PI]:** But I know the correct answer is to open an issue on Github and write a pull-request!

**10:13 [SF]:** Maybe there is already a trick in reveal.js?

**14:18 [PI]:** Indeed there is!

30<sup>th</sup> January 2016

**10:25 [PI]:** And here comes the pull request ☺

**11:27 [SF]:** Awesome!

Notice the use of the *#fsharp* hashtag – designed to get the attention of the active F# community on twitter. From there, it takes around 24 hours for an idea for a feature first being formed and ultimately to the solution being submitted as a pull request into the proper codebase.

Let's see the actual pull request in more detail (Figure 1.3). In this case, having already spoken about a new future for this project on twitter with a couple of maintainers of the project, **pirmann** submits a pull request (PR) with the code changes required for a specific feature. **forki** (one of the project maintainers) then replies and asks for some documentation to be added; shortly after it arrives. Only then does forki "accept" the pull request - and then thanks pirmann for his contribution to the project. He might have also done a quick code review of the changes to see what's actually been added. Total turnaround time from idea to release in this case was around 24 hours – not bad!

Pull requests are a low ceremony, quick and easy way to help shape a system that you yourself use and benefit from. The feature may not have been particularly large, but by accepting lots of small PRs, a system can rapidly grow in features and complexity.

Again, this collaborative approach is something that might be completely foreign to you; you might even find it rude the first time someone on a project asks *you* to do a fix *yourself* –

but this attitude is actually perfectly normal. Working together in this manner empowers us all to shape libraries and frameworks as we see fit, and with a much quicker turnaround than you might be used to. Of course, this quick turnaround means that there's often rapid change within a library, which can be unsettling, as well as the increased risk of bugs – some projects are more “bleeding edge” than others and are willing to push out new versions of tools much more quickly (the flipside is that bugs are often fixed much more quickly, too!).

Let's see the actual pull request in more detail (Figure 1.3). In this case, having already spoken about a new future for this project on twitter with a couple of maintainers of the project, **pirmann** submits a pull request (PR) with the code changes required for a specific feature. **forki** (one of the project maintainers) then replies and asks for some documentation to be added; shortly after it arrives. Only then does forki “accept” the pull request – and then thanks pirmann for his contribution to the project. He might have also done a quick code review of the changes to see what's actually been added. Total turnaround time from idea to release in this case was around 24 hours – not bad!

Pull requests are a low ceremony, quick and easy way to help shape a system that you yourself use and benefit from. The feature may not have been particularly large, but by accepting lots of small PRs, a system can rapidly grow in features and complexity.

## Nested slides data #90

Merged forki merged 3 commits into fsprojects:develop from pirrmann:nested-slides-data a day ago

Conversation 3 Commits 3 Files changed 7

**pirrmann** commented 2 days ago

These changes allow to have slide properties on nested slides. It can allow better styling of individual slides and also enables linking to named slides using properties (id=SomeSlideId)

**pirrmann** added some commits 2 days ago

- Presentation properties shouldn't be repeated on nested slides cd0a68d
- Allow properties on nested slides af79947

**forki** commented 2 days ago

Can you please add some notes to the docs as well?

- Named slides documentation update ef2d39b

**pirrmann** commented 2 days ago

Here comes the documentation!

**forki** merged commit 13c429e into fsprojects:develop a day ago Revert

**forki** commented a day ago

Thanks a lot!

Figure A.1: A real-world example pull request from the FSReveal project.

## A.4 Summary

The F# community is friendly, welcoming and enthusiastic. However, it also operates on a different mindset than you might be used to if you're comfortable with the approach that we should adopt – and rely upon – everything Microsoft serves us. If you can accept that, and are interested in becoming an active member of the F# community, you'll find it's a fun, vibrant and extremely productive way to collaborate and get things done!

# B

## *F# in my organisation*

**One of the most common difficulties that developers have once they've "tasted" F# is actually adopting it in the real world in their day-to-day job. I see this happen in three ways: -**

- People find it difficult to make the leap from writing scripts and console applications into integrating with the full .NET stack - things such as interop between C# and F#, working with NuGet as well as using frameworks such as ASP .NET Web API etc.
- Convincing others of the value of using F#, and ultimately getting permission from colleagues or management to give F# a go.
- Understanding how to actually start using F# in a practical sense in an existing tech stack.  
What areas are safe bets to start using F# that will show its strengths? How should you start applying F# within an existing solution?

I'm hoping that over the course of this book you've seen and gained confidence in carrying out tasks that fulfil the first of these three questions, and can see enough to assure you that F# works just fine in basically the same contexts as C# or VB .NET. So, this appendix deals with the latter two points – dealing with common misconceptions regarding F#, and how to start incorporating it in your existing tech stack.

### **B.1 Introducing F# to others**

Let's first discuss the "human" element, with a number of pointers on how to show F# to the rest of your team, or your boss, in such a way that they're happy for you to start adopting it within your organisation, and how to answer many of the common fears of adopting F#. But before reading this, have a quick watch of this tongue-in-cheek YouTube video that demonstrates a common outcome to such a conversion here [https://www.youtube.com/watch?v=Hd9Z9s4\\_DII](https://www.youtube.com/watch?v=Hd9Z9s4_DII) – read on to see ways to avoid this happening to you!

### B.1.1 Show the positives of F#

First and foremost, I'd encourage you to start by simply stating the overall benefits of F# in simple terms that all developers can understand – things such as improved productivity, reduced bug rates, and the ability to write code that is easy to reason about. Give tangible examples such as showing how easy it is to work with data in F# using type providers; give examples that relate to concerns or difficulties you might be facing in your current project or had in the past, such as areas where bugs crop up often and F# could help, or where you're doing something such as CSV parsing etc. Features such as exhaustive pattern matching and discriminated unions, or even simply structural equality with records – all of these are relatively simple ways of illustrating some of the quick wins of F#. There are many useful online resources such as demos or presentations that people have done in this vein that you can utilise.

### B.1.2 Avoid dismissing C#

By far the biggest mistake people make when showing F# to their colleagues is to suggest that C# is somehow obsolete or "bad". There are two reasons for not doing this: -

- Firstly, no-one likes being told that something they've invested years learning and growing fond of is somehow "wrong" – it becomes an emotional discussion that's not worth spending time on.
- Secondly, C# is not suddenly a "bad" language. As far as modern OO languages go, it's probably one of the best ones out there. The question is more about whether using it in specific areas is going to be the best fit for you and your team compared to working with perhaps a hybrid stack that also makes use of F#.

There's nothing wrong with being enthusiastic about F# - the trick is to show the benefits to F# without presenting this as a direct comparison to the "shortcomings" of C#. Instead, focus on the strengths of F# as a language on its own, and then as an afterthought relate this to how you'd perform the equivalent task in C# (avoiding straw man examples wherever possible). Ideally, show some real examples of your existing codebase and how you might have approached the problem differently using F#, and where the pain points are. Explain how if you like working with mutable data, statements and side effects with inheritance, then F# probably *isn't* a great fit – but if you like working with expressions that can be reasoned about, simple functions that compose together, and lightweight modelling without the need for a myriad of design patterns, then F# might well be a better fit.

### B.1.3 Dismiss the zero-sum game

A common misconception about a "new" way of doing things – particularly something like F# or FP in general, in which there's sadly a lot of fear / uncertainty / doubt (FUD) out there – is that to gain the benefits of F#, there must also be a cost associated with it. For example, it

can't be "ready" for production usage, or it can't possibly be used for general purpose programming.

Explain to colleagues that this simply isn't true and that, yes, it's entirely possible to reduce bug rates *and* increase developer productivity *and* improve developer satisfaction at the same time! There are plenty of case studies available online (see Appendix 4) that you can use to prove this as well; is your company or development team really so unique that these studies somehow don't apply to your organisation?

#### **B.1.4 Reduce the fear of learning**

It's often taken as read that there is a high cost to F#. Again, this idea perpetuates the zero-sum-game theme – that it *must* be hard to learn F# if the benefits are so great! Yes, there is an up-front cost to learning F#, but then by the same token, there was an upfront cost to learning LINQ in C#3, or async / await in C#5. Because F# sits on top of the .NET framework, you can re-use virtually all of the frameworks and libraries that you already know – we're just going to be *orchestrating* the use of them against data in a different way. Explain that there are ways to start small (see section 2.2) and gently introduce the use of F# without the need to throw away all your existing code. And best of all, you're holding a book which is a great way to start learning F# (although perhaps you'll need to buy a few more copies of it)!

