



Functional Programming in

How to write better C# code

Enrico Buonanno

 **MANNING**

SAMPLE CHAPTER



Functional Programming in C#

by Enrico Buonanno

Chapter 1

Copyright 2017 Manning Publications

brief contents

PART 1 CORE CONCEPTS1

- 1 ■ Introducing functional programming 3
- 2 ■ Why function purity matters 31
- 3 ■ Designing function signatures and types 52
- 4 ■ Patterns in functional programming 80
- 5 ■ Designing programs with function composition 102

PART 2 BECOMING FUNCTIONAL121

- 6 ■ Functional error handling 123
- 7 ■ Structuring an application with functions 149
- 8 ■ Working effectively with multi-argument functions 177
- 9 ■ Thinking about data functionally 202
- 10 ■ Event sourcing: a functional approach to persistence 229

PART 3 ADVANCED TECHNIQUES.....255

- 11 ■ Lazy computations, continuations, and the beauty of monadic composition 257
- 12 ■ Stateful programs and stateful computations 279

- 13 ■ Working with asynchronous computations 295
- 14 ■ Data streams and the Reactive Extensions 320
- 15 ■ An introduction to message-passing concurrency 345

Part 1

Core concepts

In this part we'll cover the basic techniques and principles of functional programming.

Chapter 1 starts by looking at what functional programming is, and how C# supports programming in a functional style. It then delves deeper into higher-order functions, a fundamental technique of FP.

Chapter 2 explains what pure functions are, why purity has important implications for a function's testability, and why pure functions lend themselves well to parallelization and other optimizations.

Chapter 3 deals with principles for designing types and function signatures—things you thought you knew but that receive a breath of fresh air when looked at from a functional perspective.

Chapter 4 introduces some of the core functions of FP: `Map`, `Bind`, `ForEach`, and `Where` (filter). These functions provide the basic tools for interacting with the most common data structures in FP.

Chapter 5 shows how functions can be chained into pipelines that capture the workflows of your program. It then widens the scope to developing a whole use case in a functional style.

By the end of part 1, you'll have a good feel for what a program written in a functional style looks like, and you'll understand the benefits that this style has to offer.

Introducing functional programming

This chapter covers

- Benefits and tenets of functional programming
- Functional features of the C# language
- Representation of functions in C#
- Higher-order functions

Functional programming is a programming *paradigm*: a different way of thinking about programs than the mainstream, imperative paradigm you're probably used to. For this reason, learning to think functionally is challenging but also very enriching. My ambition is that after reading this book, you'll never look at code with the same eyes as before!

The learning process can be a bumpy ride. You're likely to go from frustration at concepts that seem obscure or useless to exhilaration when something clicks in your mind, and you're able to replace a mess of imperative code with just a couple of lines of elegant, functional code.

This chapter will address some questions you may have as you start on this journey: What exactly is functional programming? Why should I care? Can I code functionally in C#? Is it worth the effort?

We'll start with a high-level overview of what functional programming (FP) is, and how well the C# language supports programming in a functional style. We'll then discuss functions and how they're represented in C#. Finally, we'll dip our feet in the water with higher-order functions, which I'll illustrate with a practical example.

1.1 *What is this thing called functional programming?*

What exactly is functional programming? At a very high level, it's a programming style that emphasizes functions while avoiding state mutation. This definition is already twofold, as it includes two fundamental concepts:

- Functions as first-class values
- Avoiding state mutation

Let's see what these mean.

1.1.1 *Functions as first-class values*

In a language where functions are first-class values, you can use them as inputs or outputs of other functions, you can assign them to variables, and you can store them in collections. In other words, you can do with functions all the operations that you can do with values of any other type.

For example, type the following into the REPL:¹

```
Func<int, int> triple = x => x * 3;
var range = Enumerable.Range(1, 3);
var triples = range.Select(triple);

triples // => [3, 6, 9]
```

In this example, you start by declaring a function that returns the triple of a given integer and assigning it to the variable `triple`. You then use `Range` to create an `IEnumerable<int>` with the values `[1, 2, 3]`. You then invoke `Select` (an extension method on `IEnumerable`), giving it the range and the `triple` function as arguments; this creates a new `IEnumerable` containing the elements obtained by applying the `triple` function to each element in the input range.

This short snippet demonstrates that functions are indeed first-class values in C#, because you can assign the multiply-by-3 function to the variable `triple`, and give it as an argument to `Select`. Throughout the book you'll see that treating functions as values allows you to write some very powerful and concise code.

¹ A REPL is a command-line interface allowing you to experiment with the language by typing in statements and getting immediate feedback. If you use Visual Studio, you can start the REPL by going to `View > Other Windows > C# Interactive`. On Mono, you can use the `csharp` command. There are also several other utilities that allow you to run C# snippets interactively, some even in the browser.

1.1.2 Avoiding state mutation

If we follow the functional paradigm, we should refrain from state mutation altogether: once created, an object *never* changes, and variables should never be reassigned. The term *mutation* indicates that a value is changed in-place—updating a value stored somewhere in memory. For example, the following code creates and populates an array, and then it updates one of the array’s values in place:

```
int[] nums = { 1, 2, 3 };
nums[0] = 7;

nums // => [7, 2, 3]
```

Such updates are also called *destructive* updates, because the value stored prior to the update is destroyed. These should always be avoided when coding functionally. (Purely functional languages don’t allow in-place updates at all.)

Following this principle, sorting or filtering a list should not modify the list in place but should create a new, suitably filtered or sorted list without affecting the original. Type the following into the REPL to see what happens when sorting or filtering a list using LINQ’s `Where` and `OrderBy` functions.

Listing 1.1 Functional approach: `Where` and `OrderBy` don’t affect the original list

```
Func<int, bool> isOdd = x => x % 2 == 1;
int[] original = { 7, 6, 1 };

var sorted = original.OrderBy(x => x);
var filtered = original.Where(isOdd);

original // => [7, 6, 1]
sorted   // => [1, 6, 7]
filtered // => [7, 1]
```

The original list hasn’t been affected.

Sorting and filtering yielded new lists.

As you can see, the original list is unaffected by the sorting or filtering operations, which yielded new `IEnumerables`.


Let’s look at a counterexample. If you have a `List<T>`, you can sort it in place by calling its `Sort` method.

Listing 1.2 Nonfunctional approach: `List<T>.Sort` sorts the list in place

```
var original = new List<int> { 5, 7, 1 };
original.Sort();

original // => [1, 5, 7]
```

In this case, after sorting, the original ordering is destroyed. You’ll see why this is problematic right away.

 **NOTE** The reason you see both the functional and nonfunctional approaches in the framework is historical: `List<T>.Sort` predates LINQ, which marked a decisive turn in a functional direction.

1.1.3 *Writing programs with strong guarantees*

Of the two concepts we just discussed, functions as first-class values initially seems more exciting, and we'll concentrate on it in the latter part of this chapter. But before we move on, I'd like to briefly demonstrate why avoiding state mutation is also hugely beneficial, as it eliminates many complexities caused by mutable state.

Let's look at an example. (We'll revisit these topics in more detail, so don't worry if not everything is clear at this point.) Type the following code into the REPL.

