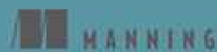
Tomas Petricek
with Jon Skeet

Functional Programming for the Real World

With examples in F# and C#



©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Table of Contents

Part I. Introduction
Chapter 1 Thinking differently about problems
Chapter 2 Functional concepts and programming languages
Chapter 3: Meet tuples, lists and functions in F# and C#
Chapter 4: Exploring F# and .NET libraries by example

Part II. Core functional techniques
Chapter 5: Creating and using common functional values
Chapter 6: Working with values using high-order functions
Chapter 7: Designing data-centric programs
Chapter 8: Designing behavior-centric programs

Part III. Advanced F# programming techniques
Chapter 9: Turning values into F# object types with members
Chapter 10: Efficiency of data structures, tail-recursion and continuations
Chapter 11: Refactoring functional programs and using lazy values
Chapter 12: Sequence expressions and non-standard computations

Part IV. Applied functional programming
Chapter 13: Obtaining data asynchronously and exploring them
Chapter 14: Writing parallel functional programs
Chapter 15: Creating domain specific language for animations
Chapter 16: Developing reactive functional programs

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

1
Thinking differently about problems

Functional programming is a paradigm that originated from ideas older than the first computers. The first functional programming language celebrated its 50th birthday in 2008. Functional languages are very succinct and expressible, yet everything is achieved using a minimal number of concepts. Despite their elegance, functional languages have largely been ignored by mainstream developers--until now.

Today we are facing new challenges and trends that open the door to functional languages. There has never been a better time to learn them. We need to write programs that process large sets of data and scale to a large number of processors or computers. We want to write programs that can be easily tested. We want to be able to express our logic in a declarative way which expresses *results* without explicitly specifying execution details–making the code easier to understand and reason about. All of these trends are embodied in functional programming, and we'll look at each of them later in this chapter.

As a result, many mainstream languages now include some functional features. In the .NET world, generics in C# 2.0 were heavily influenced by functional languages, anonymous methods in C# 2.0 and lambda expressions in C# 3.0 are examples of the most fundamental concept in functional programming and the whole of LINQ is rooted in a declarative, functional approach.

While the conventional languages are playing catch-up, truly functional languages have been receiving more attention too. The most significant example of this is probably F#, which is will be an official, fully supported Visual Studio language as of Visual Studio 2010. This evolution of functional languages on .NET is largely possible thanks to the common language runtime (CLR), which makes it possible to mix multiple languages when developing a single .NET application and also to access rich .NET libraries from new languages like F#. This makes it much easier to learn these new languages, as all of the platform knowledge that you've accumulated during your career can still be used in the new context of a functional language.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

In this book, we'll look at the most important functional programming concepts and we'll demonstrate them using real-world examples from .NET. We'll start with the description of the ideas and then turn to the aspects that make it possible to develop large scale real-world .NET applications in a functional way. We'll use both F# and C# 3.0 in this book, because many of these ideas are directly applicable to C# programming. You certainly don't need to write in a functional language to use functional concepts and patterns. However, seeing the example in F# gives you a deeper understanding of how it works and F# often makes it easier to express and implement the solution.

We'll start this chapter by looking at the functional concepts that make you more productive, and then explore several examples that demonstrate what those ideas look like in real source code. We won't go into any details in this chapter, however–the goal is just to show you an interesting and elegant example that we'll discuss more fully later in the book.

1.1 Being productive with functional programming

Many people find functional programming more elegant or even beautiful, but that's hardly a good reason to use it in a commercial environment. Elegance doesn't pay the bills, sadly. The key reason why for coding in a functional style is that it makes you and your team more productive.

In this section, we'll look at the key benefits that functional programming gives you and how it solves some of the most important problems of modern software development. We'll start by looking at the declarative programming style, which gives us a richer vocabulary for describing our intentions.

1.1.1 Declarative programming style

When writing a program, we have to explain our goals to the computer using the vocabulary that it understands. In imperative languages, this consists of commands. We can add new commands, such as "show customer details", but the whole program is a step by step description saying how the computer should accomplish the overall task. An example of a program is "Take the next customer from a list. If the customer lives in UK, show their details. If there are more customers in the list, go to the beginning."

Once the program grows, the number of commands in our vocabulary becomes too high, making it very difficult to use. This is where object-oriented programming makes our life easier, because it allows us to organize our commands in a better way. We can associate all commands that involve customer with some customer entity (a class), which makes the description a lot clearer. However, the program is still a sequence of commands specifying how it should proceed.

Functional programming provides a completely different way of extending the vocabulary. We're not limited to adding new primitive commands; we can also add new control structures–primitives that specify how we can put commands together to create a program. In imperative languages, we were able to compose commands in a sequence or

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

using a limited number of built in constructs such as loops, but if you look at typical programs, you'll still see many recurring structures; common ways of combining commands.

In our example we can see a pattern (or a control structure), which could be expressed as "Run the first command for every customer for which the second command returns true." Using this primitive, we can express our program simply by saying "Show customer details of every customer living in UK." In this sentence the part "living in UK" specifies the second command and the part "show customer details" represents the first command.

SAYING "WHAT" RATHER THAN "HOW"

If you compare these two sentences, you can see that the first describes exactly *how* to achieve our goal while the second describes *what* we want to achieve. This is the essential difference between imperative and declarative styles of programming. Hopefully you'll agree that the second sentence is far more readable and better reflected the aim of our "program".

So far I've just been using an analogy, but we'll see how this idea maps to actual source code later in this chapter. However, this isn't the only aspect of functional programming that makes life easier. In the next section, we'll look at another concept that makes it much easier to understand what a program does.

1.1.2 Understanding what a program does

In the usual imperative style, the program consists of objects that have some internal state that can be changed either directly or by calling some method of the object. This means that when we call a method, it can be hard to tell what state is affected by the operation. For example, in the C# snippet in listing 1.1 we create an ellipse, get its bounding box and then call a method on the returned rectangle. Finally, we return the ellipse to whatever has called us.

Listing 1.1 Working with ellipse and rectangle (C#)

```
Ellipse el = new Ellipse(new Rectangle(0, 0, 100, 100));
Rectangle rc = el.BoundingBox;
rc.Inflate(10, 10);                                          #1
return el;
```

#1 Is the original ellipse changed here?

How do we know what the state of the ellipse `el` will be after the code runs, just by looking at it? This is really hard, because `rc` could be a reference to the bounding box of the ellipse and `Inflate` (#1) could modify the rectangle, changing the ellipse at the same time. Or maybe the `Rectangle` type is a value type (declared using the `struct` keyword in C#) and it's copied when we assign it to a variable. Perhaps the `Inflate` method doesn't actually modify the rectangle at all, and returns a new rectangle as a result, so the third line has no effect at all.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

In functional programming, most of the data structures are immutable, which means that we cannot modify them. Once the `Ellipse` or `Rectangle` is created, we can't change it. The only thing we can do is to create a new `Ellipse` with a new bounding box. This makes it easy to understand what a program does. In listing 1.2 you can see how we could rewrite the previous snippet if `Ellipse` and `Rectangle` were immutable. As you'll see, understanding the program's behavior becomes much easier.