#### **B.1.5 We're already productive enough!**

More often than any other argument against adopting F# I hear, is that a team is "already productive enough in C#". This is also one of the the most ridiculous arguments of all. By what yardstick is your team measuring itself against? Have you arbitrarily decided that it's going to be impossible to make any more productivity gains in the future? Are your team not even *interested* in exploring the possibility in reducing the cost of development for your organisation?

Ask what areas of development they would consider for using F#; if the reply is "maths or science" then you can be sure that this is simply hearsay; if the reply is "only when we're doing something especially difficult", you can dispel this easily enough by saying that whilst F# *does* allow you to solve *difficult* problems with *simple* code, F# *also* allows you to solve *simple* problems with *really simple* code!

#### **B.1.6 Specific use cases**

Your product or project might have some areas that are especially applicable to F# - areas around data manipulation, complex modelling, or working with scripts, where C# doesn't necessarily fit especially well. You can explain, with real examples, how F# would eliminate a bottleneck in your process or current development project. The problem with going down this route is that you run the risk of F# being thought of as *only* being applicable to solving problems in that specific area. This means you might find it difficult to "break it out" later on.

On the other hand, if it's a roaring success, you can always use it as a springboard from which to justify adopting it in other areas.

### **B.1.7 Cost / benefit analysis**

Particularly when dealing with decision makers rather than pure technologists, those people that hold the purse strings will need to see why F# is of interest to them. There's always an inherent risk with doing anything new or different, so you'll need to explain in *non-technical* terms why this is important for your organisation.

Look at F# as providing your company with a competitive edge over others; explain how F# can reduce development costs both in terms of time-to-market as well as maintenance costs. Show how F# is consistently one of the most loved languages in developer surveys which could lead to improved staff retention. For example, in the 2016 Stack Overflow survey, F# was third placed most loved language, with an approval rating almost 10% higher than C#. There's also some evidence that developers are willing to take a pay cut in order to work on F# as they understand the benefit it'll give them in their careers.

Jet.com is a great case study of a large organisation that bet the farm on F#, building their *entire* back-end stack in F# on top of Microsoft Azure. 18 months after launch, they were bought out by Walmart for a whopping \$3.3bn, due in no small part to the technical assets they'd developed.

### **B.1.8 Hiring new staff**

This is another common concern – and a fair one. If you look at the number of available jobs out there for F# positions, it'll always be lower than C# ones – no surprises there, since C# is used around 100x more than F# currently according to high level figures from Microsoft.

But that's no reason to simply avoid using it - developers can always cross-train to F#. In my experience, when using F# day-in, day-out, it takes a few weeks for a competent developer to become reasonably productive in the language, and a few months to be completely comfortable in the language (this time can be reduced further if the developer is already familiar with either .NET or functional programming). It's certainly nothing that I would suggest should be a major blocker, although I have seen this concern block adoption of F# before.

### **B.1.9 Simply start using it**

This approach is somewhat riskier, and depends on the level of control you have within your organisation. I know of developers who have simply started using F# without even asking the customer or management for approval. Potentially this is a high-risk strategy because if things don't work out – for example, you struggle with a specific part of the project, or for some other reason the project isn't a success and someone else blames it on "that weird language" (this has been known to happen!) – you could end up facing some difficult questions. Also, bear in mind that doing this could mean going against the wishes of other team members who

might have a vested (and even emotional) interest in staying with a language that they feel comfortable in.

The flip-side to this is that if things go well, you could end up being the superstar – someone who was able to “get things done” by using the right tool for the job. Getting noticed like this will probably also win you points with management, as someone who was able to take personal responsibility and showed initiative to make a positive difference in the company.

## B.2 Introducing F# to my codebase

OK! You’ve got buy-in from the powers-that-be to start using F#, and your fellow devs are all eager to start trying out this F# – ideally on your current project. The question now is, how do you start? This section outlines a few different approaches that you can take in order to adopt F# in an existing team and project.

### B.2.1 Exploratory Scripts

Probably the easiest way to start using F# is with scripts. A script allows you to start to learn the language in a relatively low-risk manner, whilst quickly building confidence in the language as well as getting immediate benefit. In my experience, there’s always a need for scripts within a project (even a project whose codebase is 100% C#) – whether it’s to quickly test out a subset of the application (instead of the dreaded console application), or whether you want to quickly try out a new NuGet package. Alternatively, you might also use a script to aid with bug-fixing or fault reproduction – to quickly identify a record in a database, hit an HTTP API and then call some code with a mashup of both. You’ll be surprised just how quickly you’ll naturally start to use a variety of F# features even within a script!

Note that as of VS2015, there is *some* support for scripts in C# (as well as third-party tools and systems) and this will no doubt improve in time, but for the moment F# scripts reign supreme in terms of experience. If you acknowledge that you’d like to use this not only to provide business benefit but also as a means to help learn F#, scripts are probably the lowest hanging fruit.

### B.2.2 Ad-hoc processes

There are nearly always some ad-hoc processes needed within a system, which may also take the form of scripts, but rather than as a means to *explore* code etc., they might be something to aid the day-to-day running of your system. For example, this may include a maintenance process which clears down certain records in a database based on the result of another query, or the results of a text file that needs to be parsed that’s delivered on a daily basis from a third-party. Or you might use it to generate a report once a day, every day by pulling in data from your source systems and emitting a PDF or HTML report (see Appendix 3 for details on the FsLab project).

### B.2.3 Helper modules

If you're ready to start bringing F# into your main codebase, you have a number of options. The easiest one to start with is simply to start writing standalone helper functions or modules of functions - these show up as static classes in C# so are easy to work with from a consumption point of view, and can start to build confidence that you can write hybrid applications within your team.

However, beware of "underselling" F# here – it's easy to think that F# is great for small functions that take a number and give back another number, but can't be used for anything more than that. You'll also be somewhat hamstrung in that you won't be able to use F#'s type system for modelling etc. if you simply rely on your C# domain model – again, this is missing out on much of the power of the language.

### B.2.4 Horizontal Tiers

One way of providing a way for F# to flex its muscles a little more is to say that from an architectural perspective, a single tier of the application will be written in F# - for example, the validation tier, or data access layer, or even a large module within a system – for example a calculation engine etc. In such a case, you can provide a contract that you expose to consumers (and even wrap in a lightweight OO interface to make it "natural" to consume from C#), but internally be free to use the full power of F#, with a rich internal domain model built on discriminated unions, with exhaustive pattern matching and type providers etc.

In the proceeding figure, you can see a fictional web application that uses F# for the Validation and Business Logic, whilst leaving C# for implementing the MVC Controllers and data access. Note – I'm not suggesting that this is a recommended partition of concerns to language, but merely an example!

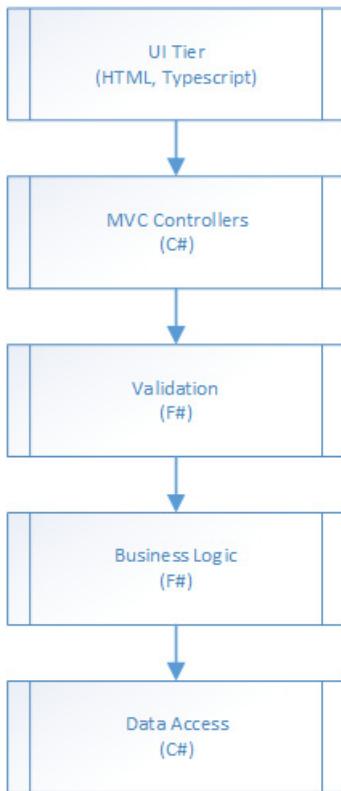
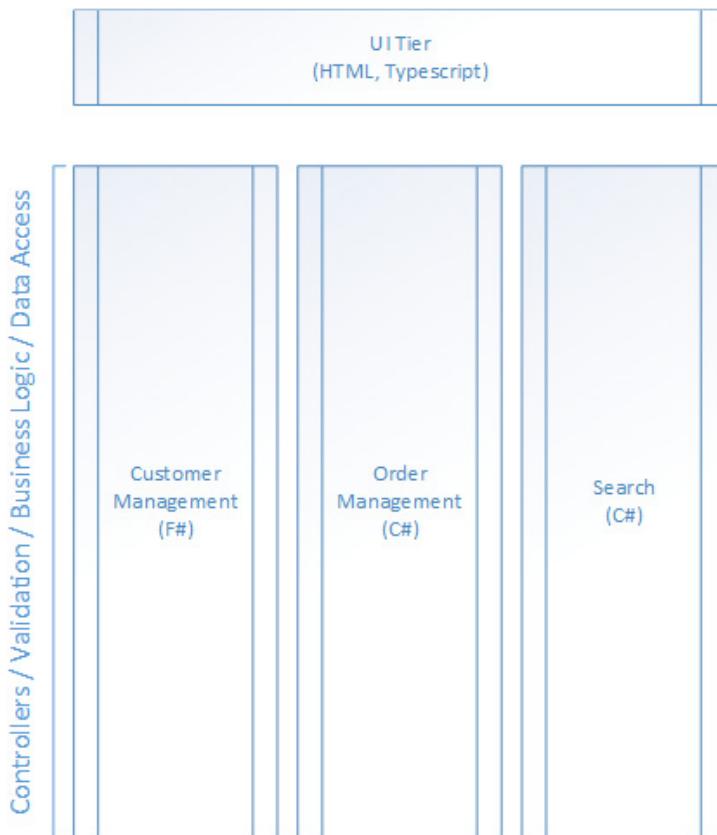