Listing 1.3 Mutating state from concurrent processes yields unpredictable results

```
using static System.Linq.Enumerable;
using static System.Console;

var nums = Range(-10000, 20001).Reverse().ToList();
// => [10000, 9999, ... , -9999, -10000]

Action task1 = () => WriteLine(nums.Sum());
Action task2 = () => { nums.Sort(); WriteLine(nums.Sum()); };

Parallel.Invoke(task1, task2);
// prints: 92332970
//           0
```

This allows you to call `Range` and `WriteLine` without full qualification.

Executes both tasks in parallel

Here you define `nums` to be a list of all integers between 10,000 and -10,000; their sum should obviously be 0. You then create two tasks: `task1` computes and prints out the sum; `task2` first sorts the list and then computes and prints the sum. Each of these tasks will correctly compute the sum if run independently. When you run both tasks in parallel, however, `task1` comes up with an incorrect and unpredictable result.

It's easy to see why: as `task1` reads the numbers in the list to compute the sum, `task2` is reordering that very same list. That's somewhat like trying to read a book while somebody else flips the pages: you'd be reading some well-mangled sentences! Graphically, this can be illustrated as shown in figure 1.1.

What if we use LINQ's `OrderBy` method, instead of sorting the list in place?

```
Action task3 = () => WriteLine(nums.OrderBy(x => x).Sum());

Parallel.Invoke(task1, task3);
// prints: 0
//           0
```

As you can see, using LINQ's functional implementation gives you a predictable result, even when you execute the tasks in parallel. This is because `task3` isn't modifying the

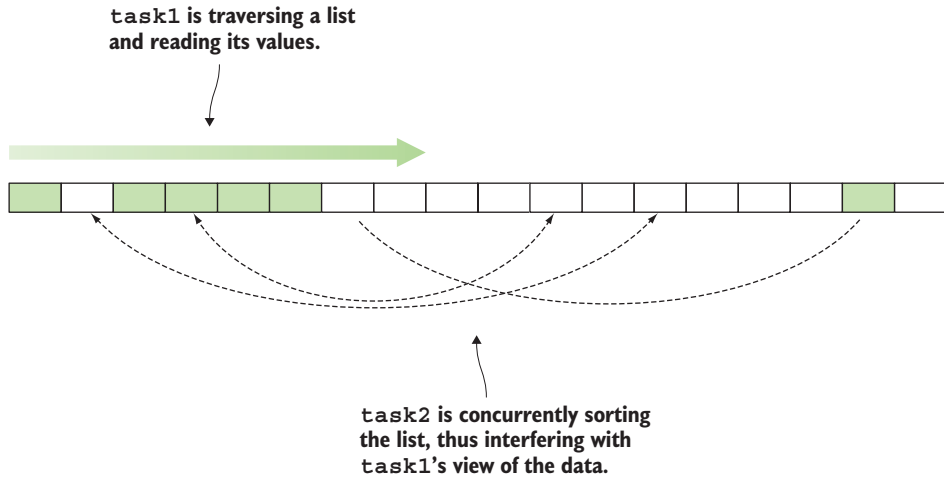


Figure 1.1 Modifying data in place can give concurrent threads an incorrect view of the data

original list but rather creating a completely new “view” of the data, which is sorted—task1 and task3 read from the original list concurrently, but concurrent reads don't cause any inconsistencies, as shown in figure 1.2.

This simple example illustrates a wider truth: when developers write an application in the imperative style (explicitly mutating the program state) and later introduce

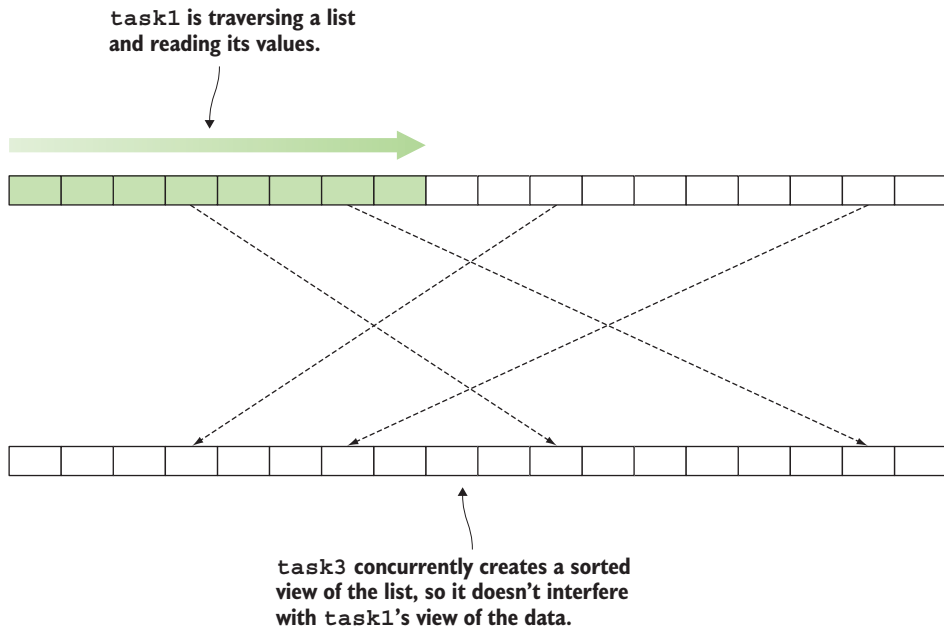


Figure 1.2 The functional approach: creating a new, modified version of the original structure

concurrency (due to new requirements, or a need to improve performance), they inevitably face a lot of work and potentially some difficult bugs. When a program is written in a functional style from the outset, concurrency can often be added for free, or with substantially less effort. We'll discuss state mutation and concurrency more in chapters 2 and 9. For now, let's go back to our overview of FP.

Although most people will agree that treating functions as first-class values and avoiding state mutation are fundamental tenets of FP, their application gives rise to a series of practices and techniques, so it's debatable which techniques should be considered essential and included in a book like this.

I encourage you to take a pragmatic approach to the subject and try to understand FP as *a set of tools* that you can use to address your programming tasks. As you learn these techniques, you'll start to look at problems from a different perspective: you'll start to think functionally.

Now that we have a working definition of FP, let's look at the C# language itself, and at its support for FP techniques.

Functional vs. object-oriented?

I'm often asked to compare and contrast FP with object-oriented programming (OOP). This isn't simple, mainly because there are many incorrect assumptions about what OOP should look like.

In theory, the fundamental principles of OOP (encapsulation, data abstraction, and so on) are orthogonal to the principles of FP, so there's no reason why the two paradigms can't be combined.

In practice, however, most object-oriented (OO) developers heavily rely on the *imperative* style in their method implementations, mutating state in place and using explicit control flow: they use OO design in the large, and imperative programming in the small. So the real question is that of *imperative vs. functional* programming, and I'll summarize the benefits of FP at the end of this chapter.

Another question that often arises is how FP differs from OOP in terms of structuring a large, complex application. The difficult art of structuring a complex application relies on several principles:

- Modularity (dividing software into reusable components)
- Separation of concerns (each component should only do one thing)
- Layering (high-level components can depend on low-level components, but not vice versa)
- Loose coupling (changes to a component shouldn't affect components that depend on it)

These principles are generally valid, regardless of whether the component in question is a function, a class, or an application.

