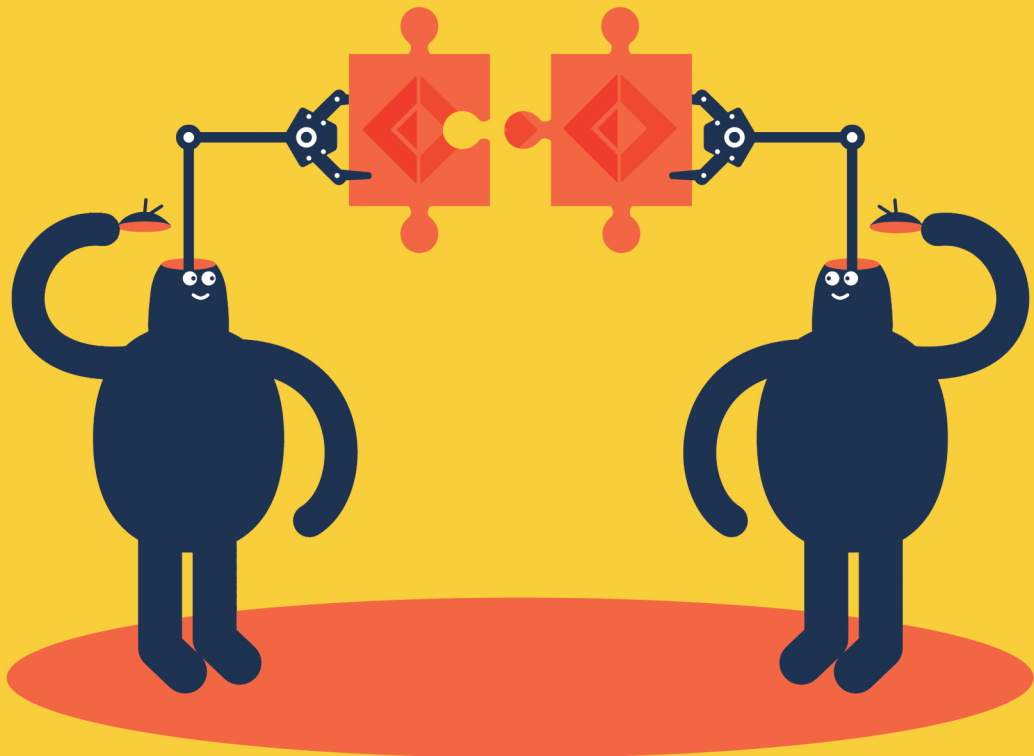


GET PROGRAMMING WITH F#

A guide for .NET developers



Isaac Abraham

Forewords
by Dustin Campbell
and Tomas Petricek

 MANNING

SAMPLE
CHAPTER



Get Programming with F#
by Isaac Abraham

Lesson 6

Copyright 2018 Manning Publications

Contents

Foreword v
Preface viii
Acknowledgments x
About this book xi
About the author xiv

Welcome to Get Programming
with F#! 1

Unit 1

F# AND VISUAL STUDIO

- Lesson 1** The Visual Studio experience 17
- Lesson 2** Creating your first F# program 25
- Lesson 3** The REPL—changing how we develop 34

Unit 2

HELLO F#

- Lesson 4** Saying a little, doing a lot 47
- Lesson 5** Trusting the compiler 58
- Lesson 6** Working with immutable data 70
- Lesson 7** Expressions and statements 81
- Lesson 8** Capstone 1 92

Unit 3

TYPES AND FUNCTIONS

- Lesson 9** Shaping data with tuples 101
- Lesson 10** Shaping data with records 111
- Lesson 11** Building composable functions 125
- Lesson 12** Organizing code without classes 138

- Lesson 13** Achieving code reuse in F# 149

- Lesson 14** Capstone 2 160

Unit 4

COLLECTIONS IN F#

- Lesson 15** Working with collections in F# 173
- Lesson 16** Useful collection functions 186
- Lesson 17** Maps, dictionaries, and sets 197
- Lesson 18** Folding your way to success 206
- Lesson 19** Capstone 3 219

Unit 5

THE PIT OF SUCCESS WITH THE F# TYPE SYSTEM

- Lesson 20** Program flow in F# 231
- Lesson 21** Modeling relationships in F# 244
- Lesson 22** Fixing the billion-dollar mistake 257
- Lesson 23** Business rules as code 270
- Lesson 24** Capstone 4 284

Unit 6

LIVING ON THE .NET PLATFORM

- Lesson 25** Consuming C# from F# 299
- Lesson 26** Working with NuGet packages 310