Figure B.1 – A sample hybrid language application stack with different languages used for different tiers

Adopting this means that you can “pick and choose” which areas you’d like to tackle one at a time, and gain the benefit of the F# code across all the different modules within the application. However, if this is an existing application, you’ll need to gently “phase in” the F# code to replace the C# code – this might prove challenging if your team are actively developing at the same time across the application.

### B.2.5 Vertical Tiers

An alternative to partitioning by horizontal tiers is to pivot this, and partition by *vertical* tiers. In this case, you provide a full end-to-end stack implementation in a single .NET language, but only for specific business areas of the system – for example, the customer module, or the order management module. We could therefore re-organise our architecture as follows: -



**Figure B.2 – Partitioning an application into vertical tiers**

This approach has several benefits: -

- You can see how a "full" F# stack works compared to an equivalent C# one. Is it any easier to work with? What are the pain points?
- The next time you create a new area of the system, you can adopt this without treading on anyone else's toes or affecting the existing product.

The cost of this approach is for "cross-cutting" code e.g. logging or caching etc. you might have to end up having two versions of code that do the same thing, one in C# and one in F#. It's also somewhat bolder than the horizontal version, as you'll rely on a new language for an entire business area of the system.

Of course, you could try a hybrid approach – perhaps you create some cross-cutting concerns in F# such as a cache layer, and perhaps a sub-section of the Customer module in

F#, with some in C#. However, you'll need to be careful that it doesn't turn into a patchwork of seemingly random language choices across your application stack!

One thing I can tell you from experience is that working in hybrid stacks often leads to a "bleed" of F# code into what is historically the "C# stack", simply because the ability to rapidly and accurately model domains in F# quickly becomes addictive. As you move between the two languages, you'll probably find yourself wanting the F# side of things to expand across the stack as you miss the extra type safety and security that the language affords you.

### B.2.6 Unit Tests

I'm actually not a huge fan of using F# for unit tests, but some people do swear by this. Yes, you can certainly use F#'s backtick methods to make tests more readable, and can perhaps take advantage of its succinct syntax to write tests more rapidly than before. However, you'll still be working with an API that is almost certainly object oriented and won't take advantage of F# features such as currying, so will be somewhat restricted to what you can do.

The only exception to this is with using something like Canopy, which has a specific DSL that makes writing tests very, very easy. However, the flipside is that because Canopy has such a specific DSL (with several custom operators and global mutable variables), it can be difficult to know where "F# the language" stops and "Canopy the DSL" begins.

### B.2.7 Build scripts

FAKE is a popular project in .NET solutions to provide a succinct, simple-to-use build system that can, in effect, replace Team City or similar in terms of build orchestration (see Appendix 3 for more details). As your build process is completely standalone from your application code, you'll be isolated from the main code base, so in one sense it's relatively low risk. However, again, because FAKE provides a DSL on top of F# with custom operators etc., it can be tricky for beginners to F# to know what is FAKE and what is F#.

## B.3 Summary

Adopting F# in your team is a two-step process – firstly getting buy-in from those that you work alongside with (and under!), and secondly having a plan in which to ease F# into your existing development process and stack. There's no right or wrong here – much depends on your team's appetite for learning something new (and the apparent risk of adopting something different), as well as the ease in which you can integrate F# into your product stack. Spending a few days in advance trying out a "proof of concept" (or speaking with someone that's done this before) is always a good idea!

# C

## *Must-visit F# resources*

One of the goals of this book was not simply to give you confidence in using F#, but also to give you the ability to join the F# community and to learn new areas within F# yourself. The objective of this appendix is to point out some of the many excellent on-line resources that you can take advantage of that will enrich your use of F#.

### C.1 Websites

There are many websites dedicated to F#; here are a few of the most popular ones that you definitely will want to check out.

#### C.1.1 FSharp.org

<http://fsharp.org/>

The official home of F#, this contains all sorts of goodies, from customer testimonials to getting started guides on a variety of platforms to helpful guides in a number of areas. There are also several areas on the F# Foundation, including how to become a member, how to contribute, and taking part in some of the programs that the foundation runs.

#### C.1.2 Community for F#

<http://c4fsharp.net/>

The C4FSharp site aims to provide a unified list of information regarding events and user groups happening in the F# world. It also contains a list of webinars and recorded user group talks demonstrating various aspects of F#. If you're looking to go to a meetup or user group near you, there's a good chance it'll be registered here!

Secondly, the site also contains a number of great coding "dojos" to do – these are challenges that usually last a few hours with a set task and model solution, usually best done in groups or pairs, often in a user group session.

### C.1.3 F# for Fun and Profit

<http://fsharpforfunandprofit.com/>

There's no way you could use F# and not know about this site. F# for Fun and Profit has been running for several years and contains a large number of educational series of posts on various aspects of F#, from error handling to domain modelling to more complex areas of F# such as monads. The set of series has evolved organically over time, so there's not necessarily a clear "path" in terms of the series, and the content is very detailed – many of the posts have a large number of examples that explain (very well!) the concepts on offer. So, whilst it's not a site that you might quickly look at for a 5-minute answer to a specific question, the concepts it deals with are definitely worth your time to read. The series on error handling and railway-oriented programming alone is worth it, but there are literally dozens of series on there to read up on. Scott Wlaschin (the author of the site) is also a regular – and excellent – speaker on F#, so if he's in a user group in your area it'd be well worth your time to go listen to him.

### C.1.4 F# Weekly

<https://sergeytihon.wordpress.com/category/f-weekly/>

F# Weekly acts performs an excellent news aggregation function for F#, based on new libraries that have been released, news on the the F# language, blog posts and videos as well as generally anything that's "happening" in the community. It's definitely worth subscribing to it – it's probably the easiest way to quickly catch up on what's been going on in F# over the past 7 days.

## C.2 Social Networks

If you're looking to quickly interact with the rest of the F# community – whether it's to ask someone's opinion on something or for advice with a problem you're facing, F# has you covered with a great presence on a number of social mediums.

### C.2.1 Twitter

<https://twitter.com/hashtag/fsharp>

The #fsharp hashtag is your friend on Twitter. Whether it's someone's ideas or news on F#, a new library that's just been released, or the latest debate as to whether F# should adopt type-classes or not, it'll almost certainly have a discussion here.

### C.2.2 Slack

Slack is an excellent browser-based medium for discussions that need more than just 140 characters in a tweet, allowing users to chat in groups on a variety of topics.

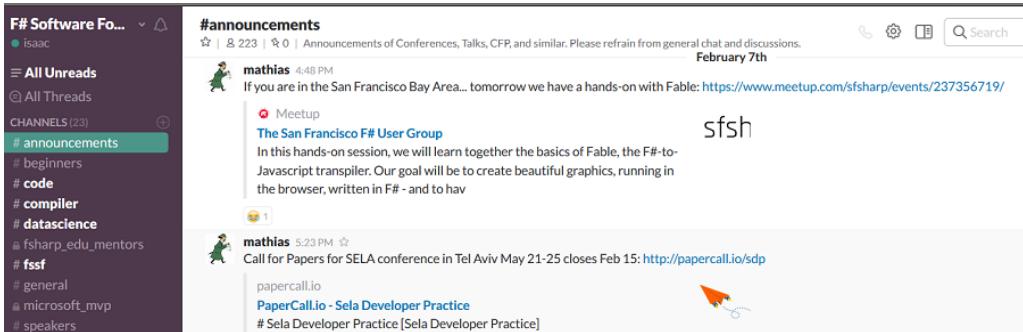


Figure C.1 – The F# Software Foundation Slack channel

There are two Slack forums for F#:

- <https://fsharp.slack.com/> - The F# Software Foundation slack channel, with chat rooms for beginners, general discussions, data science to the compiler.
- <https://functionalprogramming.slack.com> – An “unofficial” F# channel, but also equally popular.