Listing 1.2. Working with immutable ellipse and rectangle (C#)

```
Ellipse el = new Ellipse(new Rectangle(0, 0, 100, 100));
Rectangle rc = el.BoundingBox;
Rectangle rcNew = rc.Inflate(10, 10);                          #1
return new Ellipse(rcNew);                                     #2
```

#1 Returns a new rectangle

#2 Return a new ellipse with the new bounding box

When writing program using immutable types, the only thing a method can do is to return a result. It cannot modify state of any objects. You can see that for example `Inflate` returns a new rectangle as a result (#1) and that we construct a new ellipse to return an ellipse with a modified bounding box (#2). This may feel a bit unfamiliar for the first time, but keep in mind that this isn't a new idea to .NET developers. `String` is probably the best known immutable type in the .NET world, but there are many examples such as `DateTime` and other value types.

Functional programming takes this idea further, which makes it a lot easier to see what a program does, because the result of method gives us full specification of what the method does. We'll talk about immutability in a more detail later, but let's first look at one area where it is extremely useful: implementing multi-threaded applications.

1.1.3 Concurrency-friendly application design

When writing a multi-threaded application using the traditional imperative style we have to face two problems. First of all, it is difficult to turn existing sequential code into parallel code, because we have to modify large portions of the code-base to use threads explicitly. The second problem is that using shared state and locks is difficult. You have to carefully consider how to use locks to avoid race conditions and deadlocks, but leave enough space for parallel execution. Functional programming gives us answers to these two problems:

- A declarative programming style makes it easier to introduce parallelism into existing code. We can just replace a few primitives that specify how to combine commands with a version that executes commands in parallel.
- Thanks to the immutability, we cannot introduce race conditions and we can write lock-free code. This style makes it easy to see which parts of the program are independent and we can easily modify the program to run those tasks in parallel.

These two aspects influence how we design our applications and as a result make it a lot easier to write code that executes in parallel, taking full advantage of the power of multi-core

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

machines. This isn't the only change you should expect to see in your design when you start thinking functionally, either…

1.1.4 Elegant thought leads to elegant code

The functional programming paradigm no doubt influences how you design and implement applications. This doesn't mean that you have to throw away anything from your existing knowledge, because many of the programming principles that you're using today are applicable to functional applications as well. This is true especially at the design level in the way how you structure the application.

On the other hand, functional programming can cause a radical transformation of how you approach problems at the implementation level. However, when learning how to use functional programming ideas, you don't have to make any radical steps. In C# you just learn how to efficiently use the new features. In F#, you can often use direct equivalents of C# constructs while you're still getting your feet wet. As you become a more experienced functional developer, you'll learn more efficient and concise ways to express yourself.

The following list summarizes how functional programming influences your programming style, working down from a design level to actual implementation.

- Functional programs on .NET still use object-oriented design as a great way for structuring applications and components. Larger number of types and classes are designed as immutable, but it is still possible to create standard classes especially when collaborating with other .NET libraries.
- Thanks to functional programming, you can simplify many of the standard OO design patterns, because some of them correspond to language features in F# or C# 3.0. Also, some of the design patterns simply aren't needed any more when the code is implemented in the functional way. We'll see many examples of this throughout the book, especially in chapters 7 and 8.
- Perhaps the larger influence of functional programming is at the lowest level. Thanks to the combination of a declarative style, succinct syntax and type inference, functional languages make it easier to concisely express algorithms in a more readable way.

We'll talk about all of these aspects later in the book - but building up from the lowest level. We'll start with the functional values used to implement methods and functions, before raising our sights to design and architecture. We'll see new patterns that are specific to functional programming, as well as looking at how the object-oriented patterns you're already familiar with either fit in with the functional world or are no longer required. The functional world from the previous sentence isn't a strictly delimited technology, because the functional ideas can appear in different forms.

1.1.5 The functional paradigm

Functional programming is a programming paradigm. This means that it defines the concepts that we can use when thinking about problems. However, it doesn't precisely specify how

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

exactly these concepts should be represented in the programming language. As a result, there are many functional languages and they put more emphasis on different features and aspects of the functional style.

We can use an analogy with a paradigm you're already familiar with: object-oriented programming. In the object-oriented style, we think about problems in terms of objects. Each object-oriented language has some notion of what an object is, but the details vary between languages. For instance C++ has multiple inheritances and JavaScript has prototypes. Moreover, you can still use an object-oriented style in language which isn't object-oriented such as C. It is less comfortable, but you'll still get some of the benefits.

However, programming paradigms are not exclusive. The C# language is primarily object-oriented, but in the 3.0 version it supports several functional features, so we can use some techniques from the functional style directly. On the other side, F# is primarily a functional language, but it fully supports the .NET object model. The great thing about combining paradigms is that we can choose the approach that best suits the problem.

Finally, learning the functional paradigm is worthwhile even if you're not planning to use a functional language. By learning a functional style, you'll gain concepts that make it easier to think about and solve your daily programming problems. Interestingly, many of the standard object-oriented patterns describe how to encode some clear functional concept in the object-oriented programming style.

So far, we have only talked about functional programming in a very general sense. It's important to have some broad idea about what makes functional programming different and why it's worth learning, but there's nothing like seeing actual code to bring things into focus. In the next section, we'll take a quick look at a couple of more specific examples.

1.2 Functional programming by example

The goal of the upcoming few examples is to show you that functional programming isn't by any means a theoretical discipline. Instead, you'll see that you've already seen and maybe even used some functional ideas. Reading about functional programming will help you to understand these technologies at a deeper level and use them more efficiently. We'll also look at a couple of examples from later parts of the book that show important practical benefits of the functional style. In the first set of examples, we'll look at declarative programming.

1.2.1 Expressing intentions using declarative style

In the previous section, I described how a declarative coding style makes you more productive. Programming languages that support a declarative style allow us to add new ways of composing basic constructs. When using this style, we're not limited to basic sequences of statements or built-in loops, so the resulting code describes more "what" the computer should do rather than "how" to do it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

I'm talking about this style in a general way because the idea is universal and not tied to any specific technology. However, it's best to demonstrate it using a few examples that you may know already to show how it's applied in specific technologies. In the first two examples, we'll look at the declarative style of LINQ and XAML. If you don't know these technologies, don't worry. The examples are simple enough to understand without background knowledge. In fact, the ease of understanding code–even in an unfamiliar context–is one of the principal benefits of a declarative style!

WORKING WITH DATA IN LINQ

If you're already using LINQ then this example will be just a reminder. However, I'll use it to show something more important. Let's first look at an example of code that works with data using the standard imperative programming style.