(continued)

They're also in no way specific to OOP, so the same principles can be used to structure an application written in the functional style—the difference will be in what the components are, and what APIs they expose.

In practice, the functional emphasis on pure functions (which we'll discuss in chapter 2) and composability (chapter 5) make it significantly easier to achieve some of these design goals.²

1.2 How functional a language is C#?

Functions are indeed first-class values in C#, as demonstrated in the previous listings. In fact, C# had support for functions as first-class values from the earliest version of the language through the `Delegate` type, and the subsequent introduction of *lambda expressions* made the syntactic support even better—we'll review these language features in the next section.

There are some quirks and limitations, such as when it comes to type inference; we'll discuss these in chapter 8. But overall, the support for functions as first-class values is pretty good.

As for supporting a programming model that avoids in-place updates, the fundamental requirement in this area is that a language have garbage collection. Because you create modified versions, rather than updating existing values in place, you want old versions to be garbage collected as needed. Again, C# satisfies this requirement.

Ideally, the language should also *discourage* in-place updates. This is C#'s greatest shortcoming: everything is mutable by default, and the programmer has to put in a substantial amount of effort to achieve immutability. Fields and variables must explicitly be marked `readonly` to prevent mutation. (Compare this to F#, where variables are immutable by default and must explicitly be marked `mutable` to allow mutation.)

What about types? There are a few immutable types in the framework, such as `string` and `DateTime`, but language support for user-defined immutable types is poor (although, as you'll see next, it has improved in C# 6 and is likely to improve further in future versions). Finally, collections in the framework are mutable, but a solid library of immutable collections is available.

In summary, C# has very good support for some functional techniques, but not others. In its evolution, it has improved, and it will continue to improve its support for functional techniques. In this book, you'll learn which features can be harnessed, and also how to work around its shortcomings.

Next we'll review some language features from past, present, and upcoming versions of C# that are particularly relevant to FP.

² For a more thorough discussion on why imperatively flavored OOP is a *cause of*, rather than a solution to, program complexity, see *Out of the Tar Pit* by Ben Moseley and Peter Marks, 2006 (<https://github.com/papers-we-love/papers-we-love/raw/master/design/out-of-the-tar-pit.pdf>).

1.2.1 *The functional nature of LINQ*

When C# 3 was released, along with version 3.5 of the .NET Framework, it included a host of features inspired by functional languages, including the LINQ library (`System.Linq`) and some new language features enabling or enhancing what you could do with LINQ, such as extension methods and expression trees.

LINQ is indeed a functional library—as you probably noticed, I used LINQ earlier to illustrate both tenets of FP—and the functional nature of LINQ will become even more apparent as you progress through this book.

LINQ offers implementations for many common operations on lists (or, more generally, on “sequences,” as instances of `IEnumerable` should technically be called), the most common of which are mapping, sorting, and filtering (see the “Common operations on sequences” sidebar). Here’s an example combining all three:

```
Enumerable.Range(1, 100)
    .Where(i => i % 20 == 0)
    .OrderBy(i => -i)
    .Select(i => $"{i}%")
// => ["100%", "80%", "60%", "40%", "20%"]
```

Notice how `Where`, `OrderBy`, and `Select` all take functions as arguments and don’t mutate the given `IEnumerable`, but return a new `IEnumerable` instead, illustrating both tenets of FP you saw earlier.

LINQ facilitates querying not only objects in memory (LINQ to Objects), but various other data sources, like SQL tables and XML data. C# programmers have embraced LINQ as the standard toolset for working with lists and relational data (accounting for a substantial amount of a typical codebase). On the up side, this means that you’ll already have some sense of what a functional library’s API feels like.

On the other hand, when working with other types, C# programmers generally stick to the imperative style of using flow-control statements to express the program’s intended behavior. As a result, most C# codebases I’ve seen are a patchwork of functional style (when working with `IEnumerables` and `IQueryable`s) and imperative style (everything else).

What this means is that although C# programmers are aware of the benefits of using a functional library such as LINQ, they haven’t had enough exposure to the design principles behind LINQ to leverage those techniques in their own designs. That’s something this book aims to address.

Common operations on sequences

The LINQ library contains many methods for performing common operations on sequences, such as the following:

- *Mapping*—Given a sequence and a function, mapping yields a new sequence with the elements obtained by applying the given function to each element in the given sequence (in LINQ, this is done with the `Select` method).

```
Enumerable.Range(1, 3).Select(i => i * 3) // => [3, 6, 9]
```

(continued)

- **Filtering**—Given a sequence and a predicate, filtering yields a new sequence consisting of the elements from the given sequence that pass the predicate (in LINQ, `Where`).


```
Enumerable.Range(1, 10).Where(i => i % 3 == 0) // => [3, 6, 9]
```

- **Sorting**—Given a sequence and a key-selector function, sorting yields a new sequence ordered according to the key (in LINQ, `OrderBy` and `OrderByDescending`).

```
Enumerable.Range(1, 5).OrderBy(i => -i) // => [5, 4, 3, 2, 1]
```

1.2.2 Functional features in C# 6 and C# 7

C# 6 and C# 7 aren't as revolutionary as C# 3, but they include many smaller language features that, taken together, provide a much better experience and more idiomatic syntax for coding functionally.

 **NOTE** Most features introduced in C# 6 and C# 7 offer better syntax, not new functionality. If you're using an older version of C#, you can still apply all of the techniques shown in this book (with a bit of extra typing). However, these newer features significantly improve readability, making programming in a functional style more attractive.

You can see these features in action in the following listing.

Listing 1.4 C# 6 and C# 7 features relevant for FP

```
using static System.Math;
public class Circle
{
    public Circle(double radius)
        => Radius = radius;
    public double Radius { get; }
    public double Circumference
        => PI * 2 * Radius;
    public double Area
    {
        get
        {
            double Square(double d) => Pow(d, 2);
            return PI * Square(Radius);
        }
    }
    public (double Circumference, double Area) Stats
        => (Circumference, Area);
}
```

← using static enables unqualified access to the static members of System.Math, like PI and Pow.

← A getter-only auto-property can be set only in the constructor.

← An expression-bodied property

← A local function is a method declared within another method.

← C# 7 tuple syntax with named elements

IMPORTING STATIC MEMBERS WITH USING STATIC

The `using static` statement in C# 6 allows you to import the static members of a class (in this example, the `System.Math` class). As a result, in this example you can invoke the `PI` and `Pow` members of `Math` without further qualification:

```
using static System.Math;

public double Circumference
    => PI * 2 * Radius;
```

Why is this important? In FP, we prefer functions whose behavior relies only on their input arguments because we can reason about and test these functions in isolation (contrast this with instance methods, whose implementation typically interacts with instance variables). These functions are implemented as static methods in C#, so a functional library in C# will consist mainly of static methods.

`using static` allows you to more easily consume such libraries, and although overuse can lead to namespace pollution, reasonable use can make for clean, readable code.

EASIER IMMUTABLE TYPES WITH GETTER-ONLY AUTO-PROPERTIES

When you declare a getter-only auto-property, such as `Radius`, the compiler implicitly declares a `readonly` backing field. As a result, these properties can only be assigned a value in the constructor or inline:

```
public class Circle
{
    public Circle(double radius)
        => Radius = radius;

    public double Radius { get; }
}
```

Getter-only auto-properties facilitate the definition of immutable types, which you'll see in more detail in chapter 9. The `Circle` class demonstrates this: it only has one field (the backing field of `Radius`), which is `readonly`, so once it's created, a `Circle` can never change.