- Lesson 27** Exposing F# types and functions to C# 321
- Lesson 28** Architecting hybrid language applications 331
- Lesson 29** Capstone 5 342

Unit 7

WORKING WITH DATA

- Lesson 30** Introducing type providers 355
- Lesson 31** Building schemas from live data 365
- Lesson 32** Working with SQL 376
- Lesson 33** Creating type provider-backed APIs 388
- Lesson 34** Using type providers in the real world 401
- Lesson 35** Capstone 6 411

Unit 8

WEB PROGRAMMING

- Lesson 36** Asynchronous workflows 425

- Lesson 37** Exposing data over HTTP 439
- Lesson 38** Consuming HTTP data 453
- Lesson 39** Capstone 7 464

Unit 9

UNIT TESTING

- Lesson 40** Unit testing in F# 477
- Lesson 41** Property-based testing in F# 489
- Lesson 42** Web testing 501
- Lesson 43** Capstone 8 511

Unit 10

WHERE NEXT?

- Appendix A** The F# community 521
- Appendix B** F# in my organization 527
- Appendix C** Must-visit F# resources 537
- Appendix D** Must-have F# libraries 543
- Appendix E** Other F# language features 556

Index 564

6

LESSON

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, after 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 you'll see, such as expression-based development. In this lesson, you'll learn

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



6.1 Working with mutable data—a recap

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

6.1.1 The unrepeatable bug

Say you're developing an application, and one of the test team members comes up to you with a bug report. You walk over to that person's 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

How about this one? You're developing an application and have decided to use multithreading because it's cool. You recently heard about the Task Parallel Library in .NET, which makes writing multithreaded code a lot easier, and also saw that there's a `Parallel.ForEach()` method in the BCL. Great! You've also read about locking, so you carefully put locks around the bits of the shared state of your application that are affected by the multithreaded 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 multithreaded code. You don't know why it happened, though; it's caused by a race condition that occurs only under a specific load and a certain ordering of messages. Eventually, you revert your code to a single-threaded model.

6.1.3 Accidentally sharing state

Here's another one. You're working on a team and have designed a business object class. Your colleague has written code to operate on that object. You call that code, supplying an object, and then carry on. Sometime later, you notice a bug in your application: the state of the 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 realizing it. You made that property public only 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 that 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 first need to get the object into a specific state. 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 is to access a shared state that's 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.

6.1.5 Summary of mutability issues

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 we can't escape, and so look for other ways around these sorts of issues—things like encapsulation, hacks such as `InternalsVisibleTo`, or one of the many design patterns 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, you've only looked at simple values in F#, but even these show that by default, values are immutable. As you'll see in later lessons, this also applies to your own custom F# types (for example, Records).

6.2.1 Mutability basics in F#

You'll now see immutability in action. Start by opening a script file and entering the following code.

Listing 6.1 Creating immutable values in F#

```
let name = "isaac"      ← Creating an immutable value
name = "kate"          ← Trying to assign "kate" to name
```

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

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

The `false` doesn't mean that the assignment has somehow failed. It occurs because in F#, the `=` operator represents equality, as `==` does in C#. All you've done is compare `isaac` with `kate`, which is obviously false.

How do you update or mutate a value? You use the assignment operator, `<-`. Unfortunately, trying to insert that into your code leads to an error, as shown next.

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 you need to take one final step to make a value mutable, which is to use the `mutable` keyword.

Listing 6.3 Creating a mutable variable

```
let mutable name = "isaac"      ← Defining a mutable variable  
name <- "kate"                 ← 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 you use *variables* by default, and explicitly mark individual items as immutable *values* by using the `readonly` keyword.

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

As it turns out, you can easily develop entire applications (and I have, with web front ends, SQL databases, and so forth) by using only immutable data structures. You'll be surprised when you realize how little you need mutable data, particularly in request/response-style applications such as web applications, which are inherently stateless.

6.2.2 Working with mutable objects

Before we move on to 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

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()
form.Show()
form.Width <- 400
form.Height <- 400
form.Text <- "Hello from F#!"
```

← Creating the form object

← Mutating the form by using the <- operator

Mutable bindings and objects

Most objects in the BCL, such as a `Form`, are inherently mutable. Notice that the `form` symbol is immutable, so the binding symbol itself can't 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. 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, as shown in figure 6.1.

