Foreword by Don Syme, F# Community Contributor

Microsoft

F# for C# Developers

Professional

Tao Liu

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2013935410
ISBN: 978-0-7356-7026-6

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Rosemary Caperton
Editorial Production: Waypoint Press
Technical Reviewer: Daniel Mohl; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Indexer: Christina Yeager
Cover: Twist Creative • Seattle and Joel Panchot

Contents at a Glance

Table of Contents

Introduction

F# is a functional programming language from Microsoft. It is the first class language shipped in Visual Studio. It has been applied successfully in several areas, such as in the areas of financial software and web development. If you are a C# developer and want to use functional programming to write concise code with fewer bugs, F# is the right tool for you.

F# for C# Developers introduces, in an organized way, the F# language and several applications. It starts from how F# can perform imperative and object-oriented programming tasks and then moves on to covering unique F# features, such as type providers. By introducing F# design patterns with a large number of samples, this book not only delivers a basic introduction but also helps you apply F# in your daily programming work.

In addition to covering core F# core features, I also discuss F# HTML5 development, F# Azure development, and using general-purpose graphics processing units (GPGPUs) with F#. Beyond the explanatory content, each chapter includes examples and downloadable sample projects you can explore for yourself.

Who Should Read This Book

I wrote this book to help existing C# developers understand the core concepts of F# and help C# developers use F# in their daily work. It is especially useful for C# programmers looking to write concise code for algorithm design, web development, and cloud development. Although most readers will have no prior experience with F#, the book is also useful for those familiar with earlier versions of F# and who are interested in learning about the newest features.

You should have at least a minimal understanding of .NET development and object-oriented programming concepts to get the most benefit from this book. You also should have a basic understanding of data structures and generic algorithms. Experience in using C# is required as well.

Who Should Not Read This Book

This book is aimed at both experienced .NET C# developers who interested in extending their knowledge in functional programming and beginners in F# who want to understand F# and apply F# to their daily programming work. If you have no C# programming experience, this book might be difficult for you.

Organization of This Book

This book is divided into three sections, each of which focuses on a different aspect. Part I, "C# and F#," introduce how to port your C# knowledge to F#. This section introduces basic data structures and performing object-oriented implementations using F#. Part II, "F#'s Unique Features," introduces unique F# features and explains how to use them in your daily programming work. Part III, "Real-World Applications," introduces several real-world applications, including web development, Azure cloud development, and GPGPU.

Finding Your Best Starting Point in This Book

The various sections of *F# for C# Developers* cover a wide range of technologies. Depending on your needs and your existing understanding, you might want to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Follow these steps
New to F# but experienced with C#	Focus on Part I to understand the basics and Part II for some unique F# features
Familiar with earlier versions of F#	Briefly read Parts I and II if you need a refresher on the core concepts, but also want to focus on type providers.

Most of the book's chapters include hands-on samples that let you try out the concepts just learned. No matter which sections you choose to focus on, be sure to download and install the sample applications on your system.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow:

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold. A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.

System Requirements

You will need the following hardware and software to complete the practice exercises in this book:

- Windows 7 or Windows 8
- Visual Studio 2012, any edition (multiple downloads might be required if you're using Express Edition products)
- 1 GB (32 Bit) or 2 GBs (64 Bit) RAM
- 3.5 GBs of available hard disk space
- DirectX 9 capable video card running at 1024 x 768 or higher-resolution display
- DVD-ROM drive (if installing Visual Studio from DVD)
- Internet connection to download software or chapter examples
- If you want to run the GPU code, you need an NVIDIA graphics card and you need to download CUDA SDK from the NVIDIA web site.

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2012.

Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects, in both their pre-exercise and post-exercise formats, can be downloaded from F# sample pack site (*http://fsharp3sample.codeplex.com/*)

Follow the instructions to download the 670266_FSharp4CSharp_CompanionContent.zip file.

Note In addition to the code samples, your system should have Visual Studio 2012.

Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book:

1. Unzip file that you downloaded.
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the zip file.

Using the Code Samples

The sample code is organized by chapters. You can look at the folder that has the chapter name to look at the sample code.