Listing 1.3 Imperative data processing (C#)

```
List<string> res = new List<string>();                          #1
foreach(Product p in Products) {                                #2
   if (p.UnitPrice > 75.0M) {
      res.Add(String.Format("{0} - ${1}",
         p.ProductName, p.UnitPrice));                          #3
   }
}
return res;
```

#1 Create resulting list
#2 Iterate over products
#3 Add information to list of results

You'll probably need to read the code carefully to understand what it does, but that's not the only aspect we want to improve. The code is written as a sequence of some basic imperative commands. For example, the first statement creates new list (#1), the second iterates over all products (#2) and a later one adds element to the list (#3). However, we'd like to be able to describe the problem at a higher level. In more abstract terms, the code just filters a collection and returns some information about every returned product.

In C# 3.0, we can write the same code using query expression syntax. This version is closer to our real goal–it uses the same idea of filtering and transforming the data. You can see the code in listing 1.4.

Listing 1.4 Declarative data processing (C#)

```
var res = from p in Products
          where p.UnitPrice > 75.0M                             #1
          select string.Format("{0} - ${1}",
             p.ProductName, p.UnitPrice);                       #2
return res;
```

#1 Filter products using predicate
#2 Return information about product

The expression that calculates the result (`res`) is composed from basic operators such as `where` or `select`. These operators take other expressions as an argument, because they need to know exactly what we want to filter or select as a result. Using the previous

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

analogy, these operators give us a new way for combining pieces of code to express our intention with less writing. It is worth noting that the whole calculation in the listing 1.3 is written just as a single expression that describes the result rather than a sequence of statements that constructs it. You'll see this become a trend repeated throughout the book. In more declarative languages such as F#, everything you write is an expression.

Another interesting aspect is that many technical details of the solution are now moved to the implementation of the basic operators. This makes the code simpler, but also more flexible, because we can easily change implementation of these operators without making larger changes to the code that uses them. As we'll see later, this makes it much easier to parallelize code that works with data. However, LINQ is not the only mainstream .NET technology that relies on declarative programming. Let's turn our attention to Windows Presentation Foundation and the XAML language.

DESCRIBING USER INTERFACES IN XAML

Windows Presentation Foundation is a .NET library for creating user interfaces that supports the declarative programming style. It separates the part that describes the user interface from the part that implements the imperative program logic. However, the best practice in WPF is to minimize the program logic and create as much as possible in the declarative way.

The declarative description is represented as a tree-like structure created from objects that represent individual GUI elements. It can be created in C#, but WPF also provides a more comfortable way using an XML based language called XAML. Nevertheless, we'll see that there are many similarities between XAML and LINQ. The listing 1.5 shows how the code in XAML compares with code that implements the same functionality using the imperative Windows Forms library.

Listing 1.5 Creating user interface using imperative and declarative style (C#)

```
<Canvas Background="Black">
   <Ellipse x:Name="el"
      Width="75"
Height="75"
      Canvas.Left="0"
      Canvas.Top="0"
      Fill="LightGreen" />
</Canvas>
```

```
protected override void OnPaint
      (PaintEventArgs e) {
   Graphics gr = e.Graphics;
   Brush lg = Brushes.LightGreen;
   Brush bl = Brushes.Black;
   gr.FillRectangle(bl,
ClientRectangle);
   gr.FillEllipse(lg, 0, 0, 75, 75);
}
```

It isn't difficult to identify what makes the code on the left side more declarative. The XAML code describes the user interface by composing various primitives and specifying their properties. The whole code is a single expression that creates a black canvas containing a green ellipse. On the other hand, the imperative version specifies how to create the user interface. It is a sequence of statements that specify what drawing operations should be executed to get the required GUI. This example clearly demonstrates the difference between saying "what" using the declarative style and saying "how" in the imperative style.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Also, in the declarative version we don't need as much knowledge about the underlying technical details. If you just look at the code, you don't really need to know how WPF will represent and draw the GUI. On the other hand, when looking at the WinForms example, all the technical details such as representation of brushes and order of the drawing are visible in the code. In the example above, the correspondence between XAML and the drawing code was quite clear, but we can use XAML with WPF to describe more complicated runtime aspects of the program. Let's look at the following example:

```
<DoubleAnimation
   Storyboard.TargetName="el"
   Storyboard.TargetProperty="(Canvas.Left)"
   From="0.0" To="100.0" Duration="0:0:5" />
```

This single expression creates an animation that changes the `Left` property of the ellipse (specified by the name `el`) from value 0 to value 100 in 5 seconds. The code is implemented using XAML, but we could as well write it by constructing the object tree explicitly in C#. Under the hood, `DoubleAnimation` is a class, so we would just specify its properties. The XAML language adds a more declarative syntax for writing the specification. In either case, the code would be declarative thanks to the nature of WPF. On the other hand, the traditional imperative version of code that implements an animation would be rather complex. It would have to create some timer, register an event handler that would be called every couple of milliseconds and it would have to calculate new location of the ellipse.

DECLARATIVE CODING IN .NET

WPF and LINQ are two main-stream technologies that use a declarative style, but there are many others. The goal of LINQ is to simplify working with data in a general-purpose language. It draws on ideas from many data manipulating languages that use the declarative style, so you can find the declarative approach for example in SQL or XSLT.

Another area where the declarative style is used in C# or VB.NET is when using .NET attributes. Attributes give us a way to annotate a class or its members and specify how they can be used in specific scenarios, such as editing a GUI control in a designer. This is declarative, because we just specify what we expect from the designer when working with the control and we don't have to write the code that would imperatively configure the designer.

So far we've seen several technologies that are based on the declarative style and how they make problems easier to solve. However, you may be asking yourself how we use it for solving our own kinds of problems. In the next section we'll take a brief look at an example from chapter 15 that demonstrates this.

DECLARATIVE FUNCTIONAL ANIMATIONS

Functional programming gives you the ability to write your own library that allows you to solve problems in the declarative style. We've seen how LINQ does that for data

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

manipulation and how WPF does that for user interfaces, but in functional programming, we'll often create libraries for our own problem domain.

When I earlier mentioned that declarative style makes it possible to ignore implementation details, I wasn't really saying the full truth. When we're designing our own declarative library, we of course need to implement all the technical details. However, the great thing about the functional style is that it allows us to hide the implementation from developers (just like LINQ does) and makes it possible to solve the general problem once and for all.

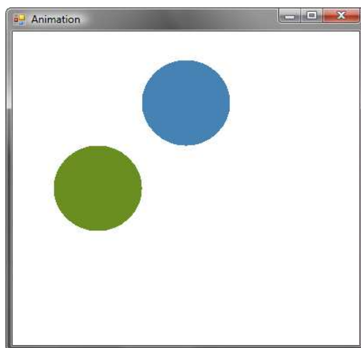
The listing 1.6 shows a code that uses a declarative library for creating animations that we'll develop in chapter 15. You don't have to fully understand the code to see the benefits that we get thanks to the declarative style. It is similar to WPF in a sense that it describes how the animation should look rather than how to draw it using a timer.

