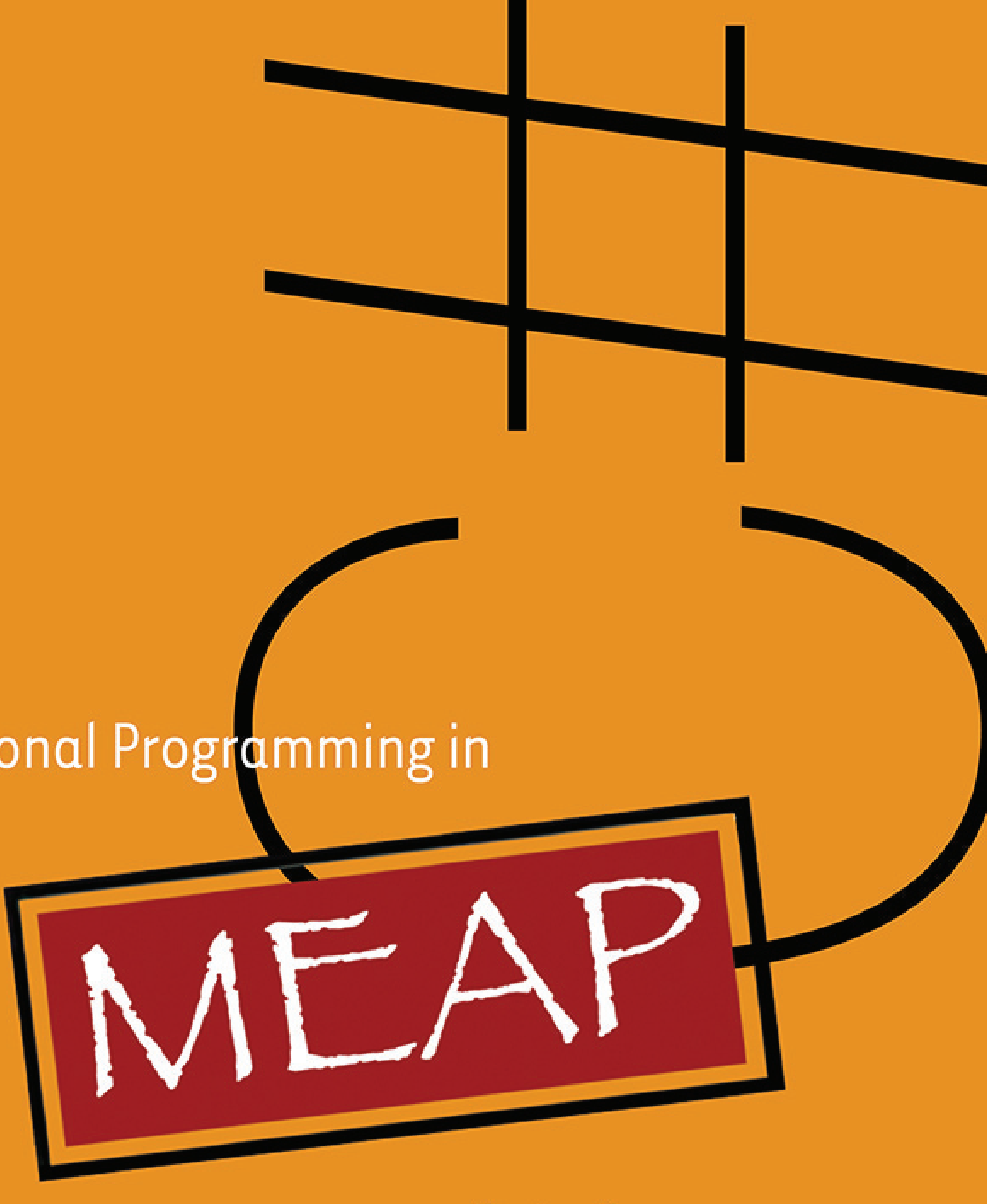
Functional Programming in



Enrico Buonanno

MANNING

MEAP Edition
Manning Early Access Program

Functional Programming in C#

Version 1

Welcome

Thank you for purchasing the MEAP for *Functional Programming in C#*. I'm excited to see the book reach this stage and look forward to its continued development and eventual release. This book is written for proficient C# developers who want to further improve their skills by leveraging functional programming techniques.

I propose a pragmatic approach to functional programming, so I strived to make the content relevant for real-world scenarios. The underlying theme is that integrating the functional techniques shown in the book will make your C# programs more maintainable and more robust, and will make you a better-rounded developer.

We're releasing the first 3 chapters to start with. Chapter 1 explains the principles of functional programming, and how well C# supports programming in a functional style. It then delves specifically into functions as first-class values, and some practical uses of higher-order functions.

Chapter 2 explains function purity and its very practical implications in concurrent scenarios, and how it affects testability.

Chapter 3 is about relearning something that we take for granted: function signatures. It explains how some function signatures are better than others, and how they can be improved with the use of specific types, such as the Option type. The Option type will accompany us in further chapters as a vehicle for introducing many other functional concepts.