MORE CONCISE FUNCTIONS WITH EXPRESSION-BODIED MEMBERS

The `Circumference` property is declared with an *expression body* introduced with `=>`, rather than with the usual *statement body* in `{ }`:

```
public double Circumference
    => PI * 2 * Radius;
```

Notice how much more concise this is compared to the `Area` property!

In FP, we tend to write lots of simple functions, many of them one-liners, and then compose them into more complex workflows. Expression-bodied methods allow you to do this with minimal syntactic noise. This is particularly evident when you want to write a function that returns a function—something you'll do a lot in this book.

The expression-bodied syntax was introduced in C# 6 for methods and properties, and it was generalized in C# 7 to also apply to constructors, destructors, getters, and setters.

LOCAL FUNCTIONS

Writing lots of simple functions means that many functions are called from one location only. C# 7 allows you to make this explicit by declaring methods within the scope of a method; for instance, the `Square` method is declared within the scope of the `Area` getter:

```
get
{
    double Square(double d) => Pow(d, 2);
    return PI * Square(Radius);
}
```

BETTER SYNTAX FOR TUPLES

Better syntax for tuples is the most important feature of C# 7. It allows you to easily create and consume tuples, and, most importantly, to assign meaningful names to their elements. For example, the `Stats` property returns a tuple of type `(double, double)`, and specifies meaningful names by which its elements can be accessed:

```
public (double Circumference, double Area) Stats
    => (Circumference, Area);
```

Tuples are important in FP because of the tendency to break tasks down into very small functions. You may end up with a data type whose only purpose is to capture the information returned by one function, and that's expected as input by another function. It's impractical to define dedicated types for such structures, which don't correspond to meaningful domain abstractions. That's where tuples come in.

1.2.3 A more functional future for C#?

As I was writing the first draft of this chapter, in early 2016, development of C# 7 was in its early days, and it was interesting to see that *all* the features for which the language team had identified “strong interest” were features normally associated with functional languages. They included the following:

- Record types (boilerplate-free immutable types)
- Algebraic data types (a powerful addition to the type system)
- Pattern matching (similar to a `switch` statement that works on the *shape* of the data, such as its type, rather than just the values)
- Better syntax for tuples

On one hand, it was disappointing that only the last item could be delivered. C# 7 also includes a limited implementation of pattern matching, but it's a far cry from the kind of pattern matching available in functional languages, and it's generally inadequate for the way we'd like to use pattern matching when programming functionally (see section 10.2.4).

On the other hand, these features are still on the table for future versions, and work has been done on the respective proposals. This means we're likely to see record types and a more complete implementation of pattern matching in future versions of C#. So C# is poised to continue in its evolution as a multi-paradigm language with an increasingly strong functional component.

This book will give you a good foundation for keeping up with the evolution of the language and the industry. It'll also give you a good understanding of the concepts and motivations behind future versions of the language.

1.3 *Thinking in functions*

In this section, I'll clarify what I mean by *function*. I'll start with the mathematical use of the word and then move on to the various language constructs that C# offers to represent functions.

1.3.1 *Functions as maps*

In mathematics, a function is a map between two sets, respectively called the *domain* and *codomain*. That is, given an element from its domain, a function yields an element from its codomain. That's all there is—it doesn't matter whether the mapping is based on some formula or is completely arbitrary.

In this sense, a function is a completely abstract mathematical object, and the value that a function yields is determined *exclusively* by its input. You'll see that this isn't always the case with functions in programming.

For example, imagine a function mapping lowercase letters to their uppercase counterparts, as in figure 1.3. In this case, the domain is the set $\{a, b, c, \dots\}$ and the codomain is the set $\{A, B, C, \dots\}$. (Naturally, there are functions for which the domain and codomain are the same set; can you think of an example?)

How does this relate to programming functions? In statically typed languages like C#, the sets (domain and codomain) are represented with types. For example, if you coded the function above, you could use `char` to represent both the domain and the codomain. The type of your function could then be written as

```
char → char
```

That is, the function maps chars to chars, or, equivalently, given a `char`, it yields a `char`.

The types for the domain and codomain constitute a function's interface, also called its type, or signature. You can think of this as a contract: a function signature declares that, given an element from the domain, it will yield an element from the

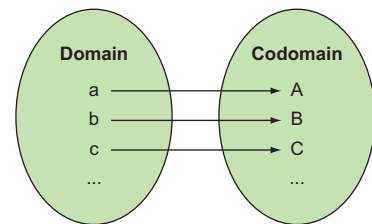


Figure 1.3 A mathematical function is a mapping between the elements of two sets.

codomain.³ This may sound pretty obvious, but you'll see in chapter 3 that in reality, violations of the signature contract abound.

Next, let's look at ways to encode the functions themselves.

1.3.2 Representing functions in C#

There are several language constructs in C# that you can use to represent functions:

- Methods
- Delegates
- Lambda expressions
- Dictionaries

If you're well-versed in these, skip to the next section; otherwise, here's a quick refresher.

METHODS

Methods are the most common and idiomatic representation for functions in C#. For example, the `System.Math` class includes methods representing many common mathematical functions. Methods can represent functions, but they also fit into the object-oriented paradigm—they can be used to implement interfaces, they can be overloaded, and so on.

The constructs that really enable you to program in a functional style are delegates and lambda expressions.

DELEGATES

Delegates are type-safe function pointers. *Type-safe* here means that a delegate is strongly typed: the types of the input and output values of the function are known at compile time, and consistency is enforced by the compiler.

Creating a delegate is a two-step process: you first declare the delegate type and then provide an implementation. (This is analogous to writing an interface and then instantiating a class implementing that interface.)

The first step is done by using the `delegate` keyword and providing the signature for the delegate. For example, .NET includes the following definition of a `Comparison<T>` delegate.

Listing 1.5 Declaring a delegate

```
namespace System
{
    public delegate int Comparison<in T>(T x, T y);
}
```

As you can see, a `Comparison<T>` delegate can be given two `T`'s and will yield an `int` indicating which is greater.

³ Interfaces in the OO sense are an extension of this idea: a set of functions with their respective input and output types, or, more precisely, *methods*, which are essentially functions, that take `this`, the current instance, as an implicit argument.

Once you have a delegate type, you can instantiate it by providing an implementation, like this.

Listing 1.6 Instantiating and using a delegate

```
var list = Enumerable.Range(1, 10).Select(i => i * 3).ToList();
list // => [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

Comparison<int> alphabetically = (l, r)
    => l.ToString().CompareTo(r.ToString());
```

Provides an implementation
of Comparison

```
list.Sort(alphabetically);
list // => [12, 15, 18, 21, 24, 27, 3, 30, 6, 9]
```

← Uses the Comparison delegate
as an argument to Sort

As you can see, a delegate is just an *object* (in the technical sense) that represents an operation—in this case, a comparison. Just like any other object, you can use a delegate as an argument for another method, as in listing 1.6, so delegates are the language feature that makes functions first-class values in C#.