Acknowledgments

First I'd like to thank Don Syme, who invented this fantastic language. I had a great time working with the Visual F# Core team, including Brian McNamara, Wonseok Chae, Vladimir Matveev, Matteo Taveggia, Jack Hu, Andrew Xiao, and Zack Zhang. Also, I would like to thank F# MVPs Daniel Mohl, Kit Eason, Zach Bray, Dave Thomas, and Don Syme for reviewing my book and providing valuable suggestions. It was a great experience exchanging ideas with so many talented software professionals. Devon Musgrave and Rosemary Caperton from Microsoft Press put a lot of work into editing this book. This book could never have been published without their efforts.

Finally, I would like to thank my wife, Rui Zhang, and my daughter, Zoey Liu, for their understanding and for sacrificing their time to support me in finishing this book. Without them, this book would never have become a reality.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://aka.ms/FsharpCsharpDev/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

Foreword

People often ask, "What can F# do that C# cannot?" In this book, you will discover much of what F# can do! You will see familiar things such as object programming and design patterns. Further, you will also see powerful new things like pattern matching, piping, first-class events, object expressions, options, tuples, records, discriminated unions, active patterns, agents, computation expressions and, perhaps most distinctively, type providers.

However, we also need to ask the other question: "What can C# do that F# cannot?" There is one important part to this answer that I will focus on here: C# can cause *NullReferenceExceptions*. "What?" I hear you ask. "Does F# not have nulls?" Right! Perhaps the most important thing the C# programmer needs to know about F# is that F# does not use nulls in routine programming.

Let's look at some evidence. People using F# at a major UK energy company did a study of two similar ETL (Extract, Transform, Load) applications.[1] Broadly speaking, the applications were in the same zone in terms of functionality or, if anything, the F# application implemented more features. The F# project had a very low bug rate, and its code was 26 times smaller. The size difference is not only the result of language differences; there are also differences in design methodology. The C# project is characterized by the inappropriate overuse of elaborate object abstractions often seen in Java projects—for example, elaborate and unnecessary class hierarchies.

Interestingly, the comparison records that the C# project had 3036 explicit null checks, where a functionally similar F# project had 27, a reduction of 112 times in the total number of null checks. The other statistics in the comparison shown are also compelling, particularly the "defects since go live": the F# code had zero defects since "go live," and the C# code had "too many." These are not unrelated: nulls cause defects. In my opinion, the lack of nulls in routine coding alone makes it worth switching your programming teams to F# where possible.

In this book, you will learn many wonderful things about F#. But don't lose sight of the big picture: F# is about writing accurate, correct, efficient, interoperable code that gets deployed on time in enterprise scenarios. It does this partly by removing the most pernicious of evils: nulls. If you and your team embrace it, then, all else being equal, your life will be simpler, happy, and more productive.

—Don Syme
F# Community Contributor

[1] *http://www.simontylercousins.net/journal/2013/2/22/does-the-language-you-choose-make-a-difference.html*

PART I

C# and F#

CHAPTER 1

C# and F# Data Structures

In this chapter, I'll compare and contrast various data structures from F# and C# programming languages. F# is a powerful multiparadigm language that supports imperative, object-oriented, and functional programming. C# is a multiparadigm language with more of a focus on imperative and object-oriented programming. A C# program usually consists of statements to change the program's state. An imperative language describes how to finish a task with exact steps. A functional-first language, like F#, is more declarative, describing what the program should accomplish.

One example of a programming language adopting functional programming is the C# version 3.0 introduction of LINQ (Language INtegrated Query). The growing popularity of Scala and Closure shows functional programming is growing. In addition, F# is another tool Microsoft ships with Microsoft Visual Studio to solve ever-changing programming challenges. Which language you choose to use depends on your experience and environment, but keep in mind you do not need to make an exclusive selection. I hope this book provides some information that helps you make appropriate decisions.

Any programming language is designed to perform some computation and to process data. The way that data is organized and stored is referred to as the *data structure*. This chapter introduces basic data structures for F#, explains how they relate to C#, and details how you can apply them to create imperative programs. I will follow the tradition in programming books of presenting a Hello-World-like application to introduce a new language. I will provide simple C# code along with the F# imperative equivalent.

Listing 1-1 shows an imperative approach that simply adds up the odd numbers from 0 to 100. C# supports functional programming (such as a LINQ feature), and there is a more concise way to implement the same functionality, which I'll show later in this chapter.