Many of the F# community will crop up on here as well, so it’s often worth having both slack channels open in separate tabs in your web browser just in case!

### C.2.3 Reddit

<https://www.reddit.com/r/fsharp/>

The F# sub-reddit contains many news items and discussion topics that are worth looking into, on a wide variety of topics. If you use the reddit website already, this is a good sub-reddit to belong to.

### C.2.4 The F# Mailing List

[fsharp-opensource@googlegroups.com](mailto:fsharp-opensource@googlegroups.com)

There’s also the F# google group and mailing list that you can subscribe to. Since the increase in popularity of Slack, the content on the group has dropped a little, but there are still interesting discussions to be had on it – plus the F# Weekly is always sent here, so this is an easy way of getting weekly news on F# delivered to your inbox!

## C.3 Projects and Language

The last area we’ll look at are some source code repositories that are important for F#.

### C.3.1 The F# Compiler

There are two repositories for F# - the Microsoft Visual F# repository, which contains the core compiler and the open-sourced tooling elements (<https://github.com/Microsoft/visualfsharp>), and the “open” edition of the core language and tools (<https://github.com/fsharp/fsharp>). The former feeds directly into Visual Studio and the official F# Nuget packages, whereas the latter is based off the former (theoretically they should always be the same) and then feeds into many of the cross-platform initiatives such as Mono. In time, the two repositories will hopefully be merged into one, but I would suggest that for any issues (or if you want to keep up to date with the changes that are currently being made to the F# language and compiler), you look at the Microsoft repository. Pull requests to the repository are definitely accepted and encouraged, although the tool chain for the compiler is quite complex, so you’ll need to have your wits about you before you dive in and start to add higher kinded types to F#.

### C.3.2 Language Suggestions

The github repository at <https://github.com/fsharp/fslang-suggestions> is nowadays used for managing suggestions to F#, and is curated by both the Microsoft Visual F# team and Don Syme (the creator of F#). You’ll see many different language suggestions on here, all of which will be read by the F# team. You can also participate in existing issues by upvoting those that you agree with – this helps the team understand demand from developers for new features in the next version of F#.

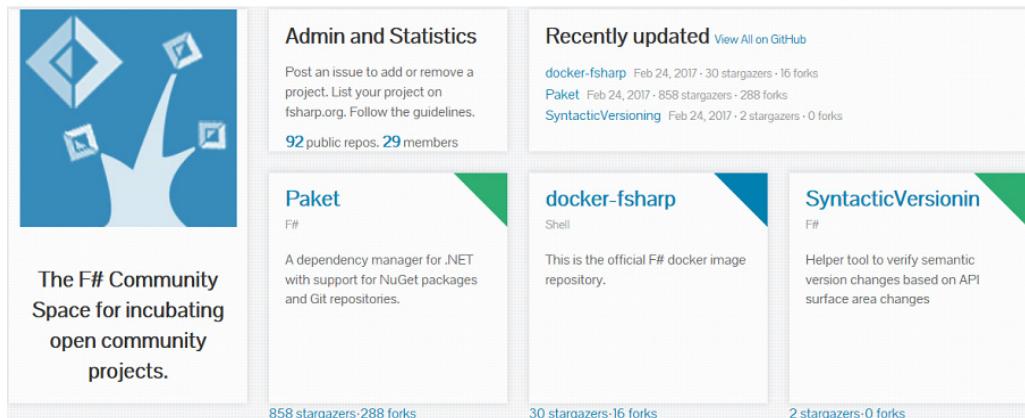
The screenshot shows a GitHub repository page for 'fsharp/fslang-suggestions'. At the top, there are filters for 'Clear current search query, filters, and sorts', 'Author', 'Labels', and a 'M' dropdown. Below this, a summary shows '156 Open' and '386 Closed'. The main area displays five open issues:

- Support type classes or implicits** (old votes:300+) #243 opened on 20 Oct 2016 by baronfel
- Allow intrinsic type extensions for provided types** (area: type providers) #509 opened on 5 Nov 2016 by 7sharp9
- Allow shorthand accessor functions** (old votes:300+) #506 opened on 3 Nov 2016 by wastaz
- Simulate higher-kinded polymorphism** (old votes:300+) #175 opened on 20 Oct 2016 by baronfel
- Allow type providers to generate types from other types** (area: type providers, old votes:101-150, started) #212 opened on 20 Oct 2016 by baronfel

Figure C.2 – Looking through some of the issues on the F# language suggestions repository

### C.3.3 FS Projects

The FS Projects website (<http://fsprojects.github.io/>) is a kind of semi-official list of popular F# projects that are on NuGet and GitHub.



The screenshot shows the FS Projects website interface. On the left, there's a large logo featuring blue diamond shapes and the text "The F# Community Space for incubating open community projects". To the right, there's a section titled "Admin and Statistics" with instructions on how to post issues and guidelines for listing projects on fsharp.org. Below this is a summary of "92 public repos" and "29 members". A "Recently updated" section lists three projects: "docker-fsharp", "Paket", and "SyntacticVersionin". Each project card includes a thumbnail, the project name, its technology stack (e.g., F#, Shell), a brief description, and its GitHub statistics (stargazers and forks).

Project	Technology	Description	GitHub Stats
docker-fsharp	Shell	This is the official F# docker image repository.	30 stargazers · 16 forks
Paket	F#	A dependency manager for .NET with support for NuGet packages and Git repositories.	858 stargazers · 288 forks
SyntacticVersionin	F#	Helper tool to verify semantic version changes based on API surface area changes	2 stargazers · 0 forks

Figure C.3 – The FS Projects website.

FS Projects is definitely worth having a look through here if you're looking for a specific library, but it's by no means a complete list – whilst Appendix 4 contains some libraries from here, there are many that are not included here, too.

## C.4 Summary

This appendix contains some of the key F# resources online. There are no doubt others that could have been added, but I'd suggest that these are a great base for you to start engaging with the F# community, learning from others, and sharing your knowledge and experiences with the community at large.

# D

## Must-have F# libraries

This appendix contains a list of popular libraries and tools within the F# ecosystem. It's by no means an exhausting list of all libraries out there – and there are always new libraries cropping up - but should have enough in it to give you a few ideas of avenues to explore of how you can start using F# in new and interesting ways.

### D.1 Libraries

First, we'll cover a whole set of F# libraries which we've not yet touched upon in this book – but don't forget the ones we have done already, such as Paket, FSharp.Data and Canopy! Note that many of these libraries are just that – libraries, *not* frameworks. This means that they can be used interchangeably in a flexible manner, without forcing you down a specific path. They're nearly all open source and free to use (available on GitHub and Nuget), and most work cross-platform without relying on Visual Studio tooling – they're just .NET assemblies that you can use as needed.

#### D.1.1 Build and DevOps

F#'s unique syntax, scripting and language features make it a great choice for using as part of (or to replace!) your build pipeline. The ability to create custom operators enable some impressive tricks that fit very well as a replacement for MSBuild and Powershell as a language to orchestrate and execute build pipelines.

##### **FAKE**

<http://fsharp.github.io/FAKE/>

FAKE is a build automation system that you can use to perform entire build and release pipelines. It comes with a large set of helper libraries to perform common tasks, such as file copying, building projects, assembly versioning, configuration file rewriting, running unit tests

- all sorts of goodies. It can, in effect, replace your reliance on a central build platform such as Team City and allow you to run builds locally as well as remotely. Builds are made up of Tasks, which are simply an arbitrary function that has a name – these are then composed together using the FAKE DSL into a pipeline.

```
"Clean"
 ==> "BuildDatabase"
 ==> "DeployCIDatabase"
 ==> "ConfigureDbConnection"
 ==> "BuildWebsite"
 ==> "CopyArtifacts"
```

Figure D.1 – An example FAKE build pipeline

One of FAKE’s great strengths is that it just runs F# scripts – so you can *total* control; if you need to make your own custom build task, you can easily do so – just write some F#! In addition, you avoid tieing yourself into a particular CI server e.g. Team City / AppVeyor / VSTS etc. Instead, FAKE handles your build process, and because you can run it locally, it’s much easier to reproduce problems in your CI build chain; your CI server normally just has a single task in it, which is to run the FAKE script.