THE FUNC AND ACTION DELEGATES

The .NET framework includes a couple of delegate “families” that can represent pretty much any function type:

- `Func<R>` represents a function that takes no arguments and returns a result of type `R`.
- `Func<T1, R>` represents a function that takes an argument of type `T1` and returns a result of type `R`.
- `Func<T1, T2, R>` represents a function that takes a `T1` and a `T2` and returns an `R`.

And so on. There are delegates to represent functions of various “arities” (see the “Function arity” sidebar).

Since the introduction of `Func`, it has become rare to use custom delegates. For example, instead of declaring a custom delegate like this,

```
delegate Greeting Greeter(Person p);
```

you can just use the type:

```
Func<Person, Greeting>
```

The type of `Greeter` in the preceding example is equivalent to, or “compatible with,” `Func<Person, Greeting>`. In both cases it’s a function that takes a `Person` and returns a `Greeting`.

There’s a similar delegate family to represent *actions*—functions that have no return value, such as void methods:

- `Action` represents an action with no input arguments.
- `Action<T1>` represents an action with an input argument of type `T1`.
- `Action<T1, T2>` and so on represent an action with several input arguments.

The evolution of .NET has been *away* from custom delegates, in favor of the more general `Func` and `Action` delegates. For instance, take the representation of a *predicate*:⁴

- In .NET 2, a `Predicate<T>` delegate was introduced, which is used, for instance, in the `FindAll` method used to filter a `List<T>`.
- In .NET 3, the `Where` method, also used for filtering but defined on the more general `IEnumerable<T>`, takes not a `Predicate<T>` but simply a `Func<T, bool>`.

Both function types are equivalent. Using `Func` is recommended to avoid a proliferation of delegate types that represent the same function signature, but there's still something to be said in favor of the expressiveness of custom delegates: `Predicate<T>`, in my view, conveys intent more clearly than `Func<T, bool>` and is closer to the spoken language.

Function arity

Arity is a funny word that refers to the number of arguments that a function accepts:

- A *nullary* function takes no arguments.
- A *unary* function takes one argument.
- A *binary* function takes two arguments.
- A *ternary* function takes three arguments.

And so on. In reality, all functions can be viewed as being unary, because passing n arguments is equivalent to passing an n -tuple as the only argument. For example, addition (like any other binary arithmetic operation) is a function whose domain is the set of all *pairs* of numbers.

LAMBDA EXPRESSIONS

Lambda expressions, called *lambdas* for short, are used to declare a function inline. For example, sorting a list of numbers alphabetically can be done with a lambda like so.

Listing 1.7 Declaring a function inline with a lambda

```
var list = Enumerable.Range(1, 10).Select(i => i * 3).ToList();
list // => [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

list.Sort((l, r) => l.ToString().CompareTo(r.ToString()));
list // => [12, 15, 18, 21, 24, 27, 3, 30, 6, 9]
```

If your function is short and you don't need to reuse it elsewhere, lambdas offer the most attractive notation. Also notice that in the preceding example, the compiler not only infers the types of x and y to be `int`, it also converts the lambda to the delegate type `Comparison<int>` expected by the `Sort` method, given that the provided lambda is compatible with this type.

⁴ A predicate is a function that, given a value (say, an integer), tells you whether it satisfies some condition (say, whether it's even).

Just like methods, delegates and lambdas have access to the variables in the scope in which they're declared. This is particularly useful when leveraging *closures* in lambda expressions.⁵ Here's an example.

Listing 1.8 Lambdas have access to variables in the enclosing scope

```
var days = Enum.GetValues(typeof(DayOfWeek)).Cast<DayOfWeek>();
// => [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]

IEnumerable<DayOfWeek> daysStartingWith(string pattern)
    => days.Where(d => d.ToString().StartsWith(pattern));
```

The pattern variable is referenced from within the lambda and is therefore captured in a closure.

In this example, `Where` expects a function that takes a `DayOfWeek` and returns a `bool`. In reality, the function expressed by the lambda expression also uses the value of `pattern`, which is captured in a closure, to calculate its result.

This is interesting. If you were to look at the function expressed by the lambda with a more mathematical eye, you might say that it's actually a *binary* function that takes a `DayOfWeek` and a `string` (the pattern) as inputs, and yields a `bool`. As programmers, however, we're usually mostly concerned about the function signature, so you might be more likely to look at it as a *unary* function from `DayOfWeek` to `bool`. Both perspectives are valid: the function must conform to its unary signature, but it depends on two values to do its work.

DICTIONARIES

Dictionaries are fittingly also called *maps* (or *hashtables*); they're data structures that provide a very direct representation of a function. They literally contain the association of *keys* (elements from the domain) to *values* (the corresponding elements from the codomain).

We normally think of dictionaries as data, so it's enriching to change perspectives for a moment and consider them as functions. Dictionaries are appropriate for representing functions that are completely arbitrary, where the mappings can't be computed but *must* be stored exhaustively. For example, to map Boolean values to their names in French, you could write the following.

Listing 1.9 A function can be exhaustively represented with a dictionary

```
var frenchFor = new Dictionary<bool, string>
{
    [true] = "Vrai",
    [false] = "Faux",
}
```

C# 6 dictionary initializer syntax

⁵ A *closure* is the combination of the lambda expression itself along with the context in which that lambda is declared (that is, all the variables available in the scope where the lambda appears).

```
};
frenchFor [true]
// => "Vrai"
```

Function application is performed with a lookup.

The fact that functions can be represented with dictionaries also makes it possible to optimize computationally expensive functions by storing their computed results in a dictionary instead of recomputing them every time.

For convenience, in the rest of the book, I'll use the term *function* to indicate one of the C# representations of a function, so keep in mind that this doesn't quite match the mathematical definition of the term. You'll learn more about the differences between mathematical and programming functions in chapter 2.

1.4 Higher-order functions

Now that you've got an understanding of what FP is and we've reviewed the functional features of the language, it's time to start exploring some concrete functional techniques. We'll begin with the most important benefit of functions as first-class values: it gives you the ability to define higher-order functions (HOFs).

HOFs are functions that take other functions as inputs or return a function as output, or both. I'll assume that you've already used HOFs to some extent, such as with LINQ. We'll use HOFs *a lot* in this book, so this section should act as a refresher and will possibly introduce some use cases for HOFs that you may be less familiar with. HOFs are fun, and most of the examples in this section can be run in the REPL. Make sure you try a few variations of your own along the way.

1.4.1 Functions that depend on other functions

Some HOFs take other functions as arguments and invoke them in order to do their work, somewhat like a company may subcontract some of its work to another company. You've seen some examples of such HOFs earlier in this chapter: `Sort` (an instance method on `List`) and `Where` (an extension method on `IEnumerable`).

`List.Sort`, when called with a `Comparison` delegate, is a method that says: "OK, I'll sort myself, as long as you tell me how to compare any two elements that I contain." `Sort` does the job of sorting, but the caller can decide what logic to use for comparing.

Similarly, `Where` does the job of filtering, and the caller decides what logic determines whether an element should be included. You can represent the type of `Where` graphically, as shown in figure 1.4.