Listing 1.6 Creating functional animation (C#)

```
var green = Anims.Circle(Brushes.OliveDrab, 100.0f.Anim());          #A
var blue  = Anims.Circle(Brushes.SteelBlue, 100.0f.Anim());          #A

var animatedPos = Time.Wiggle * 100.0f.Anim();                       #1

var greenMove = green.MoveXY(animatedPos, 0.0f.Const());             #B
var blueMove = blue.MoveXY(0.0f.Const(), animatedPos);               #B

var animation = Anims.Compose(greenMove, blueMove);                  #C
```

#A Create green and blue ellipse
#1 Value animated from -100 to +100
#B Animate X or Y coordinates of ellipses
#C Compose animation from both ellipses

We'll explain everything in detail later in chapter 15. However, you can probably guess that the animation creates two ellipses. Later, it creates animated ellipses and composes them into an animation (represented as `animation` value). If we render this animation to a form, we get a result that is displayed in figure 1.1.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Figure 1.1 The green ellipse is moving from the left to the right and the blue ellipse is moving from the top to the bottom.

The entire declarative description is based on animated values. There is a primitive animated value called `Time.Wiggle`, which has a value that swings between -1 and +1. Another primitive construct is `x.Anim()` creates an animated value that has always the same value. If we multiply `Wiggle` by 100, we'll get an animated value that ranges between -100 and +100 (#1). These animated values can be used for specifying animations of graphical objects such as our two ellipses. The screenshot shows them in a state where X coordinate of the green one and Y coordinate of the blue one are close to the -100 state.

In the code we wrote, we don't need to know anything about the representation of animated values, because we're describing the whole animation just by calculating with the primitive animated value. Another aspect of the declarative style that you can see in the code is that the animation is in principle described using a single expression. We made it more readable by declaring several local variables, but if you replaced occurrence of the variable with its initialization code, the animation would remain exactly the same.

COMPOSITIONALITY

An important feature of declarative libraries is that we can use them in a compositional manner. In LINQ, you can move a part of a complex query into a separate query and reuse it. Similarly, our previous sample is very compositional. We can declare animated values such as `animatedPos` and compose primitive animated objects using `Anim.Compose`.

On the last couple of pages, we looked at the declarative programming, which is an essential aspect of the functional style. The last example shows how this style can be used in an advanced library for describing animations. In the next section, we'll turn our attention to more technical, but also very interesting functional aspect which is immutability.

1.2.2 Understanding code using immutability

We discussed immutability before when talking about benefits of the functional style. We used an example with bounding box of an ellipse, where it wasn't clear how the code behaved. Once we rewrote the code using immutable objects, it became easier to understand. We'll talk about this topic in detail in later chapters. The purpose of this example is just to satisfy your curiosity and show how an immutable object would look in practice.

Again, don't worry if you won't understand everything in detail, because we'll talk about everything more fully later. Now, let's imagine we're writing a game with some characters that we can shoot at. Listing 1.7 shows a part of the class that represents the character.

Listing 1.7 Immutable representation of a game character (C#)

```
class GameCharacter {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

```
    readonly int health;                                          #1
    readonly Point location;                                      #1

    public Character(int health, Point location) {
       this.health = health;                                      #2
       this.location = location;                                  #2
    }
    public Character HitByShooting(Point target) {
       int newHealth = CalculateHealth(target);
       return new GameCharacter(newHealth, this.location);        #3
    }
    public bool IsAlive {
       get { return health > 0; }
    }
    // Other methods and properties omitted
}
```

#1 All fields are declared as readonly
#2 Initialize immutable fields only once
#3 Return a game character with updated health

In C#, we can explicitly mark a field as immutable using the `readonly` keyword. This means that we cannot change the value of the field, but we could still modify the target object if the field is a reference to a mutable class. When creating a truly immutable class, we need to make sure that all fields are marked as `readonly` and also that the types of these fields are also primitive types, immutable value types or other immutable classes.

According to these conditions, our `GameCharacter` class is immutable. All its fields are marked using the `readonly` modifier (#1), `int` is a primitive type and `Point` is an immutable value type. When a field is read-only it can be set only when creating the object, so we can only set the health and location of the character only in the constructor (#2). This means that we can't modify the state of the object once it is initialized. So, what can we do when an operation needs to modify the state of the game character?

You can see the answer when you look at the `HitByShooting` method (#3). It implements a reaction to a shot being fired in the game. It uses the `CalculateHealth` method (not shown in the sample) to calculate the new health of the character. In an imperative style, it would then update the state of the character, but that's not possible since the type is immutable. Instead, the method creates a new `GameCharacter` instance to represent the modified character and returns it as a result.

The class from the previous example represents a typical design of immutable C# classes and we'll use it (with minor modifications) throughout the book. Now that we know what immutable types look like, let's see some of the consequences.

READING FUNCTIONAL PROGRAMS

We've already seen an example that used immutable types when looking at the code with bounding box of an ellipse. However, that was very briefly and we just concluded that it makes the code more readable. In this section, we're going to look at two snippets that we could find somewhere in our functional game.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Listing 1.8 shows two separate examples, each with two game characters: `player` and `monster`. The first one shows how we could execute the monster AI to perform a single step and then test whether the player is in danger and the second shows how we could handle a gunshot.

Listing 1.8 Code snippets form a functional game (C#)

```
// Move the monster & test if the player is in danger
var movedMonster = monster.PerformStep();                    #1
var inDanger = player.IsCloseTo(movedMonster);               #2

// Did gunshot hit a monster or the player?
var hitMonster = monster.HitByShooting(gunShot);             #3
var hitPlayer = player.HitByShooting(gunShot);               #3
```

#1 Move the monster
#2 Test distance from the moved monster
#3 Create new monster and player

All objects in our functional game are immutable, so when we call method on an object, it cannot modify itself or any other object. If we know that, we can make several interesting observations about the previous examples. In the first snippet, we start by calling the `PerformStep` method of the monster (#1). The method returns a new monster and we assign it to a variable called `movedMonster`. On the next line, we use this monster to check whether the player is close to it and so is in danger.

One interesting point to note here is that we can see that the second line of the code relies on the first one. If we changed the order of these two lines, the program wouldn't compile because `movedMonster` wouldn't be declared on the first line. On the other hand, if you implemented this in the imperative style, the method would modify the state of the `monster` object. In that case, we could rearrange the lines and the code would compile, but it would change the meaning of the program and it could start behaving incorrectly.

Now, what can we learn by looking at the second snippet? It consists of two lines that create a new monster and a new player objects with updated health property when a shooting occurs in the game. The two lines are independent, meaning that we could change their order. Can this operation change the meaning of the program? It appears that it shouldn't and when all objects are immutable it doesn't. Surprisingly, it might change the meaning in the imperative version if `gunShot` were mutable. The first of those objects could change some property of the gunshot and the behavior would depend on the order of these two statements.

The previous example was quite simple, but it already shows how immutability eliminates many possible difficulties. In the next section, we'll see another great example, but let me just briefly review what you'll find later in the book.

REFACTORING AND UNIT TESTING

We've already seen that immutability helps us to understand what a program does. This is very helpful when refactoring the code. Another interesting functional refactoring is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

changing when some code actually executes. It may run when the program hits it for the first time, but it may as well execute when its result is actually needed. As we'll see in a few pages, this way of evolving programs is very important in F# and immutability makes refactoring easier in C# too. We'll talk about refactoring later in chapter 11.

Another area where immutability helps a lot is when creating unit tests for functional programs. The only thing that a method can do in an immutable world is to return a result, so we only have to test whether a method returns the right result for specified arguments. You'll find more information about this topic in chapter 18.

When discussing how functional programming makes you more productive, I mentioned immutability as an important aspect that makes it easier to write parallel programs. In the next section we'll briefly look at that and also at other related topics.

1.2.3 Writing efficient parallel and asynchronous programs

I said earlier that functional programming makes it easier to write parallel programs. This is one of the most important aspects of this paradigm nowadays and maybe it is also the reason why you picked this book. In this section, we'll look at a couple of samples demonstrating how functional programs can be easily parallelized. In the first two examples, we'll use Parallel Extensions to .NET. This is a new technology from Microsoft for writing parallel applications, shipping as part of .NET 4.0. As you might expect, it lends itself extremely well to functional code. As always in this chapter, we won't go into the details. I just want to demonstrate that parallelizing functional programs is significantly easier and more importantly, less error prone than it is for the imperative code.

PARALLELIZING IMMUTABLE PROGRAMS

First we'll take another look at the previous example. We've seen two snippets from a game written in a functional way. In the first snippet, the second line uses the outcome of the first line (state of the monster after movement). Thanks to the use of immutable classes, we can see that this doesn't give us any space for introducing parallelism.

On the other hand, the second snippet consists of two independent lines of code. I said earlier that in functional programming, we can run independent parts of the program in parallel. Now you can see that immutability gives us a great way to spot which parts of the program are independent. Even without knowing any details, we can look at the change that makes these two operations run in parallel. The change to the source code is minimal:

```
var hitMonster = Future.Create(() =>
   monster.HitByShooting(gunShot));
var hitPlayer = Future.Create(() =>
   player.HitByShooting(gunShot));
```

The only thing that we did is that we wrapped the computation in a `Future` type from the Parallel Extensions library. We'll talk about `Future` in detail in chapter 14. Interestingly, the benefit isn't only that we have to write less code, but also that we have a guarantee that

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

the code is correct. If you did a similar change in an imperative program, you'd have to carefully review the `HitByShooting` method (and any other method it calls) to find all places where it accesses some mutable state and add locks to protect the code that modifies shared state. In functional programming everything is immutable, so we don't need to add any locks.

The example in this section is a form of lower a level *task based parallelism*, which is one of three approaches that we'll see in chapter 14. In the next section we'll take a brief look at the second approach, which benefits from the declarative programming style.

DECLARATIVE PARALLELISM USING PLINQ

Declarative programming style gives us another great technique for writing parallel programs. I have already stated that the code written using the declarative style is composed using primitives. In LINQ, these primitives are query operators such as `where` and `select`. In the declarative style, we can easily replace the implementation of these primitives and that's exactly what PLINQ does. It allows us to replace standard query operators with query operators that run in parallel.

In the listing 1.9, you can see a query that updates all monsters in our fictive game and remove those that died in the last step of the game. The change is extremely simple, so I can show you both of the versions in a single listing.

Listing 1.9 Parallelizing data processing code using PLINQ (C#)

```
var updated =                          var updated =
   from m in monsters                     from m in monsters.AsParallel()    #1
   let nm = m.PerformStep()               let nm = m.PerformStep()
   where nm.IsAlive select nm;            where nm.IsAlive select nm;
```

The only change that we made in the parallel version on the right side is that we added a call to `AsParallel` method (#1). This call changes the primitives that are used when running the query and makes the whole fragment run in parallel. We'll see how this works in chapter 11, where we'll talk about declarative computations like this in general and in chapter 14 which focuses on parallel programming specifically.

You may have already seen this demo and you were perhaps thinking that you don't use LINQ queries that often in your programs. This is definitely a valid point, because in imperative programs, LINQ queries are used less frequently. However, functional programs do most of their data processing in the declarative style. In C#, this can be written using query expressions whereas F# provides higher order list processing functions that we'll see in chapters 5 and 6. This means that after you'll read this book, you'll be able to use declarative programming more often when working with data. As a result, your programs will be more easily parallelizable. The technique I just described also inspired an algorithm used internally by Google for massive parallel data processing.

Microsoft PLINQ and Google MapReduce

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Google has developed a framework called MapReduce [Dean, Ghemawat, 2004] for processing of massive amounts of data in parallel. This framework distributes the work between computers in large clusters and uses exactly the same ideas as PLINQ. The basic idea of MapReduce is that the user program describes the algorithm using two operations (somewhat similar to `where` and `select` in PLINQ). The framework takes these two operations and the input data, and runs the computation. You can see a diagram visualizing the computation in figure 1.2.

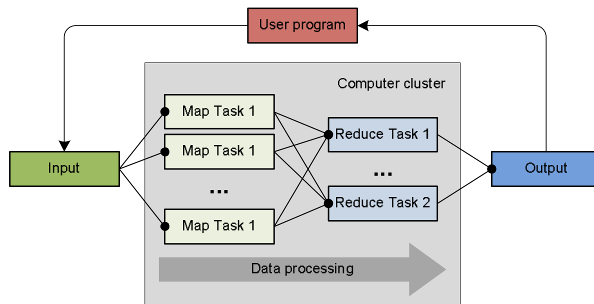


Figure 1.2 In the MapReduce framework an algorithm is described by specifying map task and a reduce task. The framework automatically distributes the input across servers and processes the tasks in parallel

The framework splits the input data into partitions and executes the map task (using the first operation from the user) on each of the partitions. For example, a map task may find the most important keywords in a web page. The results returned by map tasks are then collected and grouped by a specified key (for example the name of the domain) and the reduce task is executed for each of the groups. In our example, the reduce task may summarize the most important keywords for every domain.

We've briefly seen two ways in which functional programming makes parallelization simpler. However, there is one more related area where functional programming helps us to write more efficient and scalable code with respect to multi-threading. It is important especially when the code uses long running I/O operations.

WRITING NON-BLOCKING CODE USING F#

Long running operations are quite frequent in modern software. Many applications use HTTP requests to load some data from the internet or communicate using web services. When an application performs an operation like this, it is very hard to predict when the operation will complete, and if this is not handled properly the application will become unresponsive.

However, writing the code that performs I/O operations without blocking is very difficult using the current techniques. In F#, this is largely simplified thanks to a feature called *asynchronous workflows*. Interestingly, this is one of the F# features that are really hard to implement in C#, so it's a good reason for looking at F#. We'll talk about asynchronous

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

workflows in detail later in chapter 13, but I can show you at least a brief example to demonstrate how interesting this feature is. Let's start by looking at listing 1.10, which shows a C# example that downloads source of a web page.

Listing 1.10 Downloading web pages (C#)