### **PROJECT SCAFFOLD**

<http://fsprojects.github.io/ProjectScaffold/>

The F# Project Scaffold was designed to create a framework projects to support many common features you’ll want in an open source project, such as automated build, dependency management, automatic NuGet package creation, HTML documentation generated from F# scripts and a full, one-click release process with automatic labelling and versioning to Git.

It’s commonly used for many open source F# projects (including many that you’ll see in this appendix), and the tools it uses are popular and well understood in the community, such as Fake, Paket and FSharp Formatting – but the build scripts can be complex, so there’s a lot to get your head around.

## D.1.2 Data

We’ve already seen how well suited F# is to working with data, but we’ve not covered all the libraries that are out there. This section covers a few more libraries that can make your life much easier when trying to perform more than the typical day-to-day sort of data operations we’ve seen so far.

### **EXCEL PROVIDER**

<http://fsprojects.github.io/ExcelProvider/>

As much as we might not like to admit it, Excel is everywhere and isn't going away any time soon. Sooner or later, you'll need to work with some data that a customer sends you in Excel.

The ExcelProvider is a type provider that sits on top of Excel files. In effect, you can work with Excel files directly in F#, just as you can use the FSharp.Data type provider to work seamlessly with CSV, JSON or XML files. There's a great deal of flexibility, and you can use cell ranges to only work with subsets of data within an entire sheet – great if there are multiple datasets within a single worksheet.

### **DEEDLE**

<http://bluemountaincapital.github.io/Deedle/>

Deedle is F#'s equivalent of R's DataFrames, or Python's Pandas. It allows you to work with data sets in a two-dimensional "frame" (think of old-school .NET DataTables) and perform operations on the frame, such as inferring missing cells based on surrounding rows, time-series analysis, groupings and aggregations. It's also used within the finance domain for things such as stock tickets and price analysis. The API definitely takes a bit of getting used to, but it's extremely powerful.

### **FSLAB**

<https://fslab.org/>

If you're interested in machine learning but thought that you'd need to leave the comfortable world of .NET to get involved, then this package is for you - F# Lab is a one-stop shop for machine learning on .NET. In and of itself, FS Lab doesn't do much – it's a NuGet package that simply references a set of other NuGet packages, including the usual data ones (FSharp.Data, Deedle etc.), charting libraries, machine learning libraries and the R Type Provider, which allows you to seamlessly call out to R packages and libraries from F#. It also contains some project templates that can make your life a little easier with regards to getting up-and-running, (although it's not strictly necessary to use them).

### **FSHARP.CHARTING**

<https://fslab.org/FSharp.Charting/>

Unlike XPlot (which uses Google Charts and Plotly to render visuals), FSharp.Charting uses the charting components built-in to .NET to create charts. Whilst the results are not quite as flexible as XPlot (nor as they quite as pretty), the API is very easy to use, and it has support for streaming and animated charts – something XPlot doesn't support.

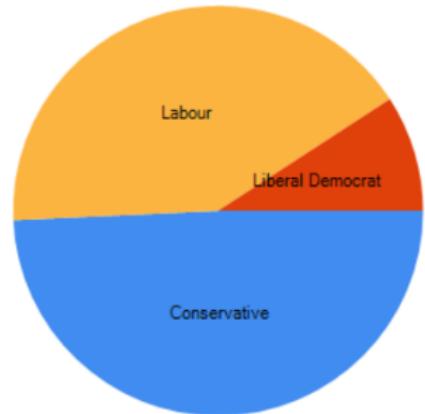
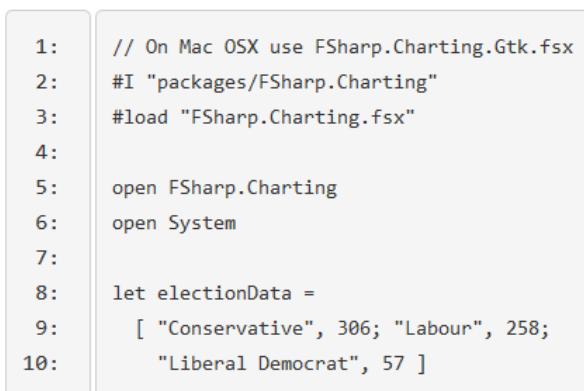


Figure D.2 – Using `FSharp.Charting` to quickly and easily generate visual charts

### D.1.3 Web

F# is close to achieving an exceptionally good web story. We've already seen how F# and functional patterns are a great fit for web on the server – but what about on the client? F# has two choices here, Fable and WebSharper, that allow you to do just that.

#### **FABLE**

<http://fable.io/>

Something that this book has steered away from is “F# on the client” – that is, running F# code *in the browser*. You can't directly do this, but what you can do with the Fable project is *transpile* F# into Javascript – in the same way that you can create Javascript from Typescript, or Dart or Coffeescript. The Fable project (and the FunScript project it replaced) do a very, very good job of doing this – not only if the generated Javascript easy-to-read, but it also maps across many calls from the BCL to Javascript libraries automatically.

There's a lot of really, really good work being done in this area, including libraries that allow you to design web pages in F# using custom DSLs; indeed, there are already developers that are reporting fantastic productivity gains by having F# on “both sides” of the fence. There's no reliance on specific Visual Studio tooling, and it works cross-platform – if you're doing any web programming, you really should check it out. Check out <http://fable.io/repl> for a page that converts F# to Javascript *in the browser* - seriously cool!

The screenshot shows a browser-based REPL interface for Fable. At the top, there are buttons for 'Samples:', 'ES2015' (selected), 'Compile', and 'Run'. The F# code in the left pane is:

```

1 let printName name =
2 let language =
3 match name with
4 | "Isaac" -> Some "F#"
5 | "Kelly" -> Some "C#"
6 | _ -> None
7
8 let message =
9 language
10 > Option.map(sprintf "%s likes working in %s" name)
11 > defaultArg & sprintf "I don't know what language %s uses!"
12
13 printfn "%O: %s" System.DateTime.UtcNow message
14
15 printName "Isaac"

```

The right pane shows the generated JavaScript code:

```

1 import { defaultArg } from "fable-core/Util";
2 import { fsFormat } from "fable-core/String";
3 import { utcNow } from "fable-core/Date";
4 export function printName(name) {
5 let language;
6
7 switch (name) {
8 case "Isaac":
9 language = "F#";
10 break;
11
12 case "Kelly":
13 language = "C#";
14 break;
15
16 default:
17 language = null;
18 }
19
20 const message = ($var2 => function (arg, defaultValue) {
21 | return arg != null ? arg : defaultValue;
22 })(defaultArg(language, null, $vari => fsFormat("%s likes working
23
24 fsFormat("%O: %s")(x => {
25 | console.log(x);
26 }))(utcNow(), message);
27 }
28 printName("Isaac");

```

Figure D.3 – Using Fable’s browser REPL to try out F# to Javascript compilation

## WEBSHARPER

<http://websharper.com/>

WebSharper is a more mature project than Fable that has the same aim – F# on the client – but tries to achieve this via a more prescriptive framework plus some Visual Studio templates. It’s extremely smart, using its own custom compiler to generate Javascript from F#, and allows you to write “reactive” (event-driven) applications entirely F#.

I should point out that WebSharper began as a commercial platform, as whilst it has become free to use now, it’s not quite the same in terms of open source as Fable. The flipside is that it has a team of developers that maintain the project, as well as consultants that offer training, so it’s not necessarily the case that you’ll be “on your own” if you use it. It’s not the easiest framework to pick up, and being a framework, it’s not only prescriptive but also locks you into a specific way of working.

## FREYA

<https://freya.io/>

Freya is another F#-first web programming framework, just like Suave, although Freya has a very different programming model. However, both support the idea of composing together small bits of functionality together to build larger systems. Freya is actively updated and uses several other libraries to provide very, very good performance (in fact, it sits on top of Kestrel, part of the new ASP .NET core framework).