LISTING 1-1 A C# snippet that adds odd numbers from 0 to 100

Imperative C# implementation

```
// add all odd numbers from 0 to 100 and print out the result in the console
int sum = 0;
for (int i = 0; i<=100; i++)
{
    if (i%2 != 0)
         sum += i;
}

Console.WriteLine("the sum of odd numbers from 0 to 100 is {0}", sum);
```

F# implementation

```
let mutable sum = 0
for i = 0 to 100 do
    if i%2 <> 0 then sum <- sum + i
printfn "the sum of odd numbers from 0 to 100 is %A" sum
```

By porting this C# code to the F# equivalent, I'll cover the follow topics:

- The basic data type (such as primitive type literals). See the "Basic Data Types" section.
- The *if*, *while*, and *for* syntax. See the "Flow Control" section.

After implementing the same functionality in F#, I'll cover some F# data structures, such as Seq and *tuple*. Although this particular sample does not require Microsoft Visual Studio 2012, it is highly recommended that you install it, which is the minimum requirement for various samples in this book. I'll also introduce F# Interactive and some other useful add-ins to improve your overall F# programming experience.

Note Because Visual Studio IDE features are not the focus of this book, I encourage you to look at the MSDN website (*www.msdn.com*) or *Coding Faster: Getting More Productive with Microsoft Visual Studio* (Microsoft Press, 2011) to explore the topic by yourself.

Now it's time to start our journey!

Basic Data Types

F# is a .NET family language; therefore, the basic type definition and reference are similar to C#. Table 1-1 lists the C# and F# data types as well as the way to define a variable with each type. F# is a strongly typed language. Any errors related to type conversion are reported at compile time. These errors can be detected at an early stage of development and checked, which enables them to be fixed at compile time.

One big difference between the C# and F# definitions is that the F# examples do not need an explicitly defined type. This is because F# is often able to infer a type from the assigned value. To most C# developers, this feature is a lot like the *var* keyword in C#. There are some fundamental differences between *var* and *let*, but you can think of them as equals for now.

TABLE 1-1 Basic data types

Data Type	C# Representation	F# Representation
Int	int i = 0;	let i = 0 or let i = 0l
Uint	uint i = 1U;	let i = 1u or let i = 1ul
Decimal	decimal d = 1m;	let d = 1m or let d = 1M
Short	short c = 2;	let c = 2s
Long	long l = 5L;	let l = 5L
unsigned short	ushort c = 6;	let c = 6us
unsigned long	ulong d = 7UL;	let d = 7UL
byte	byte by = 86;	let by = 86y let by = 0b00000101y let by = 'a'B
signed byte	sbyte sby = 86;	let sby = 86uy let sby = 0b00000101uy
bool	bool b = true;	let b = true

double	double d = 0.2;
	double d = 0.2d;
	double d = 2e-1;
	double d = 2;
	double d0 = 0;
		let d = 0.2 or
		let d = 2e-1 or
		let d = 2.
		let d0 = 0x0000000000000000LF

float	float f = 0.3; or
	foat f = 0.3f;
	float f = 2;
	float f0 = 0.0f;
		let f = 0.3f or
		let f = 0.3F or
		let f = 2.f
		let f0 = 0x00000000000000000lf

native int	IntPtr n = new IntPtr(4);	let n = 4n
unsigned native int	UIntPtr n = new UIntPtr(4);	let n = 4un

Data Type	C# Representation	F# Representation
char	char c = 'c';	let c = 'a'
string	string str = "abc";	let str = "abc"
big int	BigInteger i = new BigInteger(9);	let i = 9I

One particular F# feature I'd like to call out is the syntax for creating an array of bytes to represent an ASCII string. Instead of asking you to constantly call into the *Encoding.ASCII.GetBytes* function, F# provides the "B" suffix to define an ASCII string. The string in .NET is Unicode-based. If you are mainly programming an ASCII string, you will not like this. In the following code, the representation for *asciiString* is a *byte[]* type internally:

```
let asciiString = "abc"B  // F# code
byte[] asciiBytes = Encoding.ASCII.GetBytes(value);  // C# code
```

Unlike C#, *float* in F# is a double-precision floating point number, which is equivalent to a C# *double*. The float type in C# is a single-precision numerical type, which can be defined in F# via the *float32* type. The .NET 32-bit and 64-bit floating numbers can be positive infinite or a NaN value. F# uses shortcut functions to represent these values:

- **Positive Infinity** infinity is System.Double. PositiveInfinity and infinityf is System.Single. PositiveInfinity
- **NaN** nan is System.Double.NaN and nanf is System.Single.NaN

The F# compiler does not allow any implicit type conversion. For a C# developer, an integer can be converted to a float implicitly, and this gives the impression that *29* is the same as *29.0*. Because implicit conversion is not allowed in F#, the explicit conversion *float 29* is needed to convert the integer *29* to a float type. The explicit conversion can eliminate the possibility of lose precision when the conversion is implicit.

F# 2.0 had two syntaxes for strings: normal strings, and verbatim strings, which are prefixed by the at sign (@). F# 3.0 introduces a new feature to define strings using a triple-quoted string.

Triple-Quoted Strings

F# supports normal strings and verbatim strings. This is equivalent to the options that C# provides. Examples of normal and verbatim string definitions are shown in Listing 1-2. The execution result shown in the listing is an example of a normal string and verbatim string being bound to specific values within the F# Interactive window (which I'll introduce shortly in the "Using F# Interactive" section). The result shows the variable name, type, and value.

LISTING 1-2 Normal and verbatim strings

```
let a = "the last character is tab\t"
let b = @"the last character is tab\t"
```

Execution result in the F# Interactive window

```
val a : string = "the last character is tab        "
val b : string = "the last character is tab\t"
```

Normal and verbatim strings are useful for a variety of tasks. However, scenarios that require included characters, such as double quotes, are still difficult to implement because of the need to escape these characters. Listing 1-3 shows examples of this.

LISTING 1-3 The escape double quote (")

```
// use backslash (\) to escape double quote
let a = "this is \"good\"."

// use two double quote to escape
let b = @"this is ""good""."
```

F# 3.0 introduces a new string format—a triple-quoted string—that alleviates this pain. Everything between the triple quotes (""") is kept verbatim; however, there is no need to escape characters such as double quotes. Triple-quoted strings have a number of use cases. A few examples include the creation of XML strings within your program and the passing of parameters into a type provider. Listing 1-4 shows an example.

LISTING 1-4 A triple-quoted string

```
let tripleQuotedString = """this is "good"."""

// quote in the string can be at the beginning of the string
let a = """"good" dog"""

// quote in the string cannot be at the end of the string
// let a = """this is "good""""
```

Note Quotes in the triple-quoted string cannot end with a double-quote ("), but it can begin with one.

Variable Names

How to define a variable name is a much-discussed topic. One design goal for F# is to make variable names resemble more normal human language. Almost every developer knows that using a more readable variable name is a good practice. With F#, you can use double-backticks to include

nonalphabet characters in the variable name and eventually improve the readability of your code. Examples are shown in Listing 1-5.

LISTING 1-5 Defining a variable

```
// variable with a space
let ``my variable`` = 4

// variable using a keyword
let ``let`` = 4

// apostrophe (') in a variable name
let mySon's = "Feb 1, 2010"
let x' = 3

// include # in the variable name
let ``F#`` = "this is an F# program."
```

Flow Control

To write F# applications in an imperative style, you need to know how to define flow-control statements. F# supports several types of flow control to accomplish this, including the *for* loop, *while* loop, and *if* expression. These statements segment the program into different scopes. C# uses "{" and "}" to segment code into different scopes, while F# does not use those items. Instead, F# uses the space indent to identify different program scopes. This section discusses these three statements in detail.

Note Visual Studio can automatically convert the Tab key to a space. If you edit F# code in another editor that does not support this conversion, you might have to do it manually.

for Loop

There are two forms of the *for* loop: *for...to/downto* and *for...in*. The *for...to/downto* expression is used to iterate from a start value inclusively to or down to an end value inclusively. It is similar to the *for* statement in C#.

FOR...IN is used to iterate over the matches of a pattern in an enumerable collection—for example, a range expression, sequence, list, array, or other construct that supports enumeration. It is like *foreach* in C#. Looking back at the C# code that began this chapter, you see that you can use two F# options (as shown in Listing 1-6) to accomplish the loop of code for each number between 0 and 100. The first approach uses FOR...TO, and the second approach uses *for...in*.