```
var req = HttpWebRequest.Create("http://manning.com");
var resp = req.GetResponse();                                  #1
var stream = resp.GetResponseStream();
var reader = new StreamReader(stream);
var html = reader.ReadToEnd();                                 #2
Console.WriteLine(html);
```

#1 Initialize HTTP connection
#2 Download the web page content

The listing shows a fairly simple code that downloads HTML source code of a specified web page. You'd also have to add some `using` directives to reference the necessary .NET namespaces if you wanted to compile the code, but we'll show this properly in later chapters. The program needs to perform HTTP communication in two places. In the first (#1) it needs to initialize HTTP connection with the server and in the second (#2) it downloads the web page.

Both of these operations could potentially take quite a long time and each of them could block the active thread, causing our application to become unresponsive. We could run the download on a separate thread, but using threads is expensive, so this would limit the number of downloads we can run in parallel. Also, most of the time, the thread would be just waiting for the response, so we'd be consuming thread resources for no good reason. To implement this properly, we need to use asynchronous .NET methods that allow us to trigger the request and call some code that we provide when the operation completes. This version of code is quite difficult to write. Even if we use anonymous delegates from C# 2.0, the code still looks quite complicated:

```
var req = HttpWebRequest.Create("http://manning.com");
req.BeginGetResponse(delegate(IAsyncResult ar) {
   var rsp = req.EndGetResponse(ar);
   // TODO: Use the response to read the HTML
});
```

Anonymous delegates or lambda expressions make this a bit nicer, because we don't have to write a method to handle the response, but we still have to change the structure of the code. In fact, if we decide to change a synchronous version of the code into asynchronous, we'll have to rewrite it almost completely.

The previous snippet isn't complete, but if we tried to finish it, we'd find another issue. There is no `BeginReadToEnd` method, so we'd have to implement this functionality ourselves. This is quite difficult, because we need to download the page in a buffered way. If we want to write this in an asynchronous style, we can't use any of the built-in constructs such as `while` loop.

In F#, it's common to start with the simplest possible solution to a problem and then turn it into a more sophisticated version. We'll talk about this principle later in this chapter,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

but we can see it in action right now. One of the things you may want to do is to take synchronous code that downloads a web page and make it asynchronous. When working with .NET libraries, the F# code is quite similar to C#, so you can just imagine that the listing 1.10 was in F# (you'd just delete semicolons and change the `var` keyword to `let`). The listing 1.11 shows an asynchronous version using F# asynchronous workflows.

Listing 1.11 Downloading web page asynchronously (F#)

```
let op =
   async {                                                       #1
      let req = HttpWebRequest.Create("http://manning.com")
      let! resp = req.AsyncGetResponse()                         #2
      let stream = resp.GetResponseStream()
      let reader = new StreamReader(stream)
      let! html = reader.AsyncReadToEnd()                        #A
      Console.WriteLine(html)
   }
Async.Run(op)                                                    #B
```

#1 Wrap in an asynchronous workflow
#2 Run operation asynchronously
#A Asynchronous download
#B Run the workflow

The process of turning a synchronous code into asynchronous in F# is quite easy. First of all, we wrap the whole computation into an `async` block (#1). The next thing to do is to identify all asynchronous operations in the block and to change the method to a corresponding asynchronous version. The workflow needs to know which of the methods should be executed in a non-blocking way, so we also change the usual value declaration using `let` into a workflow-specific declaration that uses `let!` syntax (#2). What is even more interesting is that methods like `AsyncReadToEnd` are quite easy to implement, because asynchronous workflows can use `while` loops and other basic constructs[1].

This feature is very easy to use but it isn't easy to see how the code actually executes at first glance. We'll explain everything in detail in chapter 13, but it's worth noting that asynchronous workflows aren't a built-in feature of the F# language. It is just a very useful instance of a more general feature that allows you to write non-standard computations. This feature is also covered in this book and we'll talk about it in chapter 12.

---

[1] There are projects that attempt to simplify this problem in C# such as the Concurrency and Coordination Runtime (CCR), but all of them rely on using some C# language features in an unexpected and slightly unnatural ways. We'll mention a couple of these projects briefly when discussing asynchronous workflows in chapter 13.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Asynchronous workflows are very important. They allow us to write programs that wait for an operation to complete without using a dedicated thread (which consumes valuable resources). This also enables us to use different models for concurrency, such as the message passing style which is used in a successful functional language called Erlang.

> **Message passing in the Erlang language**
>
> Erlang is a language developed and heavily used by Ericsson for developing large scale real-time systems. It can be found in many of the Ericsson's telecommunication equipment. Erlang has been used commercially by Ericsson for programming their network hardware used concurrently by hundreds of users as well as by other companies.
>
> Concurrent applications in Erlang are described using independent processes (written in a functional way) that can communicate with each other using messages. The process waits for a message and when a message arrives, it processes it. We'll see how to use this style in F# using asynchronous workflows in chapter 13.

Before we take a look at the F# language and talk about the F# programming style, let's briefly talk about the history of functional programming, which is surprisingly rich.

1.3 The path towards real-world functional programming

The history of functional programming goes as far back as the 1930s when Alonzo Church and Stephen C. Kleene introduced a theory called Lambda calculus as part of their investigation of the foundations of mathematics. Even though it didn't fulfill their original expectations, it is still used in some branches of logic and has evolved into a very useful theory of computation. For curiosity and to show the basic principles of functional programming, you'll find a brief introduction to lambda calculus in the next chapter. However, lambda calculus escaped its original domain when computers were invented and served as an inspiration for the first of functional programming languages.

1.3.1 Functional languages

The LISP language, created by John McCarthy in 1958, was based on lambda calculus. LISP is an extremely flexible language, and it pioneered many programming ideas that are still used today, including data structures, garbage collection and dynamic typing.

In the 1970s, Robin Milner developed a language called ML. This was the first of a family of languages which now includes F#. Inspired by typed lambda calculus, it added the notion of types and even allowed writing "generic" functions in a same way as we can do now with .NET generics. ML was also equipped with a powerful type inference mechanism, which is now essential for writing terse programs in F#. OCaml, a pragmatic extension to the ML language appeared in 1996. It was one of the first languages that allowed the combination of object-oriented and functional approaches. OCaml was a great inspiration for F#, which has to mix these paradigms in order to be a first-class .NET language *and* a truly functional one.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Other important functional languages include Haskell (a language with surprising mathematical purity and elegance) and Erlang, which I have already mentioned in a sidebar. We'll learn more about some of these languages in the rest of the book, when talking about topics where they have some interesting benefits over F#–but first, let's finish our story by looking at the history of F#.

1.3.2 Functional programming on the .NET platform

The first version of .NET was released in 2002 and the history of the F# language dates back to the same year. F# started off as a Microsoft Research project by Don Syme and his colleagues, with the goal of bringing functional programming to .NET. F# and typed functional programming in general gave added weight to the need for generics in .NET and the designers of F# were deeply involved in the design and implementation of generics in .NET 2.0 and C# 2.0.

With generics implemented in the core framework, F# began evolving more quickly and the programming style used in F# also started changing. It began as a functional language with some support for objects, but as the language matured, it seemed more and more natural to take the best from both of these styles. As a result F# can be now more precisely described as a multi-paradigm language, which combines functional and object-oriented approach, together with a great set of tools that allow using F# interactively for scripting.

F# has been a first-class .NET citizen since its early days. This means that not only can it access any of the standard .NET components, but equally importantly any other .NET language can access code developed in F#. This makes it possible to use F# to develop standalone .NET applications as well as parts of larger projects. F# has always come with support in Visual Studio, and in 2007 a process was started to turn F# from a research project to a full production-quality language. In 2008 it was announced that F# will become one of the languages shipped with Visual Studio 2010. Now we know its origins, let's take a look at the language itself.

1.4 Introducing F#

We'll introduce F# in stages throughout the book, as and when we need to. In this section we'll just look at the very basics, writing a couple of short examples so you can start to experiment for yourself. We'll examine F# more carefully after summarizing important functional concepts in chapter 2. Our first real-world F# application will come in chapter 4.

After discussing the "Hello world" sample, I'll talk a little bit about F# to explain what you can expect from the language. We'll also discuss the typical development process used by F# developers, because it is quite different to what you're probably used to with C#.

1.4.1 Hello world in F#

The easiest way to start using F# is to create a new script file. Scripts are lightweight F# sources that don't have to belong to a project and usually have an extension "fsx". In Visual

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Studio, you can go to "File" - "New" - "File…" (Ctrl + N) and then select "F# Script File" from the "Scripts" category. Once we have the file, we can jump directly to the "Hello world" code.

Listing 1.11 Printing hello world (F#)