## FSHARP.FORMATTING

<http://tpetricek.github.io/FSharp.Formatting/literate.html>

F# Formatting is a fantastic tool that allows you to generate HTML documentation based on either F# scripts, or a combination of markdown files with embedded F#. Indeed, the documentation sites that most F# open source projects is created using this. It also has support for embedding images, charts, tables and even source code (with tooltips!) inline: -

You can use F# Formatting for more than just documentation of a site – you can also use it for generating reports. For example, imagine you want to create a report on a daily basis that contains the latest sales figures – no problem. Simply create an F# script with the F# formatting library and generate your report as an HTML document. When you’re done generating it, email it off to key stakeholders, perhaps using a FAKE script to orchestrate the tasks!

```
Rapid navigation

You can easily move between containers, folders and blobs. Simpl
or folder will automatically request the children of that node f
easy exploration of your blob assets, directly from within the R
page and block blobs.
*)

(** define-output: blobStats **)
let container = Azure.Containers.samples
let theBlob = container."folder/".`"childFile.txt"
printfn "Blob '%s' is %d bytes big." theBlob.Name theBlob.Size
(** include-output: blobStats **)
```

**Rapid navigation**

You can easily move between containers, folders and blobs. Simply dotting into the children of that node from Azure. This allows easy exploration of your blob assets for both page and block blobs.

```
1: let container = Azure.Containers.samples
2: let theBlob = container.`"folder/`.`"childFile.txt`
3: printfn "Blob '%s' is %d bytes big." theBlob.Name theBlob.Size
```

Blob "folder/childFile.txt" is 16 bytes big.

Figure D.4 – Using FSharp.Formatting to generate a HTML content from combined Markdown and F#

### D.1.4 Cloud

Cloud applications is a subject that I spend a lot of time working with – and F# is very well placed to take advantage of the cloud (something Microsoft are well aware of). If writing multi-threaded applications is difficult to reason about, think about the difficulty of writing applications that run across multiple *machines* across an unreliable network – F#’s approach to immutable data and expressions really helps reason about things here!

## FSHARP AZURE STORAGE

<https://github.com/fsprojects/FSharp.Azure.Storage>

The Microsoft Azure storage service provides access to multiple services, one of which are Tables – a cheap, simple two-dimensional storage system. FSharp.Azure.Storage provides a pleasant F# DSL on top of the table service, with support for easy insertion, updates and queries of data directly from F# records.

## AZURE STORAGE TYPE PROVIDER

<http://fsprojects.github.io/AzureStorageTypeProvider/>

The Azure Storage Type Provider gives you a full type provider over the three main Azure Storage services – Blobs, Tables and Queues. Using the type provider, you can literally point it to a live Azure Storage account, and receive intellisense over the assets in the account. It supports “safe” querying over tables (in that it guarantees not to generate queries that the service will reject at runtime), and intelligently generates table types based on the contents of a remote table – there’s no need to create F# records in advance.

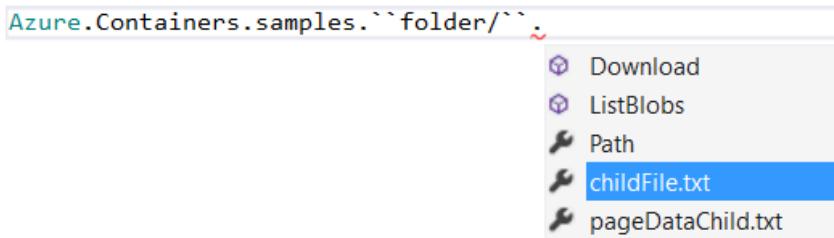


Figure D.5 – Navigating through a live Azure blob storage account in real-time with intellisense.

## FSHARP AWS DYNAMODB

<https://github.com/fsprojects/FSharp.AWS.DynamoDB>

If you’re an Amazon Web Services developer, rather than Microsoft Azure, there’s also the FSharp AWS DynamoDB library, which is similar to the FSharp Azure Storage library in that it’s not a type provider, but rather aims to provide a simple API that takes advantage of F#’s language features to make it easier to work with the DynamoDB storage system.

## MBRACE

<http://mbrace.io/>

MBrace is possibly the most exciting project of all here. It’s a general-purpose, flexible framework for distributed computing on .NET. It allows you to “wrap” arbitrary code blocks – be they accessing F# functions, values, or even VB or C# code – and distribute work across a cluster of machines, before returning back with the result. It’s extremely smart, handling distribution of captured values, exception propagation, parallel workloads – there’s even a “big data” library that support LINQ-style queries against massive data sets. The programming model is extremely easy to grok if you understand how `async { }` blocks work, as it’s effectively the same – you simply replace `async { }` with `cloud { }`! There’s support for both Azure and Amazon cloud systems as well as an on-premise model.

## D.1.5 Desktop

When we touched on WPF in this book, it was explicitly left as the domain of C# for the presentation layer, with F# providing the “business logic”. However, that’s not entirely necessary, as F# also has several excellent libraries for working with WPF.

### FsXAML

<http://fsprojects.github.io/FsXaml/>

FsXaml is a type provider that can create a strongly-typed view based on a XAML file. In effect, it replaces the need for the code-generation phase that is used in Visual Studio when working with XAML in C#. Using FsXaml, you can start to create complete WPF applications in F# projects, with full intellisense over controls that live in those views. If you already know WPF, this type provider will allow you to continue to work with WPF but in a 100% F# environment.

Whilst the official documentation is *very* sparse, there are some useful samples on the source code repository, as well as a collection of F# WPF templates that you can add to Visual Studio which show some larger examples of working with it.

### FSharp.ViewModule

<https://github.com/fsprojects/FSharp.ViewModule>

The companion project to FsXaml, FSharp.ViewModule provides a framework for creating GUI applications that adhere to the Model-View-ViewModel (MVVM) pattern that is popular in the WPF world. The MVVM pattern is typically very object-oriented, with changes to your state combined with the `INotifyPropertyChanged` interface to push changes to the view.

FSharp.ViewModule encapsulates much of this and allows you to focus on working with simple mutable objects with automatic change tracking. There’s also an excellent set of helpers for binding XAML Commands to standard F# functions; this abstraction means that you can write code that doesn’t stray too far from F# patterns and practices whilst still being able to work with WPF views.

## D.1.6 Miscellaneous

Alongside the above libraries, here are a few other miscellaneous libraries which, whilst not necessarily fitting into any specific category, are nonetheless useful and worth pointing out.

### Argu

<http://fsprojects.github.io/Argu/>

How many times have you written a console application and needed to parse some configuration arguments supplied to it? Argu is solves this problem for you. You simply define a declarative model that represents the possible arguments that can be supplied, and the tool

maps the model to the input arguments. There's support for mandatory and optional arguments, automatic help and friendly error messages, as well as automatic parsing of types.

### FSHARP.MANAGEMENT

<http://fsprojects.github.io/FSharp.Management/>

The FSharp.Management package contains a set of utility type providers that you'll always want to have around when working on Windows with local resources: -

- **File System** – strongly-typed access to files on the local file system.
- **Registry** – strongly-typed access to the Windows registry
- **WMI** - strongly-typed access to the Windows Management Instrumentation service
- **Powershell** – provides the ability to call Powershell functions and modules directly from F#
- **SystemTimeZonesProvider** – strongly-typed access to all time zones in .NET

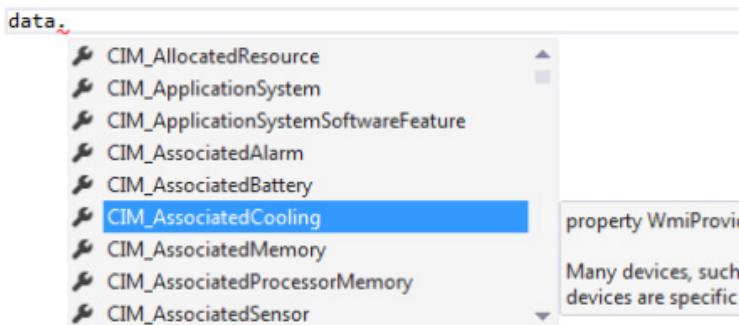


Figure D.6 – Accessing WMI components through intellisense via the WMI Type Provider

### FsREVEAL

<http://fsprojects.github.io/FsReveal/>

If you're fed up of doing presentations using Powerpoint, FSReveal is an easy-to-use tool that can generate web-ready slide decks based on simple markdown files – no need for a full application like powerpoint; you simply open a text editor (ideally one that can preview markdown) and off you go. Instead of heavyweight tooling, there's a simple FAKE script which converts the markdown files into a fully interactive website.