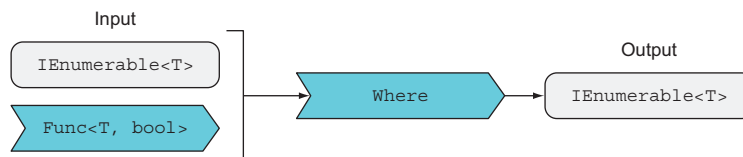


Figure 1.4 `Where` takes a predicate function as input.

Let's look at an idealized implementation of `Where`.⁶

Listing 1.10 `Where`: a typical HOF that iteratively applies the given predicate

The task of iterating over the list is an implementation detail of `Where`.

```
public static IEnumerable<T> Where<T>
    (this IEnumerable<T> ts, Func<T, bool> predicate)
{
    foreach (T t in ts)
        if (predicate(t))
            yield return t;
}
```

The criterion determining which items are included is decided by the caller.

The `Where` method is responsible for the sorting logic, and the caller provides the *predicate*, which is the criterion based on which the `IEnumerable` should be filtered.

As you can see, HOFs can help with the separation of concerns in cases where logic can't otherwise be easily separated. `Where` and `Sort` are examples of *iterated applications*—the HOF will apply the given function repeatedly for every element in the collection.

One very crude way of looking at this is that you're passing as the argument a function whose code will ultimately be executed inside the body of a loop within the HOF—something you couldn't do by only passing static data. The general scheme is shown in figure 1.5.

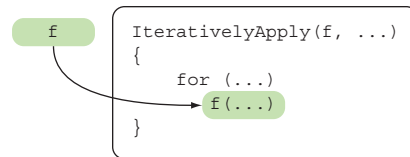


Figure 1.5 A HOF that iteratively applies the function given as an argument

Optional execution is another good candidate for HOFs. This is useful when you want to invoke a given function only in certain conditions, as illustrated in figure 1.6.

For example, imagine a method that looks up an element from the cache. A delegate can be provided and can be invoked in case of a cache miss.

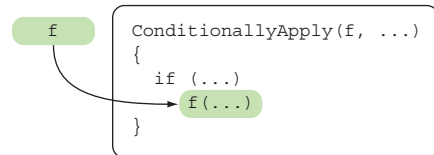


Figure 1.6 A HOF that conditionally applies the function given as an argument

Listing 1.11 A HOF that optionally invokes the given function

```
class Cache<T> where T : class
{
    public T Get(Guid id) => //...

    public T Get(Guid id, Func<T> onMiss)
        => Get(id) ?? onMiss();
}
```

⁶ This implementation is functionally correct, but it lacks the error checking and optimizations in the LINQ implementation.

The logic in `onMiss` could involve an expensive operation such as a database call, so you wouldn't want this to be executed unnecessarily.

The preceding examples illustrate HOFs that take a function as input (often referred to as a *callback* or a *continuation*) and use it to perform a task or to compute a value.⁷ This is perhaps the most common pattern for HOFs, and it's sometimes referred to as inversion of control: the caller of the HOF decides what to do by supplying a function, and the callee decides when to do it by invoking the given function.

Let's look at some other scenarios in which HOFs come in handy.

1.4.2 Adapter functions

Some HOFs don't *apply* the given function at all, but rather return a new function, somehow related to the function given as an argument. For example, say you have a function that performs integer division:

```
Func<int, int, int> divide = (x, y) => x / y;
divide(10, 2) // => 5
```

You want to change the order of the arguments so that the divisor comes first. This could be seen as a particular case of a more general problem: changing the order of the arguments.

You can write a generic HOF that modifies any binary function by swapping the order of its arguments:

```
static Func<T2, T1, R> SwapArgs<T1, T2, R>(this Func<T1, T2, R> f)
    => (t2, t1) => f(t1, t2);
```

Technically, it would be more correct to say that `SwapArgs` returns a *new* function that invokes the given function with the arguments in the reverse order. But on an intuitive level, I find it easier to think that I'm getting back a modified version of the original function.

You can now modify the original division function by applying `SwapArgs`:

```
var divideBy = divide.SwapArgs();
divideBy(2, 10) // => 5
```

Playing with this sort of HOF leads to the interesting idea that functions aren't set in stone: if you don't like the interface of a function, you can call it via another function that provides an interface that better suits your needs. That's why I call these *adapter functions*.⁸

⁷ This is perhaps the most common pattern for HOFs, and it's sometimes referred to as *inversion of control*: the caller of the HOF decides *what* to do by supplying a function, and the function decides *when* to do it by invoking the given function.

⁸ The well-known adapter pattern in OOP can be seen as applying the idea of adapter functions to an object's interface.

1.4.3 Functions that create other functions

Sometimes you'll write functions whose primary purpose is to create other functions—you can think of them as *function factories*. The following example uses a lambda to filter a sequence of numbers, keeping only those divisible by 2:

```
var range = Enumerable.Range(1, 20);

range.Where(i => i % 2 == 0)
// => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

What if you wanted something more general, like being able to filter for numbers divisible by any number, n ? You could define a function that takes n and yields a suitable predicate that will evaluate whether any given number is divisible by n :

```
Func<int, bool> isMod(int n) => i => i % n == 0;
```

We haven't looked at a HOF like this before: it takes some static data and returns a function. Let's see how you can use it:

```
using static System.Linq.Enumerable;

Range(1, 20).Where(isMod(2)) // => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Range(1, 20).Where(isMod(3)) // => [3, 6, 9, 12, 15, 18]
```

Notice how you've gained not only in generality, but also in readability! In this example, you're using the `isMod` HOF to produce a function, and then you're feeding it as input to another HOF, `Where`, as shown in figure 1.7.

You'll see many more uses of HOFs in the book. Eventually you'll look at them as regular functions, forgetting that they're higher order. Let's now look at how they can be used in a scenario closer to everyday development.

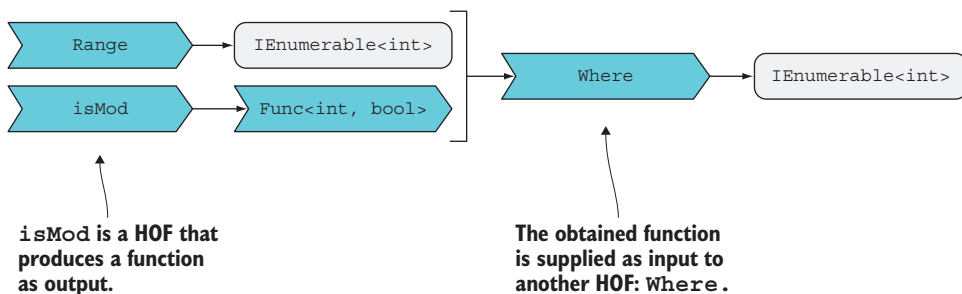


Figure 1.7 A HOF that produces a function that's given as input to another HOF

1.5 Using HOFs to avoid duplication

Another common use case for HOFs is to encapsulate setup and teardown operations. For example, interacting with a database requires some setup to acquire and open a connection, and some cleaning up after the interaction to close the connection and return it to the underlying connection pool. In code, it looks like the following.

Listing 1.12 Connecting to a DB requires some setup and teardown

```
string connString = "myDatabase";

var conn = new SqlConnection(connString);
conn.Open();

// interact with the database...

conn.Close();
conn.Dispose();
```

Setup: acquire and open a connection.

Teardown: close and release the connection.

The setup and teardown are always identical, regardless of whether you're reading or writing to the database, or performing one or many actions. The preceding code is usually written with a using block, like this:

```
using (var conn = new SqlConnection(connString))
{
    conn.Open();
    // interact with the database...
}
```

This is both shorter and better,⁹ but it's still essentially the same. Consider the following example of a simple DbLogger class with a couple of methods that interact with the database: Log inserts a given log message, and GetLogs retrieves all logs since a given date.

Listing 1.13 Duplication of setup/teardown logic

```
using Dapper;
// ...

public class DbLogger
{
    string connString;

    public void Log(LogMessage msg)
    {
        using (var conn = new SqlConnection(connString))
```

Exposes Execute and Query as extension methods on the connection

Assume this is set in the constructor.

Setup

⁹ It's shorter because Dispose will be called as you exit the using block, and it will in turn call Close; it's better because the interaction will be wrapped in a try/finally, so that the connection will be disposed even if an exception is thrown in the body of the using block.