```
let message = "Hello world!"          #1
printfn "%s" message                  #2
```

#1 Value binding for 'message'
#2 Call to the 'printfn' function

I admit that this isn't the simplest possible "Hello world" in F#, but it would be fairly difficult to write anything interesting about the single line version. The listing 1.11 starts with a value binding (#1). This is similar to variable declaration, but there is one important difference - the value is immutable and we cannot change its value later. This matches with the overall functional style to make things immutable and we'll talk about this in detail in the next two chapters.

After assigning a value "Hello world" to a symbol `message`, the program continues with a call to a `printfn` function. It is important to note that arguments to F# functions are usually just separated spaces with no commas between them or surrounding parentheses. We'll sometimes write parentheses when it makes the code more readable, such as when writing `cos(1.57)`, but even in this case the parentheses are optional. I'll explain the convention that I'll use as we learn the core concepts of F# in the next couple of chapters.

The first argument to the `printfn` function is a format string. In our example, it specifies that the function should take only one additional parameter, which will be a string. The type is specified by the `%s` in the format string (the letter "s" stands for "string") and the types of arguments are even checked by the compiler. Now that we understand the code, let's look how we can run it.

INTERACTIVE PROGRAMMING IN F#

The easiest way to run the code is to use the interactive tools provided by F# tool chain. These tools allow you to use the interactive style of development. This means that you can easily try what code would do and verify whether it behaves correctly by running it with a sample input. Some languages have an interactive console, where you can paste code and execute it. This is called read-eval-print loop (REPL), because the code is evaluated immediately.

In F#, we can use a command prompt called F# interactive, but the interactive environment is also integrated inside the Visual Studio environment. This means that one can write the code with the full IDE and IntelliSense support, but also select a block of code and execute it immediately to test it.

Let's have a look at the results that we get when we run the code. If you're using F# interactive from command line, you'd just paste the previous code and type ";;" to execute

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

it. If you're using Visual Studio, you can select the code and hit Alt + Enter to send it to the interactive window. Listing 1.12 shows the result that you'll get.

Listing 1.12 Running the Hello world program (F# interactive)

```
MSR F# Interactive, (c) Microsoft Corporation, All Rights Reserved
F# Version 1.9.4.10, compiling for .NET Framework Version v2.0.50727

> (...);;                                                          #A

val message : string                                               #1
Hello world!                                                       #2
```

#A Source code goes here
#1 Information about value binding
#2 Printed output of 'printfn' call

The first line (#1) is generated by the value binding. It reports that a value called `message` was declared and that the type of the value is string. We didn't explicitly specify the type, but F# uses a technique called type inference to deduce the types of values, so the program is statically typed, just as in C#. The second line (#2) is an output from the `printfn` function, which prints the string and doesn't return any value.

Writing something like "Hello world" example doesn't demonstrate how working with F# looks at the larger scale. Let's now briefly look at the usual development process, because is quite interesting.

1.4.2 From simplicity to robustness

When starting a new project, you don't usually know at the beginning how the code will look at the end. At this stage, the code evolves quite rapidly. However, as it becomes more mature, the architecture becomes more solid and we're more concerned with the robustness of the solution rather than with the flexibility. Interestingly, these requirements aren't reflected in the programming languages and tools that you use. F# is very appealing from this point of view, because it reflects this in both tools and the language.

F# development process in a nutshell

I have already mentioned the F# interactive tool. It allows you to verify and test your code immediately while writing it. This tool is extremely useful at the beginning of the development process, because it encourages you to quickly try various different approaches and choose the best one. Also, when solving some problem where you're not 100% sure, you can immediately try the code. When writing F# code, you'll never spend a large time debugging the program. Once you first compile and run your program, you've already tested substantial part of it interactively.

When talking about "testing" in the early phase, I mean that you tried to execute the code with various inputs a couple of times to interactively verify that it works. In the later

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

phase, we can turn these snippets into unit tests, so the term "testing" means a different thing in the later phase. When working with the mature version of the F# code, we can use tools such as Visual Studio debugger or various unit testing frameworks.

Moreover, F# as a language reflects this direction as well. When you start writing a solution to some problem, you start with only the most basic functional constructs, because they make writing the code as easy as possible. Later, when you find the right way to approach the problem and you face the need to make the code more polished, you end up using more advanced features that make the code more robust, easier to document and also accessible from other .NET languages like C#.

Let's see what the development process might look like in action. I'll use a few more F# constructs, but we won't focus primarily on the code. The more important aspect is how the development style changes as the program evolves.

STARTING WITH SIMPLICITY

When starting a new project, you'll usually create a new script file and try implementing the first prototype or experiment with the key ideas. At this point, the script file contains sources of various experiments, often in an unorganized order. The figure 1.3 shows how your Visual Studio IDE might look like at this stage.

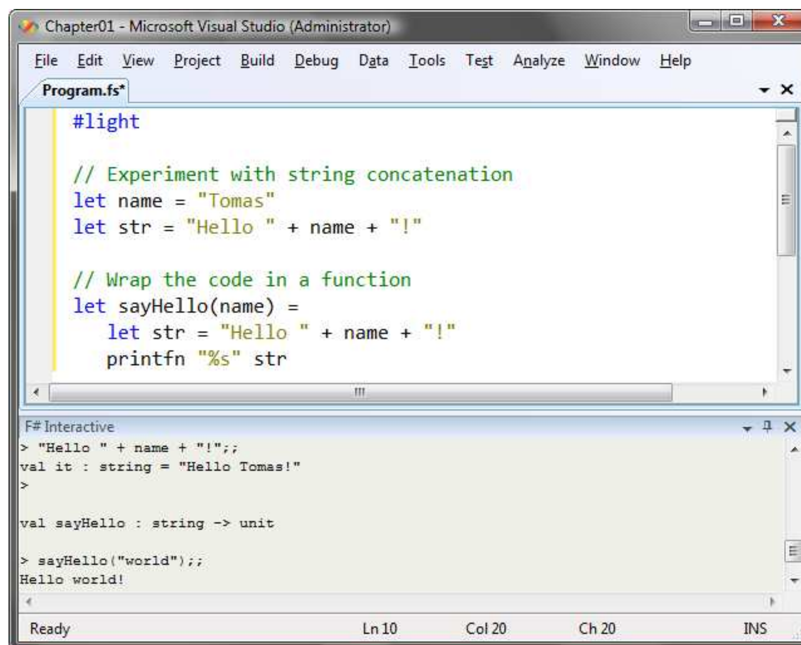


Figure 1.3 Using F# interactive we can first test the code and then wrap it into a function.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

The screenshot shows only the editor and the F# interactive window, but that's really all we need now, because we don't yet have any project. As you can see, I first wrote a few value bindings to try how string concatenation works in F# and entered the code to the F# interactive window to verify that it works as expected. Once I knew how to use string concatenation, I wrapped the code in a function. We'll talk about functions in chapter 3.

Next, I selected the function and hit Alt + Enter to send it to the F# interactive. After that, I entered an expressions `sayHello("world")` to test the function I just wrote. Note that the commands in F# interactive are terminated with "`;;`". This allows you to easily enter multi-line commands.

Once we start writing more interesting examples, you'll see that the simplicity is greatly supported by using of the functional concepts. Many of them allow you to write the code in a surprisingly terse way and thanks to the ability to immediately test the code F# is very powerful in the first phase of the development. We'll talk about the easy-to-use functional constructs mostly in the part 2 of this book. However, as the program grows larger, we need to write it in a more polished way and make it coherent with the usual .NET techniques. Fortunately, F# helps us to do this too.

Ending with robustness

Unlike many other languages that are popular for their simplicity, F# lives on the other side as well. In fact, it can be used for writing very mature, robust and safe code. The usual process is that you start with very simple code, but as the codebase becomes larger you refactor it in a way that makes it more accessible to other F# developers, enables writing better documentation and supports better interoperability with .NET and C#.

Perhaps the most important step in order to make the code well accessible from other .NET languages is to encapsulate the functionality into .NET classes. The F# language supports the full .NET object model, and classes authored in F# appear just like ordinary .NET classes with all the usual accompaniments such as static type information and XML documentation.

We'll talk about F# object types in chapter 9 and you'll see many of the robust techniques in part 4, but let me just shortly demonstrate this, to prove that you can use F# in a traditional .NET style as well. The listing 1.13 shows how to wrap the `sayHello` function in a C# style class and add Windows Forms user interface.

Listing 1.13 Object-oriented Hello world using Windows Forms (F#)