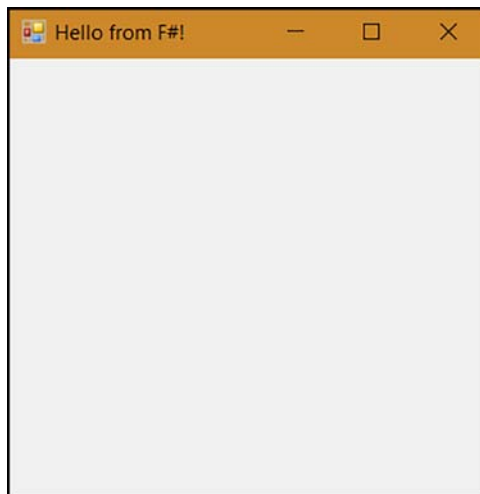



Figure 6.1 Creating a simple form from an F# script

F# also has a shortcut for creating mutable data structures in a way that assigns all properties in a single action. This shortcut is somewhat similar to object initializers in C#, except that in F# it works 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)
form.Show()
```

Creating and mutating properties
of a form in one expression



If actual constructor arguments are 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 6.1

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



6.3 Modeling 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 follows a simple model: you create an object, and then modify its state through operations on that object, as depicted in figure 6.2.

QC 6.1 answer

- 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.

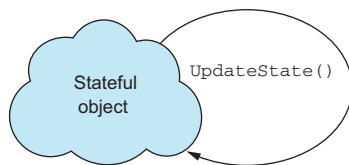


Figure 6.2 Mutating an object repeatedly

What's tricky about this model of working is that it can be hard to reason about your code. Calling a method such as `UpdateState()` in the preceding example 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 put this into practice with a simple example: driving a car. You want to be able to write code that allows you to `drive()` a car, tracking the amount of petrol (gas) used. Depending on the distance you drive, you should use up a different amount of petrol.

Listing 6.6 Managing state with mutable variables

```
let mutable petrol = 100.0           ← Initial state
let drive(distance) =               ← Modify state through mutation
    if distance = "far" then petrol <- petrol / 2.0
    elif distance = "medium" then petrol <- petrol - 10.0
    else petrol <- petrol - 1.0

drive("far")                         ← Repeatedly modify state
drive("medium")
drive("short")

petrol                                ← Check current state
```

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

- Calling `drive()` has no outputs. You call it, and it silently modifies the mutable `petrol` variable; you can't know this from the type system.
- Methods aren't deterministic. You can't know the behavior of a method without knowing the (often hidden) state. If you call `drive("far")` three times, the value of `petrol` will change every time, depending on the previous calls.
- You 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 compare working with mutable data structures with working with immutable ones, as per figure 6.3.

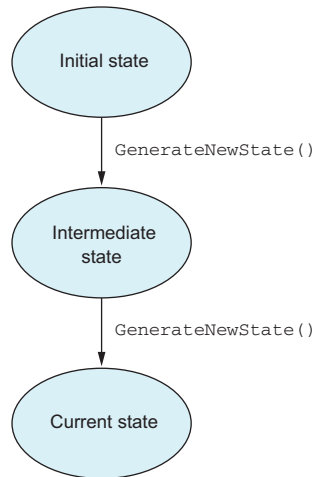


Figure 6.3 Generating new states working with immutable data

In this mode of operation, you can't mutate data. Instead, you create *copies* of the state with updates applied, and return that 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's slower to copy an object graph than to make an in-place update. But unless you're in a tight loop performing millions of mutations, the cost of doing so is negligible compared to, for example, opening a database connection. Plus, many languages (including F#) have specific data structures designed to work with immutable data in a highly efficient manner.

Now rewrite your code to use immutable data.

Listing 6.7 Managing state with immutable values

```

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

← Function explicitly dependent on state—takes in petrol and distance, and returns new petrol

```
else petrol - 1.0
let petrol = 100.0
let firstState = drive(petrol, "far")
let secondState = drive(firstState, "medium")
let finalState = drive(secondState, "short")
```

← Initial state

← Storing output state in a value

← Chaining calls together manually

There are now a few key changes to the code. The most important is that you aren't using a mutable variable for your state any longer, but a set of immutable values. You "thread" the state through each function call, storing the intermediate states in values that are then manually passed to the next function call. Working in this manner, you gain a few benefits immediately:

- You can reason about behavior much more easily. Rather than hidden side effects on private fields, each method or function call can return a new version of the state that you can easily understand. This makes unit testing much easier, for example.
- Function calls are repeatable. You can call `drive(50, "far")` as many times as you want, and it will always give you the same result. This is because 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 nice properties, such as being able to be cached or pregenerated, as well as being easier to test.
- The compiler is able to protect you in this case from accidentally misordering function calls, because each function call is explicitly dependent on the output of the previous call.
- You can also see the value of each intermediate step as you "work up" toward the final state.