Looking ahead at chapters 4 and 5, we'll concentrate on defining workflows by composing functions, and working with lists and options. With these basic concepts covered, Part 2 will introduce more functional constructs and techniques. By the end of part 2, you'll have acquired a set of tools enabling you to effectively tackle many programming tasks using a functional approach.

Part 3 will tackle more advanced topics, including lazy evaluation, stateful computations, asynchrony and event streams.

As you're reading, I hope you'll take advantage of the Author Online forum. I'll be happy to receive your feedback and suggestions, and answer your questions. Code samples are available on https://github.com/la-yumba/functional-csharp-code and will be updated as the book progresses.

— Enrico Buonanno

brief contents

1 Introducing Functional Programming

In this chapter:

- benefits and tenets of functional programming
- 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 objective with this book is that you'll never look at code with the same eyes as before!

The learning process can be a bumpy ride, and can take you 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 in the back of your mind as you start on this journey, like: 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 it differs from object-oriented programming (OOP), we'll then look at 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 we'll illustrate with a practical example.

To make sure learning FP is an enjoyable and rewarding process:

1. be patient: you may have to read some sections more than once; you may put the book down for a few weeks, and find that when you pick it up again something that seemed obscure starts to make more sense
2. experiment in code: the book provides examples and exercises, but it’s essential that you play around and "get your hands dirty" in your own projects

1.1 The FP view of the world

Object-oriented programming (OOP) has been the de facto standard for developing "Line Of Business" (LOB) applications in the last 20 years or so. As a result, object orientation has become almost second nature to developers working in industry — a second nature we must now question, in order to understand the different paradigm offered by FP.

In OOP, we think of objects as the basic units that the program deals in. Simple objects can be defined from primitive types, and composed to create complex object hierarchies. As OO developers, when facing a programming task, we think of what objects we can define, to model the application domain. Let’s see how FP differs.

1.1.1 Functions as elementary units

In FP, functions are also treated as basic units. When facing a programming task, we think of what functions we need to define, to model the tasks that our program needs to perform.

For example, to model the behaviour of greeting a person, you can:

1. OOP: use an IGreeter object:

   ```
   interface IGreeter
   {
      Greeting Greet(Person p);
   }
   ```

   this code states that an `IGreeter` *has* a method that will take a `Person` and yield a `Greeting`

2. FP: use a Greeter function:

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

   this code states that a `Greeter` *is* a function that will take a `Person` and yield a `Greeting`

Does it make sense to encapsulate the behaviour of greeting in an object that implements `IGreeter`, as the OO paradigm leads us to do, or should it just be expressed as a "free standing" function, which would be the natural choice in FP?

The answer depends on context and on preference, and as you learn to think functionally you will probably use functions more often, avoiding the proliferation of objects sometimes necessary for even very simple behaviours when using a purely OO approach.

As you can see, there is no contradiction in principle between OOP and FP, but rather a change in focus. While FP puts more of an emphasis on functions, a good object model is still essential; after all, functions act on objects!

This also means that OOP can be effectively combined with FP; you can use OOP to model your domain entities and the main actors in your system, and use FP when it comes to actually implementing the behaviours of your application.

Some important programming concepts apply to both objects and functions, but take a different meaning in each case:

- **Composition**: like objects, functions can be composed to create complex programs. But, unlike objects, composing functions does not create hierarchies: the composition of 2 or more functions is just another function; this will be the focus of Chapter 4.
- **Interfaces**: just as objects expose interfaces (whether explicitly, by implementing a given interface, or implicitly, by virtue of the members they expose), functions also have interfaces: *the function signature*. In C#, explicit object interfaces are declared with the `interface` keyword, while function signatures are explicitly declared with the `delegate` keyword — the previous `IGreeter`/`Greeter` example demonstrates this.

To further demonstrate that OOP and FP are not mutually exclusive approaches, let’s make a thought experiment — and, yes, it’s deliberately provocative.

You may know that the "I" in the SOLID principles[1] stands for *interface segregation*. Roughly, it means that interfaces should be specific, and should not include "too many" methods. It turns out that interface segregation helps to keep a system decoupled.

The question is — how many methods is too many? If we want to maximize decoupling in our system, we would take interface segregation to its ultimate conclusion; namely, that an interface should only include *one* method. In this case, an interface includes no information other than the function signature, and an object can be reduced to a function — like the `Greeter` function above. So, OOP with interface segregation pushed to the extreme can be reduced to FP.

[1] The SOLID principles are 5 basic principles for good OO design. If you consider yourself an OO programmer, make sure you’re familiar with them!