```
open System.Drawing                                                   #1
open System.Windows.Forms                                             #1

type HelloWindow() =                                                  #2
   let frm = new Form(Width = 400, Height = 140)                      #A
   let fnt = new Font("Times New Roman", 28.0f)                       #A
   let lbl = new Label(Dock = DockStyle.Fill, Font = fnt,             #A
                       TextAlign = ContentAlignment.MiddleCenter)     #A
   do frm.Controls.Add(lbl)                                           #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

```
    member x.SayHello(name) =                                          #3
       let msg = "Hello " + name + "!"
       lbl.Text <- msg                                                 #A

    member x.Run() =                                                   #4
       Application.Run(frm)
```

#1 Import necessary .NET namespaces
#2 F# class declaration
#A Constructor initializes the user interface
#3 Builds and displays the hello message
#4 Method that runs the application
#A Modify property of a .NET type

The example starts with several `open` directives (#1) that import types from .NET namespaces. Next, we declare the `HelloWindow` class (#2), which wraps the code to constructs the user interface and exposes two methods. The first method (#3) wraps the functionality for concatenating hello world messages that we interactively developed earlier. The second one runs the form as a standard windows forms application (#4). The class declaration appears just like ordinary C# class, with the difference that F# has a more lightweight syntax for writing classes. The code that uses the class in F# will look just like your usual C# code:

```
let hello = new HelloWindow()
hello.SayHello("dear reader")
hello.Run()
```

At this stage, we're developing the application in a traditional .NET style, so we'll run it as a standalone application. However, the interactive style helped us, because we had already interactively tested a part of the application. You can see how the resulting application looks in figure 1.4.

Figure 1.4 Running WinForms application created using object-oriented programming style in F#

In this section, we've had a quick taste of what the typical F# development process feels like. I haven't explained every F# construct we've used, because we'll see how everything works in detail in later chapters. We used a very simple example, so the second version of the code was still quite simple. However, it demonstrated that you can use F# language for writing a pretty standard .NET programs.

What can F# offer to a C# developer?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

As I said earlier, F# is well suited for writing code using simple concepts at the beginning and turning it into a traditional .NET version later, while C# is largely oriented towards the traditional .NET style. If you're C# developer, creating real-world applications you can easily take advantage of F# in two ways.

The first option is to use F# for rapid prototyping and experimenting with the code as well for exploring how .NET libraries work. As you've seen, using F# interactively is very easy, so writing a first sketch of the code can be done in F# and you save a lot of time when trying several approaches to a problem or exploring how a new library works. If you require code written in C#, then you can rewrite your prototype to C# later and still save a lot of development time.

However, F# is a fully compiled .NET language, so there are no technical reasons for preferring C# source code. This means that you can simply make sure that your library can be easily accessed from C# by turning the code from a simple to a traditional .NET version and use F# for example for writing parts of a larger .NET solution.

That should be enough about F# for now. It's possible that you're still finding some of the F# language constructs puzzling, but the purpose of this introduction wasn't to teach you F# in 4 pages, but I wanted to show you how F# looks and feels, so you can experiment with it as we'll look at more interesting examples in the subsequent chapters.

1.5 Summary

This chapter gave you a very brief overview of what makes functional programming interesting. We've talked about the declarative programming style, which is used when writing applications and libraries in a functional style. We've seen that this is already used in many successful technologies such as WPF and LINQ, but I also demonstrated that we can use it for writing functional solutions to other kinds of problems in C# 3.0.

We've also looked at parallel and asynchronous programming, which is a big challenge for modern software development. Using a functional approach makes it significantly easier thanks to the use of immutability and declarative programming. The first one gives us guarantees about the code and helps us writing correct and safe code and the second one is more expressive when solving problems.

In the next chapter, you'll see a much broader picture of functional programming. We'll look at all of the important ideas from a high-level perspective and you'll also see how they relate to each other. Even though we won't look at much real code yet, the next chapter will give you a solid foundation we can build on in the rest of the book.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460