Passing immutable state in F#

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

Now you try

Let's see how to make some changes to your drive code:

- 1 Instead of using a string to represent how far you've driven, use an integer.
- 2 Instead of `far`, check whether the distance is more than 50.
- 3 Instead of `medium`, check whether 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, return the same state that was provided.

6.3.3 Other benefits of immutable data

Immutable data has a few other benefits that aren't necessarily obvious from the preceding example:

- When working with immutable data, encapsulation isn't necessarily as important as it is when working with mutable data. At times encapsulation is still valuable (for example, as part of a public API), but on other occasions, making your data read-only eliminates the need to "hide" your data.
- You'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. Because 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 6.2

- 1 How do you handle changes in state when working with immutable data?
- 2 What is a pure function?
- 3 What impact does working with immutable data have with multithreading code?

QC 6.2 answer

- 1 By creating copies of existing data with applied changes.
- 2 A function that varies based only on the arguments explicitly passed to it.
- 3 Immutable data doesn't need to be locked when working across multiple threads.



Summary

In this lesson

- You learned about areas where mutable data structures can cause problems.
- You saw how immutable data can act as a form of state through copy-and-update that works particularly well with pure functions, while avoiding side effects to allow you to more easily reason about your code.
- You saw a simple example of how to create and work with immutable data in F#.

This is only the beginning, and you'll see examples throughout this book of how immutable data is a core part of F#. Also important is that F# encourages you to work with immutable data *by default*, but because 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. But just as 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 modeling another state machine with immutable data—for example, a kettle that can be filled with water, which is then poured into a teapot or directly into a cup.
- 2 Look at working with BCL classes that are inherently mutable, such as `System.Net.WebClient`. Explore various ways to create and modify them.

GET PROGRAMMING WITH F#

A guide for .NET developers | Isaac Abraham

Your .NET applications need to be good for the long haul. F#'s unique blend of functional and imperative programming is perfect for writing code that performs flawlessly now and keeps running as your needs grow and change. It takes a little practice to master F#'s functional-first style, so you may as well get programming!

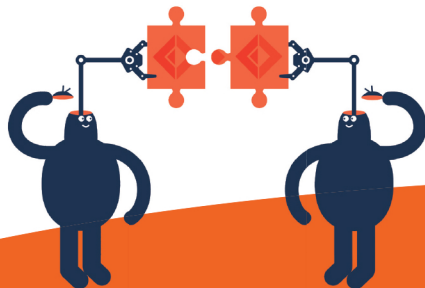
Get Programming with F#: A guide for .NET developers teaches F# through 43 example-based lessons with built-in exercises so you can learn the only way that really works: by practicing. The book upgrades your .NET skills with a touch of functional programming in F#. You'll pick up core FP principles and learn techniques for iron-clad reliability and crystal clarity. You'll discover productivity techniques for coding F# in Visual Studio, functional design, and integrating functional and OO code.

WHAT'S INSIDE

- Learn how to write bug-free programs
- Turn tedious common tasks into quick and easy ones
- Use minimal code to work with JSON, CSV, XML, and HTML data
- Integrate F# with your existing C# and VB.NET applications
- Create web-enabled applications

Written for intermediate C# and Visual Basic .NET developers. No experience with F# is assumed.

Isaac Abraham is an experienced .NET developer and trainer. He's an F# MVP for his contributions to the .NET community.



To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/get-programming-with-f-sharp

 MANNING

US \$44.99 | Can \$59.99 [Including eBook]

www.itbook.store/books/9781617293993

FREE EBOOK

See first page

“Leads you on a journey of F# that's both pragmatic and relevant.”

— FROM THE FOREWORD
BY DUSTIN CAMPBELL
MICROSOFT

“Does a great job of explaining F# clearly and effectively.”

— FROM THE FOREWORD
BY TOMAS PETRICEK
FSHARPWORKS

“A wonderful introduction to the subtleties of the F# language. You'll be productive within minutes!”

— JASON HALES
DIGITAL TIER

“Combines excellent explanations, real-world use cases, and a steady supply of questions and exercises to make sure you really understand what is being taught. Highly recommended!”

— JOEL CLERMONT
GROWTHPOINT

“Puts the FUN into functional programming with F#.”

— STEPHEN BYRNE
ACTION POINT

ISBN-13: 978-1-61729-399-3

ISBN-10: 1-61729-399-7



9 781617 293993