1.1.2 Programs as functions

Part of learning to think functionally is seeing functions everywhere:

- square root is a function: it takes a number *x* and returns a number *y* such that *y* * *y* = *x*
- a metal press is a function: it takes a sheet of metal and returns it in the shape of, say, a car door
- a blender is a function: it takes oranges and returns orange juice

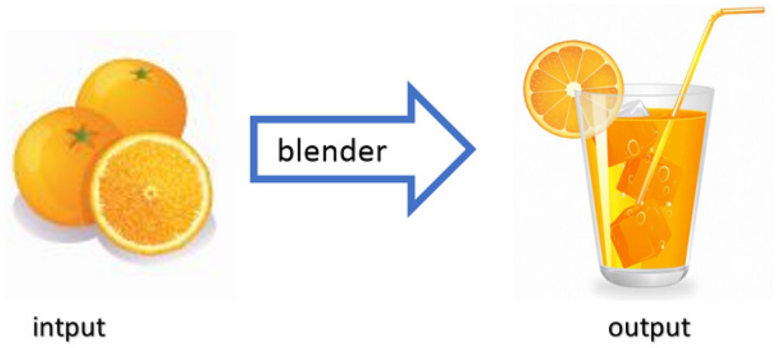


Figure 1.1. Many entities can be viewed as functions rather than objects

Another part is to try to see how functions can be related. From complex to simple: you can use short, simple functions that do very little, and compose them to perform more complex tasks. For example, normalizing the input of an email field can be seen as the composition of the two simpler task, of normalizing the casing and removing any spurious whitespace; in pseudocode:

```
normalizeEmail = removeWhitespace • toLowerCase
```

And, conversely, from simple to complex:

- the square root function can be composed with the variance function to calculate standard deviation
- the metal-press function can be composed with a spray-paint function to get a painted door, and with many more functions to get a fully automated factory that takes raw materials and energy and returns finished cars and waste
- the blender function can be composed with a money-changing function and several others to get a fully automated fruit bar, that takes money and raw foods and returns healthy drinks

Ultimately, we can think of our programs as functions, or as sets of functions. For example, you can view a web server as a function that takes HTTP requests as input, and yields responses as output.

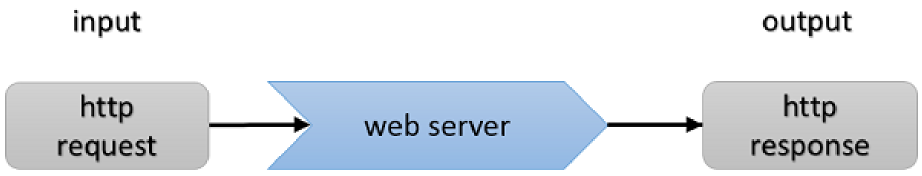


Figure 1.2. A web server can be viewed as a function mapping requests to responses

"Ok, but when I click on 'purchase' — you may protest — the web server doesn't just return an HTTP response: it also creates an order!" That's correct, and this is called a*side effect*; side effects should also be viewed as outputs — we'll develop this topic in Chapter 2.

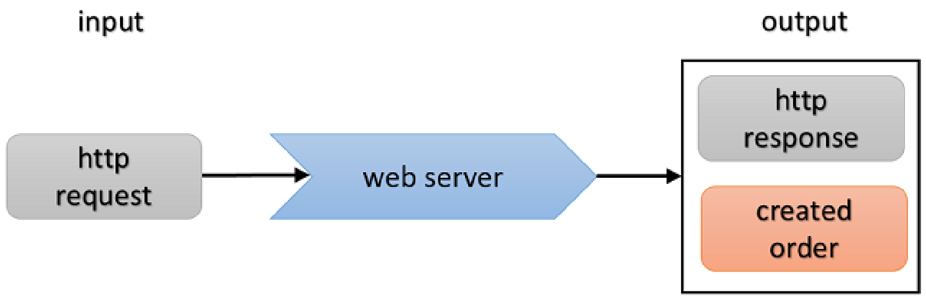


Figure 1.3. In the function methaphor, side effects are also viewed as outputs

1.1.3 Definition of functional programming

So, what exactly is functional programming? As with most things that matter, it is difficult to precisely lay your finger on the essence of functional programming. At a very high level, we can say that FP is a style of programming based on 2 fundamental principles:

1. **Functions are 1st-class values**: functions can be used as input or as return value of other functions, can be assigned to variables, and so on, just like values of any other type.

   For example, instead of using the `foreach` statement:

   ```
   var ints = new List<int> {1, 2, 3, 4, 5};

   foreach (var i in ints)
      Console.WriteLine(i);
   ```

   we can write:

   ```
   ints.ForEach(Console.WriteLine);
   ```

because we are able to treat `WriteLine` as a value that we can pass to the `ForEach` method — a method on `List<T>` that takes an action, which it will invoke for each item in the list.

2. **In-place updates should be avoided**, where "place" means a place in memory. In-place updates, such as `x++`, are also called *destructive* updates: whatever value `x` was holding is "destroyed" when overwritten with the new, incremented value. In fact, "purely" functional languages do not allow in-place updates at all.

   In-place updates occur whenever we *mutate*, that is, change the state of an object or a collection; so, if we want to follow the functional paradigm, we should refrain from state mutation altogether.

   For example, a method that sorts a list should not sort its elements in place, but leave the list unaffected, and return a new list with the same elements as the input list, but sorted:

```
var original = new[] { 5, 7, 1 };
var sorted = original.OrderBy(x => x).ToList();

Assert.AreEqual(new[] { 5, 7, 1 }, original);
Assert.AreEqual(new[] { 1, 5, 7 }, sorted);
```

   The test above demonstrates that this is indeed the behaviour of LINQ’s `OrderBy` method. On the other hand, if we use `List.Sort`, the list is sorted in place, so that the list in its original state is lost forever, violating the FP approach:

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

   The reason why you see both approaches in the framework is historical: `List.Sort` predates .NET 3, which marked a decisive turn in a functional direction.