Since FSReveal sits on top of the JSReveal library to do most of the heavy lifting, it has all the nice features that JSReveal has such as speaker notes, support for clickers, animated images, tables etc. And since it also uses FSharp Formatting, you can embed code in your slide decks and have it automatically generated in a pleasant HTML web-based slide deck, complete with tooltips!

## FSHARP.CONFIGURATION

<http://fsprojects.github.io/FSharp.Configuration/>

FSharp.Configuration is a set of easy-to-use type providers that support the reading of various configuration file formats: -

- AppSettings – Application Settings (and connection strings) for a .NET config file
- ResX – .NET Resource files
- Yaml – YAML configuration files
- INI – Old-school .ini files

Most projects have some form of configuration settings, so you'll probably be using this package more often than not!

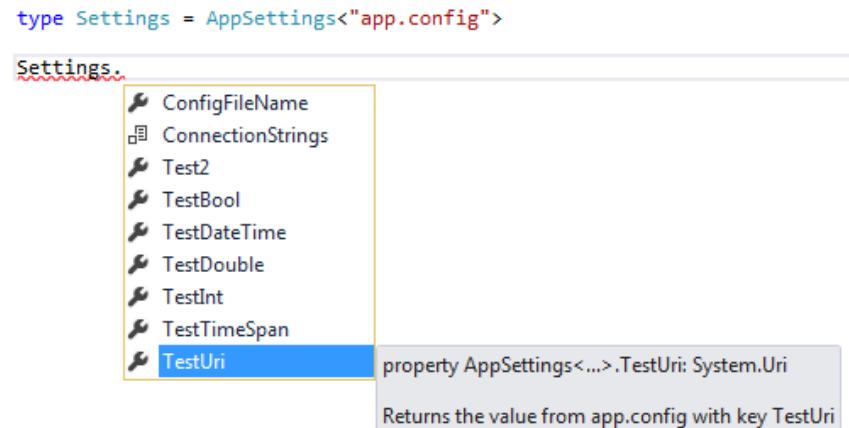


Figure D.7 – Accessing application settings from an app.config file using the Configuration Type Provider

## CHESSIE

<http://fsprojects.github.io/Chessie/>

The last project we'll address in this appendix is Chessie – a ready-made library for working with Result code (i.e. code that might be a Success or Failure), also known as "railway-oriented" programming. F#4.1 will actually have an in-built Result type, so I imagine that Chessie will change at some point to build on top of that, but for now it contains a complete Result type along with many useful helper methods, such as mapping, binding and so on – in effect, the same sort of behaviour that you can achieve with the Option module.

So, if you find that Option isn't quite enough for you, and you want to encode error details along the way rather than simply "throwing away" the None path, Chessie is a ready-made library for you (and one that has some very good documentation).

## D.2 The F# tool-chain

Having looked at all of these libraries, many of the points I made at the start of this appendix (and indeed at the very start of the book) should now become clear: -

- No reliance on custom tooling in Visual Studio
- Emphasising a code-first approach to development
- Type Providers instead of code generation
- Simple, independent libraries that can be composed together
- Open source, community-led projects

With that in mind, let's compare a typical tooling stack that you might be used to today, and an alternative stack that an F# developer *might* choose to adopt.

**Table D.1 – Comparing alternative technology stacks on .NET and F#**

Function	“Microsoft” stack	Pure F# Stack
Complex build process	MS Build Custom tasks	FAKE script
Continuous Integration	Team City / TFS pipeline etc.	FAKE script
Dependency Management	NuGet	Paket with NuGet + GitHub dependencies
Project system	Solution and Projects	Consider standalone scripts
Ad-hoc processing	Console Applications	Standalone scripts
Test Libraries	XUnit / NUnit	Expecto, QuickCheck, Unquote
SQL ORM	Entity Framework	SQLProvider
SQL micro-ORM	Dapper	FSharp.Data.SqlClient
Server-side web	Full-blown Web API project	Bare-bones NET Web API OWIN, or Suave
Front-end web	ASP .NET MVC / Typescript	F# with Fable
IDE	Visual Studio	VSCode / Emacs / Visual Studio etc. etc.

The main point here is that whilst you can *definitely* continue to work with virtually all of the technologies and tools in the “Microsoft” stack in the F# world, you’ll also find an “alternative” stack on the right-hand side, which more fully embraces the points previously made, with the overall aim of trying to improve productivity (and developer satisfaction!).

There’s nothing saying that you must pick everything on one of the two sides; the key take-away is to understand that there is more than one way to solving many of the problems where you may previously had assumed there was only a single option.

## D.3 Summary

This Appendix has hopefully given you a whirlwind tour around some of the tools and libraries that we've not introduced you to within the book, and shown you that there's a wide-ranging ecosystem in the F# world that takes full advantage of its features, in addition to the standard set of .NET libraries you'll already be familiar with. Doubtless there are other libraries which could just as easily have made this list, and you should investigate finding new libraries through the community on a regular basis.

# E

## Other F# Language Features

**Get Programming with F#** focused on a core subset of the F# language. As such, there are some features which were either only partially demonstrated, only references but not shown, or completely ignored. This appendix contains a list of many of those features (based on the current production version of F#, that is, F#4.0) for you to read up on in your own time - the best place to start is probably the F# language reference on MSDN (<https://docs.microsoft.com/en-gb/dotnet/articles/fsharp/language-reference/>).

### E.1 Object Oriented Support

This book deliberately steered clear of the rich OO-support in F#, as the aim of the book was not to simply show you how to write code that you write today but in another syntax, but to get you thinking about solving problems in a *different* way. Nonetheless, here's a very quick sample of typical OO features in F# - this should map relatively closely to C# / VB .NET OO features that you know, although there are a few differences that may surprise you!

#### E.1.1 Basic Classes

##### Listing E.1 – Basic classes in F#

```
type Person(age, firstname, surname) = #A
 let fullName = sprintf "%s %s" firstname surname #B

 member __.PrintFullName() = #C
 printfn "%s is %d years old" fullName age

 member this.Age = age
 member that.Name = fullName #D
 member val FavouriteColour = "Green" with get, set
```

#A Type definition with public constructor

#B Private field based on constructor args

```
#C Public method
#D Public getter-only property
#E Mutable, public property
```

Of interest in listing 5.1 is that we still can reap the benefits of type inference, whilst also taking advantage of the fact that member methods and properties can access constructor arguments without the need to set them as private backing fields first. Also of interest is that if a member does not need to access other members, you can omit the `this.` and replace it with simply `__` (as per the `PrintFullName` method).

## E.1.2 Interfaces and Inheritance

### Listing E.2 – Interfaces in F#

```
type IQuack =
 abstract member Quack : unit -> unit #A

type Duck (name:string) =
 interface IQuack with
 member this.Quack() = printfn "QUACK!" #B

let quacker =
 { new IQuack with
 member this.Quack() = printfn "What type of animal am I?" } #C

#A Defining an interface in F#
#B Creating a type that implements an interface
#C Creating an instance of an instance through an object expression
```

This should be pretty familiar to you, except for the last section, which shows how F# can create an *instance* of an interface without first formally defining a type. This is extremely useful if you're creating an abstract factory, as you don't need to formally define the types behind the interface first. Lastly – interfaces don't need to be explicitly marked as such; in F#, any type that contains only abstract members is automatically declared as an interface.

### Listing E.3 – Inheritance in F#

```
[<AbstractClass>] #A
type Employee(name:string) =
 member __.Name = name
 abstract member Work : unit -> string #B
 member this.DoWork() =
 printfn "%s is working hard: %s!" name (this.Work()) #C

type ProjectManager(name:string) =
 inherit Employee(name) #D
 override this.Work() = "Creating a project plan" #E
```

```
#A Creating an abstract class
#B Defining an abstract method
#C Calling an abstract method from a base class
#D Defining an inheritance hierarchy
```

## #E Overriding a virtual or abstract method

I'm not going to explain all the ins and outs of inheritance to you here – the sample above maps almost directly 1:1 with equivalent C# code in terms of concepts. The only real thing to note is that in F# you need to mark the type with the [`<AbstractClass>`] attribute.

## E.2 Exception Handling

You've seen exception handling occasionally in this book. Again, there's not much to say except that in F#, we tend to avoid exceptions as a way of messaging passing, and only really use them for truly exceptional cases. F# allows us to use pattern matching on the *type* of exception in order to create separate handlers; VB .NET has had this for some time, and C# recently added a similar feature.