```

Teardown is performed as part of Dispose.
}
}

        {
            int affectedRows = conn.Execute("sp_create_log"
                , msg, CommandType: CommandType.StoredProcedure);
        }

        public IEnumerable<LogMessage> GetLogs(DateTime since)
        {
            var sqlGetLogs = "SELECT * FROM [Logs] WHERE [Timestamp] > @since";
            using (var conn = new SqlConnection(connString))
            {
                return conn.Query<LogMessage>(sqlGetLogs
                    , new {since = since});
            }
        }
    }
}

```

Persists the LogMessage to the DB

Setup

Teardown


Queries the DB and deserializes the results

Notice that the two methods have some duplication, namely the setup and teardown logic. Can we get rid of the duplication?

The specifics of the interaction with the database are irrelevant for this discussion, but if you're interested, the code uses the Dapper library (documented on GitHub: <https://github.com/StackExchange/dapper-dot-net>), which is a thin layer on top of ADO.NET allowing you to interact with the database through a very simple API:

- Query queries the database and returns the deserialized LogMessages.
- Execute runs the stored procedure and returns the number of affected rows (which we're disregarding).

Both methods are defined as extension methods on the connection. More importantly, notice how in both cases, the database interaction depends on the acquired connection and returns some data. This will allow you to represent the database interaction as a function from IDbConnection to "something."

 **ASYNCHRONOUS I/O OPERATIONS** In a real-world scenario, I'd recommend you always perform I/O operations asynchronously (so, in this example, GetLogs should really call QueryAsync and return a Task<IEnumerable<LogMessage>>). But asynchrony adds a level of complexity that's not helpful while you're trying to learn the already challenging ideas of FP. For pedagogical purposes, I'll wait until chapter 13 to discuss asynchrony.

As you can see, Dapper exposes a pleasant API, and it will even open the connection if necessary. But you're still required to create the connection, and you should dispose it as soon as possible, once you're done with it. As a result, the meat of your database calls ends up sandwiched between identical pieces of code that perform setup and teardown. Let's look at how you can avoid this duplication by extracting the setup and teardown logic into a HOF.

1.5.1 Encapsulating setup and teardown into a HOF

You're looking to write a function that performs setup and teardown and that's parameterized on what to do in between. This is a perfect scenario for a HOF, because you can represent the logic in between with a function.¹⁰ Graphically, it looks like figure 1.8.

Because connection setup and teardown are much more general than DbLogger, they can be extracted to a new ConnectionHelper class.

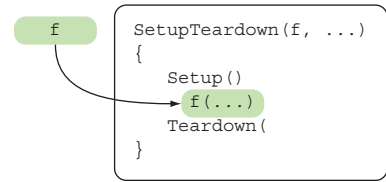


Figure 1.8 A HOF that wraps a given function between setup and teardown logic

Listing 1.14 Encapsulating setup and teardown of the database connection into a HOF

```

using System;
using System.Data;
using System.Data.SqlClient;

public static class ConnectionHelper
{
    public static R Connect<R>(string connString
        , Func<IDbConnection, R> f)
    {
        Setup
        {
            using (var conn = new SqlConnection(connString))
            {
                conn.Open();
                return f(conn);
            }
        }
        Teardown
    }
}
  
```

What happens in between is now parameterized.

The Connect function performs the setup and teardown, and it's parameterized by what it should do in between. The signature of the body is interesting; it takes an IDbConnection (through which it will interact with the database), and returns a generic object R. In the use cases we've seen, R will be IEnumerable<LogMessage> in the case of the query and int in the case of the insert. You can now use the Connect function in DbLogger as follows:

```

using Dapper;
using static ConnectionHelper;

public class DbLogger
{
    string connString;

    public void Log(LogMessage message)
        => Connect(connString, c => c.Execute("sp_create_log"
            , message, CommandType.StoredProcedure));
}
  
```

¹⁰ For this reason, you may hear this pattern inelegantly called “hole in the middle.”

```

public IEnumerable<LogMessage> GetLogs(DateTime since)
    => Connect(connString, c => c.Query<LogMessage>(@"SELECT *
        FROM [Logs] WHERE [Timestamp] > @since", new {since = since}));
}

```

You got rid of the duplication in `DbLogger`, and `DbLogger` no longer needs to know the details about creating, opening, or disposing of the connection.

1.5.2 *Turning the using statement into a HOF*

The previous result is satisfactory. But to take the idea of HOFs a bit further, let's be a bit more radical. Isn't the `using` statement itself an example of setup/teardown? After all, a `using` block always does the following:

- *Setup*—Acquires an `IDisposable` resource by evaluating a given declaration or expression
- *Body*—Executes what's inside the block
- *Teardown*—Exits the block, causing `Dispose` to be called on the object acquired in the setup

So...Yes, it is! At least sort of. The setup isn't always the same, so it too needs to be parameterized. We can then write a more generic setup/teardown HOF that performs the `using` ceremony.

This is the kind of widely reusable function that belongs in a library. Throughout the book, I'll show you many such reusable constructs that have gone into my `LaYumba.Functional` library, enabling a better experience when coding functionally.

Listing 1.15 A HOF that can be used instead of the `using` statement

```

using System;

namespace LaYumba.Functional
{
    public static class F
    {
        public static R Using<TDisp, R>(TDisp disposable
            , Func<TDisp, R> f) where TDisp : IDisposable
        {
            using (disposable) return f(disposable);
        }
    }
}

```

The preceding listing defines a class called `F` that will contain the core functions of our functional library. The idea is that these functions should be made available without qualification with `using static`, as shown in the next code sample.

This `Using` function takes two arguments: the first is the disposable resource, and the second is the function to be executed before the resource is disposed. With this in place, you can rewrite the `Connect` function more concisely:

```
using static LaYumba.Functional.F;

public static class ConnectionHelper
{
    public static R Connect<R>(string connStr, Func<IDbConnection, R> f)
        => Using(new SqlConnection(connStr)
            , conn => { conn.Open(); return f(conn); });
}
```

The `using static` on the first line enables you to invoke the `Using` function as a sort of global replacement for the `using` statement. Notice that unlike the `using statement`, calling the `Using` function is an *expression*.¹¹ This has a couple of benefits:

- It allows you to use the more compact expression-bodied method syntax.
- An expression has a value, so the `Using` function can be composed with other functions.

We'll dig deeper into the ideas of composition and statements vs. expressions in section 5.5.1.

1.5.3 Tradeoffs of HOFs

Let's look at what you've achieved by comparing the initial and the refactored versions of one of the methods in `DbLogger`:

```
// initial implementation
public void Log(LogMessage msg)
{
    using (var conn = new SqlConnection(connString))
    {
        int affectedRows = conn.Execute("sp_create_log"
            , msg, CommandType.StoredProcedure);
    }
}

// refactored implementation
public void Log(LogMessage message)
    => Connect(connString, c => c.Execute("sp_create_log"
        , message, CommandType.StoredProcedure));
```

This is a good illustration of the benefits you can get from using HOFs that take a function as an argument:

- *Conciseness*—The new version is obviously more concise. Generally speaking, the more intricate the setup/teardown and the more widely it's required, the more benefit you get by abstracting it into a HOF.

¹¹ Here's a quick refresher on the difference: *expressions* return a value; *statements* don't.

- *Avoid duplication*—The whole setup/teardown logic is now performed in a single place.
- *Separation of concerns*—You’ve managed to isolate connection management into the ConnectionHelper class, so DbLogger need only concern itself with logging-specific logic.

Let’s look at how the call stack has changed. Whereas in the original implementation the call to Execute happened on the stack frame of Log, in the new implementation they’re four stack frames apart (see figure 1.9).

```
class DbLogger
{
    public void Log(LogMessage message)
    => Connect(connString, c => c.Execute("sp_create_log",
        , message, CommandType: CommandType.StoredProcedure));
}

public static class ConnectionHelper
{
    public static R Connect<R>(string connStr, Func<IDbConnection, R> f)
    => Using(new SqlConnection(connStr)
        , conn => { conn.Open(); return f(conn); });
}

public static class F
{
    public static R Using<TDisp, R>(TDisp disposable
        , Func<TDisp, R> f) where TDisp : IDisposable
    {
        using (var disp = disposable) return f(disp);
    }
}
```

Figure 1.9 HOFs call back into the calling function.

When Log executes, the code calls Connect, passing it the callback function to invoke when the connection is ready. Connect in turn repackages the callback into a new callback, and passes it to Using.

So, HOFs also have some drawbacks:

- You’ve increased stack use. There’s a performance impact, but it’s negligible.
- Debugging the application will be a bit more complex because of the callbacks.

Overall, the improvements made to DbLogger make it a worthy tradeoff.

You probably agree by now that HOFs are very powerful tools, although overuse can make it difficult to understand what the code is doing. Use HOFs when appropriate, but be mindful of readability: use short lambdas, clear naming, and meaningful indentation.

1.6 Benefits of functional programming

The previous section demonstrated how you can use HOFs to avoid duplication and achieve better separation of concerns. Indeed, one of the advantages of FP is its *conciseness*: you can achieve the same results with fewer lines of code. Multiply that by the tens of thousands of lines of code in a typical application, and conciseness also has a positive effect on the maintainability of the application.

There are many more benefits to be reaped by applying the functional techniques you'll learn in this book, and they roughly fall into three categories:

- *Cleaner code*—Apart from the previously mentioned conciseness, FP leads to more expressive, more readable, and more easily testable code. Clean code is not just a developer's intellectual pleasure, but it also leads to huge economic benefits for the business through reduced maintenance costs.
- *Better support for concurrency*—Several factors, from multi-core CPUs to distributed systems, bring a high degree of concurrency to your applications. Concurrency is traditionally associated with difficult problems such as deadlocks, lost updates, and more; FP offers techniques that prevent these problems from occurring. You'll see an introductory example in chapter 2 and more advanced examples toward the end of the book.
- *A multi-paradigm approach*—They say that if the only tool you have is a hammer, every problem will look like a nail. Conversely, the more angles from which you can view a given problem, the more likely it is that you'll find an optimal solution. If you're already proficient in OOP, learning a different paradigm such as FP will inevitably give you a richer perspective. When faced with a problem, you'll be able to consider several approaches and pick the most effective.

Exercises

I recommend you take the time to do the the exercises and come up with a few of your own along the way. The code samples repository on GitHub (<https://github.com/la-yumba/functional-csharp-code>) includes placeholders so that you can write, compile, and run your code with minimal setup effort. It also includes solutions that you can check your results against:

- 1 Browse the methods of `System.Linq.Enumerable` (<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable>). Which are HOFs? Which do you think imply iterated application of the given function?
- 2 Write a function that negates a given predicate: whenever the given predicate evaluates to `true`, the resulting function evaluates to `false`, and vice versa.
- 3 Write a method that uses quicksort to sort a `List<int>` (return a new list, rather than sorting it in place).
- 4 Generalize the previous implementation to take a `List<T>`, and additionally a `Comparison<T>` delegate.

- 5 In this chapter, you've seen a `Using` function that takes an `IDisposable` and a function of type `Func<TDisp, R>`. Write an overload of `Using` that takes a `Func<IDisposable>` as the first parameter, instead of the `IDisposable`. (This can be used to avoid warnings raised by some code analysis tools about instantiating an `IDisposable` and not disposing it.)

Summary

- FP is a powerful paradigm that can help you make your code more concise, maintainable, expressive, robust, testable, and concurrency-friendly.
- FP differs from OOP by focusing on functions, rather than objects, and on data transformations rather than state mutation.
- FP can be seen as a collection of techniques that are based on two fundamental tenets:
 - Functions are first-class values
 - In-place updates should be avoided
- Functions in C# can be represented with methods, delegates, and lambdas.
- FP leverages higher-order functions (functions that take other functions as input or output); hence the necessity for the language to have functions as first-class values.

Functional Programming in C#

Enrico Buonanno



Functional programming changes the way you think about code. For C# developers, FP techniques can greatly improve state management, concurrency, event handling, and long-term code maintenance. And C# offers the flexibility that allows you to benefit fully from the application of functional techniques. This book gives you the awesome power of a new perspective.

Functional Programming in C# teaches you to apply functional thinking to real-world problems using the C# language. You'll start by learning the principles of functional programming and the language features that allow you to program functionally. As you explore the many practical examples, you'll learn the power of function composition, data flow programming, immutable data structures, and monadic composition with LINQ.

What's inside

- Write readable, team-friendly code
- Master async and data streams
- Radically improve error handling
- Event sourcing and other FP patterns

Written for proficient C# programmers with no prior FP experience.

Enrico Buonanno studied computer science at Columbia University and has 15 years of experience as a developer, architect, and trainer.

"Functional programming can make your head explode. This book stitches it back together."

—Daniel Marbach, Particular Software

"A top-ten technical book that turned me on to functional programming. The author does a fantastic job of organizing the content in a clear and concise manner—with humor."

—Alex Basile, Bloomberg

"Mind-bending. If you are an experienced C# developer with lots of questions about good code practice and general architecture, this book lifts you to another level."

—Aurélien Gounot, SNCF

"Best way to start learning FP using C# and getting in touch with future versions, C# 6 and C# 7."

—Gonzalo Barba López, MoneyMate

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

 **MANNING** US \$49.99 / Can \$65.99 [including eBook]