While most people will agree on these 2 principles as the fundamental tenets of FP, their application gives rise to a series of practices and techniques (see sidebar below), and it is somewhat up to discussion which techniques should be considered essential, and which should be included in a book like this.

One possible approach is to look at FP as *a set of tools* that can be used in addressing your programming tasks. Although purists may disagree, I would encourage you to take this pragmatic approach, and try to understand functional techniques as tools that you can learn and add to your programmer’s toolset.

It is also worth noting that, although the concepts of functional programming are language-independent, in practice, functional programming in C# is not the same as functional programming in another language. Each language has its strengths and weaknesses, and I would not find it meaningful to write some Haskell-like code in C#; instead, I prefer to show how to best leverage the functional features of

C#.

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

Subjects that fall under FP

FP is a vast field, that cannot possibly be covered in full in a single book. Some of the techniques and applications associated with FP include:

- Techniques for working with functions (higher-order functions, recursion, partial application, memoization, combinators, etc)
- Functional data types: types that are immutable and usually interacted with through higher-order functions. These types include:
    - Collections (lists, maps, trees, etc)
    - Wrappers for values, enriching a value with some accompanying data (option, either, writer, etc)
    - Abstractions on operations: specialized functions or wrappers for functions that represent, for example, asynchronous or future execution of a computation
- Functions for interacting with such types; the most common being: map, apply, bind, fold, etc.
- Specific applications, such as parsers, for which the functional approach has proven particularly effective

While this list is not exhaustive, it may give you an idea of the variety of topics relevant to FP. My choice has been to include in this book the techniques that I find most useful in the day-to-day development of a typical LOB application.

1.2 How functional a language is C#?

Well... let’s see. Given that the core principles of FP are:

1. functions are 1st-class values
2. in-place updates should be avoided

we need to assess how well C# supports programming with these 2 principles.

Functions can certainly be treated as values in C#: you can assign a function to a field, pass it as an argument to a function, or return a function from a function. 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.

To be fair, there are still some respects in which functions in C# are second-class to objects. For example, unlike classes and interfaces, delegates cannot declare any methods (although you can still achieve the same via extension methods, something we will show in later chapters). You cannot declare a type that inherits from a delegate; nor does the compiler infer the type of functions as well as it does with objects — this last one being perhaps the biggest limitation.

Overall, the support for functions as 1st-class values is pretty good. Now for supporting a programming model that discourages in-place updates. Here the tables are turned: in C# everything is mutable by default, and the programmer has to do a substantial amount of effort to achieve immutability.

Fields and variables are all mutable by default, and 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? While there are a few immutable types in the framework, such as `string` and `DateTime`, 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 further improve in C#7). 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 will continue to improve its support for functional techniques. In this book, you will learn which features can be harnessed, and how to work around its shortcomings.

1.2.1 Functional features in C#6

Code samples in this book are in C#6, and make extensive use language features not available in previous versions of the language. Some of these language features are relevant for FP, namely:

1. expression-bodied members
2. using static
3. getter-only auto-properties

These features are demonstrated in the snippet below.

Listing 1.1. C#6 features relevant for FP

```
using static System.Math; ❷
public class Circle
{
   public Circle(double radius) { Radius = radius; } ❷
   public double Radius { get; } ❸
   public double Area ❶
      => PI * Pow(Radius, 2); ❸
}
```

❶ an expression-bodied property
❷ `using static` enables unqualified access to the static methods in `System.Math`
❸ a getter-only auto-property can be set only in the constructor

Let's examine these relatively recent language features as they appear in the snippet.

1. **Expression bodied members**: the `Area` property is declared with an "expression body" rather than the usual "statement body". This syntax works for both methods and properties; the body is introduced with `=>` instead of the usual `{ }`. As the name suggests, it can only be used if the method/property consists of a single expression.
2. The **using static** statement on the 1st line allows us to "import" the members of a class (in this example, the `System.Math` class). As a result, we can invoke the `PI` and `Pow`members of `Math` without further qualification.
3. **Getter-only auto-properties**, such as `Radius` above implicitly declare a `readonly` backing field. As a result, the field can only be assigned in the constructor or inline.

These features do *not* introduce any new functionality: anything you can do with C#6 you could also do with C#4.5 (with a bit of extra typing). However these features do encourage a more functional programming style:

- Expression bodied members will encourage the use of simple function-like methods, over procedural multi-line methods.
- The `using static` feature will encourage the creation of libraries of functions (such as the library we will develop throughout this book), that can be imported and easily consumed in client code.
- Getter-only auto-properties implicitly declare `readonly` fields, so that they facilitate the definition of immutable types (the `Circle` class demonstrates this: it only has one field, which is `readonly`, so it's immutable).

As a result, it is quite likely that C# code in a few years will look rather different from what you are used to seeing today, and probably more similar to the code in this book.