### **Listing E.4 – Exception Handling in F#**

```
open System
let riskyCode() =
 raise(ApplicationException()) #A
()
let runSafely() =
 try
 riskyCode() #B
 with
 | :? ApplicationException as ex -> printfn "App exception! %0" ex #C
 | :? MissingFieldException as ex -> printfn "Missing field! %0" ex
 | ex -> printfn "Got some other type of exception! %0" ex
```

#A Throwing a specific exception using the `raise()` function

#B Placing code within a `try` block

#C Multiple catch handlers based on different Exception subtypes.

## E.3 Resource Management

The dispose pattern still exists within F#, and has language support, just like C#. Unlike C#, there are two ways to work with automatic disposal of objects: The first is the `use` keyword, whilst the second is the `using` block. Here's how they look: -

### **Listing E.5 – Exception Handling in F#**

```
let createDisposable() =
 printfn "Created!"
 { new IDisposable with member __.Dispose() = printfn "Disposed!" } #A

let foo() =
 use x = createDisposable() #B
 printfn "inside!"

let bar() =
 using (createDisposable()) (fun disposableObject -> #
 printfn "inside!")
```

```
#A A function which creates a disposable object
#B The use keyword with implicit disposal of resources
#C The using keyword with explicit disposal of resources
```

As you can see, the main difference between the two alternatives is that the former one lets the compiler determine when a disposable object goes out of scope, whereas the latter allows us to explicitly determine when scope ends, by virtue of a lambda function.

## E.4 Casting

F# has support for two types of casts – *upcast* and *downcast*. These are similar, but not quite the same as the `as` keyword and the cast functionality in C# or VB .NET: -

- Upcast (`:>`) The first one will safely upcast to a parent type in the type hierarchy – it will only allow you to do this for a type that is known to be “above” in the hierarchy.
- Downcast (`:?>`) will allow you to unsafely downcast from one type to another, but *only if the compiler knows that this is possible*. For example, you can’t downcast from `string` to `Exception` because the compiler knows that this will never pass. This is *not* the same as the cast functionality in C#, which will blindly allow casts that can be proved to be invalid even at compile-time.

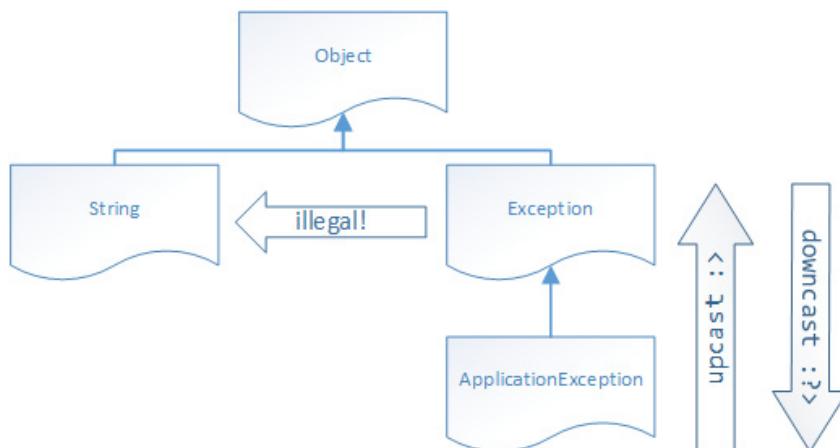


Figure E.1 – Upcast and Downcasts in F# are stricter than in C#

### Listing E.6 – Casting in F#

```
let anException = Exception()
let upcastToObject = anException :> obj #A
let upcastToAppException = anException :> ApplicationException #B
let downcastToAppException = anException :?> ApplicationException #C
let downcastToString = anException :?> string #D
```

```
#A Safely upcasting to Object
#B Trying to safely upcast to an incompatible type (error)
#C Unsaferly downcasting to an ApplicationException
#D Trying to unsafely downcasting to an incompatible type (error)
```

The first and third examples will compile – the first is guaranteed to work at compile- and at run-time, whilst the third compiles but might cause a run-time exception. The second and the fourth won't compile at all – the second fails because the `ApplicationException` is not above `Exception` in the type hierarchy. The last one fails because the compiler knows that `Exception` can never be treated as a string.

Don't forget that F# can safely pattern matching on types (as seen with exception handling) so you can safely try to cast across, and handle incompatibilities without the risk of run-time exceptions.

## E.5 Active Patterns

We briefly touched on Active Patterns in the book. They're a form of "lightweight" discriminated unions, and a way to *categorise* values in many different ways. For example, you could create an active pattern that categorised strings into long / medium / short and then pattern matched directly on string. There are also two more sophisticated forms of Active Patterns – *partial* active patterns and *parameterised* active patterns. Both allow you even more flexibility when pattern matching, but are beyond the scope of the book. I'd definitely recommend you look into these once you've mastered the basics of pattern matching and discriminated unions, because they allow for some really powerful abstractions.

## E.6 Computation Expressions

We mentioned these several times in the book; computation expressions allow us to create language support for a specific abstraction, directly in code – whether that's asynchronous work, optional objects, sequences or cloud computations. F# also allows you to create your own computation expressions to capture some type of behaviour that you want to abstract away, with your own versions of `let!` etc. Here's a quick example of a computation expression for working with Options.

### **Listing E.7 – A custom computation expression**

```
type Maybe() = #A
 member this.Bind(opt, func) = opt |> Option.bind func
 member this.Return v = Some v
let maybe = Maybe()

let rateCustomer name =
 match name with | "isaac" -> Some 3 | "mike" -> Some 2 | _ -> None

let answer =
 maybe { #B
 let! first = rateCustomer "isaac" #C}
```

```

let! second = rateCustomer "mike"
return first + second }

#A Creating our own computation expression
#B Creating a maybe {} block
#C Safely “unwrapping” an option type

```

Try this out in a script yourself. I don’t expect you to understand *all* of this code – but essentially the methods in the `Maybe` type map to calls in the `maybe{}` block (F# automatically maps `let!` to `Bind()` and `return` to `Return()`). Next, we have a fictional function that rates customers based on their name. Lastly, we call that function from within the `maybe` block. Notice that `rateCustomer` returns `Option<int>`, yet inside the block we don’t need to check for `Some` or `None` – the value is *safely* unwrapped by `let!`; if we try to get a customer that doesn’t exist, the entire block will prematurely return with `None`.

You can even create your own custom keywords in the language when inside the computation expression. They’re very powerful, and commonly found within custom DSLs – definitely a more advanced feature, but one that’s worth looking into.

## E.7 Code Quotations

Essentially the equivalent of C#’s Expression Trees, these allow you to wrap a block of code inside a `<@ quotation block @>` and then programmatically interrogate the abstract syntax tree (AST) within it. There are two forms of quotations in F# - *typed* and *untagged*. You’ll not find yourself using these in everyday code, but if you ever need to do some low-level meta programming, or write your own type provider, you’ll come into contact with these soon enough.

## E.8 Units of Measure

Units of Measure (UoM) is an *incredibly* useful feature of F#. The only reason why it wasn’t included in the book was because in my mind it’s not needed *very often*. UoM allows you to create a kind of “generic types” against numerics, so you can have `5<Kilogram>` as opposed to `5<Meter>`. You can also combine types, so you can model things such as `15<Meter/Second>` and so on. It’s extremely useful because the compiler will prevent you accidentally mixing and matching incompatible numeric types. UoM are also erased away at compile-time, so there’s no runtime overhead – they appear as regular floats or integers at runtime.

## E.9 Lazy Computations

Lazy computations in F# allow you to create `System.Lazy` values by wrapping any expression in a `lazy` scope.

```

let lazyText =
 lazy #A
 let x = 5 + 5

```

```

printfn "%0: Hello! Answer is %d" System.DateTime.UtcNow x
x

let text = lazyText.Value #B
let text2 = lazyText.Value #C

#A Creating a lazy scope
#B Explicitly evaluating the result of a lazy computation
#C Returning the result without re-executing the computation

```

In the example above, the code to print out the answer to the console will only occur the first time the `Value` property is accessed.

## E.10 Recursion

Lastly, as a functional-first language F# has excellent support for recursion, with special CLR support for *tail recursion* – the ability to call a recursive function indefinitely without error getting a stack overflow. To create a recursive function, you simply prefix it with the `rec` keyword.

```

let rec factorial number total = #A
 if number = 1 then total
 else
 printfn "Number %d" number
 factorial (number - 1) (total * number) #B
let total = factory 5 1 #C

#A Specifying that a function can be called recursively
#B Making a recursive function call
#C Calling a recursive function

```