1.2.2 Functional features in C#7

At the time of this writing, C#7 is still a distant prospect. However, it's interesting to point out that *all* the features for which the language team has currently identified "strong interest" are features normally associated with functional languages. These include, among others:

- 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, for example its type, instead of the value)

- Better syntactic support for Tuples (which makes it attractive to use Tuples instead of defining objects for simple structures)

Although this list is just work in progress, it does highlight the fact that the C# language is poised to continue in its evolution as a multi-paradigm language with an increasingly strong functional component.

What you learn in this book will give you a good foundation to keep up with the evolution of the language and the industry, and have a good understanding of the concepts and motivations behind future versions of the language.

1.3 Functions as 1st class values

Functional programming is only possible in languages in which functions are 1st-class values, that is, functions can be declared, initialized, used as input/output of other functions.

In this section, we'll clarify what we mean by *function*. We'll briefly show what the term means in the mathematical sense; and the various language constructs that support the definition of "functions" in a programming context.

1.3.1 Mathematical functions

Just a *brief* maths refresher: a function is a mapping between two sets, respectively called domain and codomain. Given an element from its domain, a function returns an element from its codomain.

For example, we could define a function mapping lowercase letters to their uppercase counterparts:

```
a → A
b → B
c → C
...
```

In this case, the domain is the set `{a, b, c, ...}` and the codomain the set `{A, B, C, ...}`. Naturally, there are functions for which the domain and codomain are the same set.

In this sense, a function is a completely mathematical object, and the value that a function evaluates to is determined exclusively by its argument. We'll see this is not always the case with their programming counterparts.

1.3.2 Functions in C#: methods, delegates, lambdas

In C#, we don't really have a strict representation for mathematical functions. We typically rely on **methods** instead: for example, methods in the `System.Math` class represent many common mathematical functions.

While methods are the most common representation for functions, we can also use delegates and lambdas. While methods fit into the object-oriented paradigm (they can be used to implement interfaces, can be overloaded, etc), delegates and lambdas are what really enables us to program in a functional style.

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, then provide an implementation. (This is analogous to writing an `interface` and then instantiating a `class` implementing that interface.)

The first step is done using the `delegate` keyword, and providing the signature for the delegate. For example, the declaration for the generic `Comparison` delegate (that compares two objects of type `T`, returning an `int` representing which is greater) looks like this.

Listing 1.2. Declaring a delegate

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

Although you can use this syntax to declare your own delegate types, and this is sometimes useful for clarity, in practice, *most of the time* you will just use one of the delegate types in the framework (we'll discuss some of those in a moment).

Once you have a delegate type, you can "instantiate" it by providing an implementation, for example:

Listing 1.3. Instantiating and using a delegate

```
using System;
using System.Collections.Generic;

class Delegate_Example
{
   Comparison<string> caseInsensitive = (x, y)
      => x.ToUpper().CompareTo(y.ToUpper()); ❶

   public void Sort(List<string> names)
      => names.Sort(caseInsensitive); ❷
}
```

❶ provide an implementation of `Comparison`

❷ use 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 the listing above; hence delegates are the language feature that makes functions 1st-class values in C#.

Lambda expressions, for short just called *lambdas*, are used to declare a function inline. So the above code could have been written like this using a lambda:

Listing 1.4. Declaring a function inline with a lambda

```
class Lambda_Example
{
   public void Sort(List<string> names)
      => names.Sort((x, y) => x.ToUpper().CompareTo(y.ToUpper()));
}
```

So, 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 above example, the compiler not only infers the types of `x` and `y` to be `string`, but also converts the lambda to the delegate type `Comparison` expected by the `Sort` method, given that the provided lambda is in the right form.

Just like methods, delegates and lambdas have access to the variables in the scope in which they are declared. This is particularly useful when leveraging *closures* in lambda expressions.[2] For example:

Listing 1.5. Lambdas have access to variables closed over

```
private IEnumerable<Employee> employees;

public IEnumerable<Employee> FindByName(string pattern)
   => employees.Where(e => e.LastName.StartsWith(pattern));
```

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

So, if we were to look at the function expressed by the lambda with a more mathematical eye, we would say that it is actually a *binary* function that takes an `Employee` *and* a `string` (the name) as inputs, to return a `bool`. In practice, as programmers we are usually mostly concerned about the function signature, so we would look at it as a *unary* function from `Employee` to `bool`.

For convenience, in the rest of the book, I will use the term *function* loosely, to indicate one of the C# representations of a function; so, just keep in mind that this does not quite match the mathematical meaning of the term. We'll learn more

[2] 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.

about the differences between mathematical and programming functions in the next chapter.

1.3.3 Function vs Action

The .NET framework includes a couple of delegate "families" that can represent pretty much any function type (in the loose sense):

- `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 sidebar).

Since the introduction of `Func`, it's become rare to use custom delegates, for example, instead of using a custom delegate like this:

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

you can just use the type:

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

Above, the type of `Greeter` 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 is 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 favour of the more general `Func` and `Action` delegates. For instance, take the representation of a "predicate":[3]

- 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>`

Needless to say, both function types are equivalent. Using `Func` is recommended to avoid a proliferation of delegate types that represent the same function signature; on the other hand, there is still something to be said in favour of the expressiveness

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

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 indicates with the number of arguments that a function accepts:

- a *nullary* function takes no arguments
- a *unary* function takes one argument
- a *binary* function takes 2 arguments
- a *ternary* function takes 3 arguments

and so on. In reality, all functions can be viewed as being unary, since passing *n* arguments is equivalent to passing an *n*-tuple as the only argument.

1.4 Higher-order functions

Higher order functions (HOFs) are functions that take other functions as inputs, and/or return a function as output. We've already seen some examples 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." So `Sort` does the job of sorting, but the caller can decide what logic to use for comparing. Similarly,

`Where` does the job of filtering, while the caller decides what logic determines whether an element should be included. We can represent the type of `Where` graphically like this:

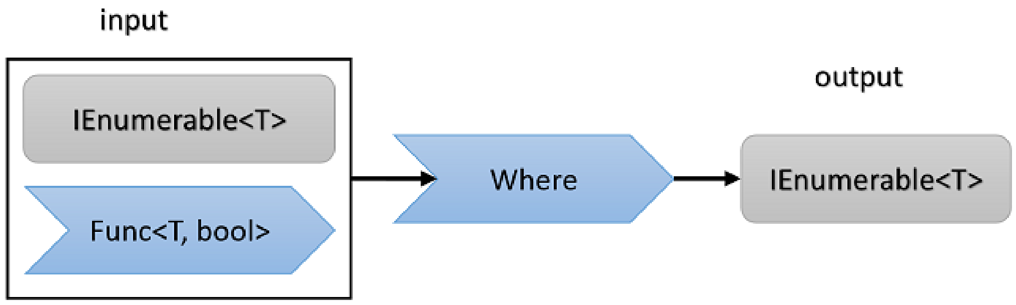


Figure 1.4. `Where` **takes a predicate function as input**

Let's show a possible implementation for `Where`:[4]

[4] This implementation is functionally correct, but lacks the error checking and optimizations in the LINQ implementation.

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

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

❷ The criterion according to which items are included is decided by the caller

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

As you can see, *HOFs can help with separation of concerns, in cases where logic cannot otherwise be easily separated.* `Where` and `Sort` are examples of **iterated application**, that is, 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 are passing as argument a function, whose code will ultimately be injected inside the body of a loop within the HOF — something you could not do by passing a regular object. The general scheme looks like this:

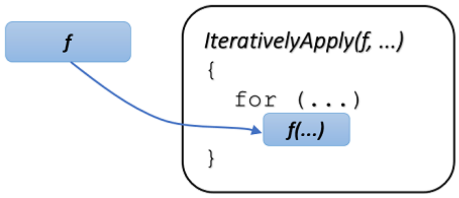


Figure 1.5. A HOF that iteratively applies the function given as argument

There are some more scenarios that are particularly good candidates for HOFs.

Optional execution. This is useful for scenarios in which we want to invoke a given function only depending on certain conditions:

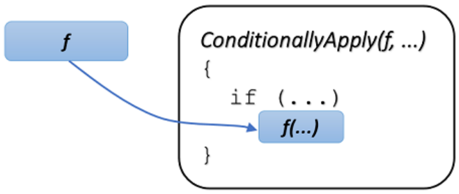


Figure 1.6. A HOF that conditionally applies the function given as argument

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

```
class Cache<T> where T : class
{
   public T Get(Guid id, Func<T> onMiss)
      => Get(id) ?? onMiss();
}
```

The logic in `onMiss` could involve an expensive operation such as a database call, so we don't want this to be executed unless necessary.

Setup/teardown. Here the pattern is that of wrapping a varying action or function between some setup and teardown code that is always the same.[5] Graphically, it looks like this:

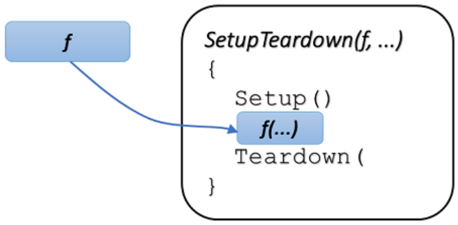


Figure 1.7. A HOF that wraps a given function between setup and teardown logic

We'll see an example of the setup/teardown pattern further down. But first, let's look at some more HOFs.

1.4.1 Functions that modify other functions

All the above examples are common patterns HOFs that *apply* a function that is given as input. There are many more scenarios in which HFOs come in handy. Some don't *apply* the given function at all, but rather return a new function, somehow related to the function (or object) given as argument.

For example, say you have a function that performs integer division:

```
Func<int, int, int> divide = (dividend, divisor) => dividend / divisor;
divide(10, 2); // => 5
```

and you would like 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 arguments. We 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> func)
   => (t2, t1) => func(t1, t2);
```

[5] For this reason, this pattern is also known as "hole in the middle".

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 a modified version of the original function. So, 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 consideration that functions are not set in stone, but you can write functions that modify functions to suit your needs.

I call these "adapter" functions, because they allow you to change the interface (i.e.: the signature) of the original function to suit your needs.

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

1.4.2 In practice: avoiding duplication with HOFs

Imagine the following `DbLogger` class, that has a couple of methods:

- `Log` writes a log message to the database
- `DeleteOldLogs` deletes any logs more than a week old

```
using Dapper;
using System;
using System.Data;
using System.Data.SqlClient;

class DbLogger
{
   string connString;

   public void Log(LogMessage message)
   {
      using (var conn = new SqlConnection(connString))    ❶
      {
         conn.Open();
         conn.Execute("sp_create_log", message              ❷
            , commandType: CommandType.StoredProcedure);
      }                                                     ❸
   }

   public void DeleteOldLogs()
   {
      using (var conn = new SqlConnection(connString))
      {
         conn.Open();
         conn.Execute($@"DELETE [Logs] WHERE
            [Timestamp] < {DateTime.Now.AddDays(-7)}";
      }
   }
}
```

A couple of clarifications on this code:

- let’s assume that the database connection string `connString` is injected; we’re not interested in the details right now
- we’re using the `Dapper` library to access the database via an `Execute` extension method on the connection

Now let’s get to the interesting part. While `Dapper` is great because of the lightweight syntax it offers, it does require that you call `Execute` on an *open* connection; and it is best practice to dispose the connection as soon as possible, once you’re done with it.

As a result, the "meat" of your database calls ends up "sandwiched" between code that opens and disposes the connection. In the listing above, both the `Log` and the `DeleteOldLogs` methods, despite different intents, have the same structure:

❶ **setup**: create and open a connection
❷ **body**: write or delete from the database, respectively
❸ **teardown**: close and dispose the connection (this is done implicitly as we exit the `using` block)

Next, we’ll see how we can extract the setup and teardown logic into a HOF.

1.4.3 *Abstracting setup and teardown logic*

Since the setup and teardown are identical, we would like to remove the duplication. If the actions were more similar (say, two different stored procedures) we could just parameterize a method with the procedure name. But here this is not possible, since we’re calling `Execute` with a completely different set of parameters.

So, what we need is an abstraction that performs the setup and teardown, and is parameterized on the action to invoke in between — a perfect scenario for a HOF. Since the setup and teardown are specific to accessing a database, and have nothing to do with logging, this abstraction can be extracted to a new class:

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

public static class ConnectionExt
{
   public static R Connect<R>(string connString
      , Func<IDbConnection, R> func) ❷
   {
      using (var conn = new SqlConnection(connString)) ❶
      {
         conn.Open();
         return func(conn); ❷
      } ❸
   }
}
```

❶ setup
❷ the body is now a parameter
❸ teardown

As you can see, we've extracted the setup/teardown structure into a `Connect` function, a HOF parameterized by a function that depends on an open connection. `Connect` will perform the standard setup and teardown, and evaluate the function in between, passing it the properly initialized `SqlConnection`. We can now use this in our `DbLogger` as follows:

```
using Dapper;
using static ConnectionExt;

class DbLogger
{
   string connString;

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

   public void DeleteOldLogs()
      => Connect(connString, conn => conn.Execute(DeleteOldLogsSql));

   string DeleteOldLogsSql => $@"DELETE [Logs]
      WHERE [Timestamp] < {DateTime.Now.AddDays(-7)}";
}
```

So, this is already pretty satisfactory: we got rid of the duplication in our `DbLogger` as we intended, and `DbLogger` no longer needs to know the details about opening, closing, or disposing the connection.

Tip

Keep your lambdas short to ensure that your code remains readable. For example, above, I've extracted the creation of the SQL string into the `DeleteOldLogsSql` property to achieve a shorter lambda

1.4.4 Abstracting statement constructs

The above is a satisfactory result. But, just 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:

- **setup**: acquires an `IDisposable` resource by evaluating the declaration/expression inside the brackets
- **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! Well, at least sort-of: because the setup is not 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. We'll add this to our functional library:

```
using System;

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

❶ This is the namespace where we will develop our functional library

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

This `Using` function takes 2 argument: the first one is the disposable resource; the second, the function to be executed before the resource is disposed.

With this in place, we can rewrite our setup/teardown helper function in the `ConnectionExt` class more compactly:

```
using static LaYumba.Functional.F;

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

The `using static` on the first line enables us to invoke the `Using` function as a sort of global replacement for the `using` statement.[6] Notice that, unlike the `using` *statement*, calling the `Using` function is an *expression*. This not only allows us to use the more compact expression-bodied method syntax in the `Connect` function, but also leads us to concentrate on inputs and outputs of functions, instead of the rather imperative style of statements.

1.4.5 Code review: tradeoffs of HOFs

Let's look at what we have achieved, by comparing the initial and the refactored version of one of the methods in `DbLogger`:

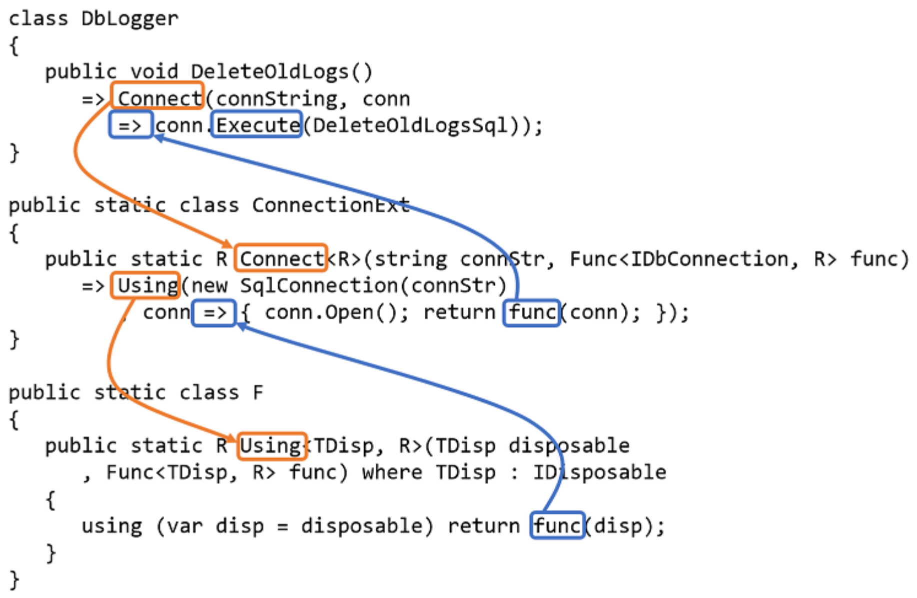
```
// naive implementation
public void DeleteOldLogs()
{
   using (var conn = new SqlConnection(connString))
   {
      conn.Open();
      conn.Execute(DeleteOldLogsSql);
   }
}

// functional implementation
public void DeleteOldLogs()
   => Connect(connString, conn => conn.Execute(DeleteOldLogsSql));
```

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

- **Conciseness**: the new version consists of 2 lines instead, of 8
- **Avoiding duplication**: the whole setup/teardown logic is now being performed in a single place
- **Separation of concerns**: we've managed to isolate Dapper-specific logic into the `ConnectionExt` class, so that `DbLogger` need only concern itself with logging-specific logic. If we were to change Dapper for a different data access mechanism, we could do so with fewer changes to `DbLogger`.

Let's look at how the callstack has changed. Whereas in the original implementation the call to `Execute` happens on the stack frame of `DeleteOldLogs`, in the new implementation they are 4 stack frames apart:

[6] If you don't like the idea, or are on an older version of C#, you could type `F.Using` instead.



```
class DbLogger
{
   public void DeleteOldLogs()
      => Connect(connString, conn
         => conn.Execute(DeleteOldLogsSql));
}

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

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

Figure 1.8. HOFs call back into the calling function

So, when you enter `DeleteOldLogs`, the code will call `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:

- we have increased stack use; there's a performance impact, but it's truly negligible
- it will be a bit more complex to debug the application because of the callbacks

Overall, the improvements we managed to make to `DbLogger` make it a worthy tradeoff.

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

1.5 Benefits of functional programming

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

There are many more benefits to be reaped by applying functional techniques, and they roughly fall into 3 categories.

Cleaner code: apart from the already-mentioned conciseness, FP leads to more expressive, readable, and more easily testable code, as you will see in coming chapters. Clean code is not just the developer's intellectual pleasure, but also leads to huge economic benefit for the business through reduced cost for maintenance.

Better support for concurrency: several factors, from multi-core CPUs to distributed system, bring a high degree of concurrency to our programs. Concurrency is traditionally associated with difficult problems such as deadlocks, but FP offers techniques that prevent the possibility of these problems from even occurring. We'll see an introductory example in Chapter 2 and more advanced examples towards the end of the book.

Note

It is not by chance that sexy new frameworks like Reactive Extensions or Akka.NET are steeped in FP. Learning the core concepts of FP in this book will enable you to better understand and leverage these frameworks.

A multi-paradigm approach: they say that if the only tool you know is a hammer, every problem will look like a nail of some sort. Conversely, the more angles you have from which to view a given problem, the more likely it is that you will find an optimal solution. If you are already proficient in OOP, learning a different paradigm such as FP will inevitably give you a richer perspective. When faced with a problem, you will be able to consider several approaches, and pick the most effective.

1.6 Summary

- FP is a powerful paradigm that can help you to 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:

  1. functions as 1st-class values
  2. avoiding in-place updates

- Functions in C# can be represented with methods, delegates and lambdas
- Compared to mathematical functions, programming functions are more difficult to reason about, since their output may depend on variables other than their input arguments
- FP leverages higher-order functions (functions that take other functions as input or output), hence the necessity for the language to have functions as 1st-class values

1.7 Exercises

1. Browse through the methods of `System.Linq.Enumerable`. Which are HOFs? Which do you think imply iterated application of the given function?
2. Write a method that uses quicksort to sort a `List<int>` (return a new list, rather than sorting it in place).
3. Generalize your implementation to take a `List<T>`, and additionally a `Comparison<T>` delegate.
4. 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 first parameter, instead of the `IDisposable`. (This can be used to fix warnings given by some code analysis tools about instantiating an `IDisposable` and not disposing it.)