LISTING 1-6 A *for* loop

C# version

```
for (int i=0; i<=100; i++)
```

F# versions

```
// for loop with i from 0 to 100
for i=0 to 100 do ...

// for iterate the element in list 0 to 100
for i in [0..100] do ...
```

***for...downto* sample**

```
// downto go from 100 to 0
for i=100 downto 0 do ...
```

Note The *[0..100]* defines a list with elements from 0 to 100. The details about how to define a list are discussed later in this book.

Some readers might immediately ask how to make the *for...to/downto* to increase or decrease by 2. *for...to/downto* does not support this, so you have to use *for...in* with a sequence or list. And I can assure you, you will not use *for* loop that often when you understand how to use a sequence or list.

while Loops

Another approach that could be used to accomplish the goal of this example is to use a *while* loop. F# and C# approach the *while* loop in the same way. Listing 1-7 shows an example.

LISTING 1-7 A *while* loop

C# version

```
int i = 0;
while (i<=100)
{
    // your operations
    i++;
}
```

F# version

```
let mutable i = 0
while i <=100 do
    <your operations>
    i <- i + 1
```

Note It's optional to use a semicolon to end a statement. The semicolon is needed only when multiple statements are placed on the same line.

The definition for variable *i* in the previous code snippet has the *mutable* keyword in the definition. The *mutable* keyword indicates *i* is a mutable variable, so its content can be modified by using the <– operator. This brings up an interesting and crucial concept in F#: a variable without the *mutable* keyword is an immutable variable, and therefore its value cannot be changed. The C# code `int i = 0` is equivalent to the F# code `let mutable i = 0`. This looks like a small change, but it is a fundamental change. In C#, the variable is mutable by default. In F#, the variable is immutable by default. One major advantage to using immutable variables is multi-thread programming. The variable value cannot be changed and it is very easy and safe to write multi-thread program.

Although F# does not provide *do…while* loop, it won't be a problem for an experienced C# developer if he is still willing to use the C# imperative programming model after learning about F#. Actually, the more you learn about F#, the less important the *do…while* loop becomes.

if Expressions

At this point, the only part left is the *if* expression. In the earlier example, you need *if* to check whether the value is an odd number. Note that in F# *if* is an expression that returns a value. Each *if/else* branch must return the same type value. The *else* branch is optional as long as the *if* branch does not return any value. The *else* must be present if the *if* branch returns a value. It is similar to the "?:" operator in C#. Although a value must be returned, that returned value can be an indicator of no value. In this case, F# uses *"unit"* to represent the result. Unlike C#'s *if...else*, F# uses *elif* to embed another *if* expression inside. Listing 1-8 shows an example of this. In Listing 1-9, you can see a comparison between the C# and F# code required to check that a value is odd or even.

LISTING 1-8 An *if* expression

```
if x>y then "greater"
elif x<y then "smaller"
else "equal"
```

LISTING 1-9 An *if* expression

C# version

```
if (i%2 != 0) ...
```

F# version

```
if i%2 <> 0 then ...
```

Match

In addition to the *if* statement, C# and F# have another way to branch the execution of code. C# provides a *switch* statement, and F# provides a *match* expression. F# developers can use a *match* expression to achieve the same functionality as the *switch* statement in C#, but the power of *match* expressions does not stop there. I will discuss the additional features that *match* provides in Chapter 6, "Other Unique Features." An example of a simple implementation of *match* that is similar in concept to a C# *switch* statement is shown in Listing 1-10.

LISTING 1-10 A *match* and *switch* sample

C# *switch* statement

```
int i = 1;
switch (i)
{
    case 1:
        Console.WriteLine("this is one");
        break;
    case 2:
        Console.WriteLine("this is two");
        break;
    case 3:
        Console.WriteLine("this is three");
        break;
    default:
        Console.WriteLine("this is something else");
        break;
}
```

F# *match* statement

```
let intNumber = 1

match intNumber with
    | 1 -> printfn "this is one"
    | 2 -> printfn "this is two"
    | 3 -> printfn "this is three"
    | _ -> printfn "this is anything else"
```

Console Output

Now you have almost everything to make the functionality work. The last missing piece is to let the computer tell you what was achieved by using console output. In the C# code, you use the *Console.WriteLine* method. Because F# is a .NET language, you can use *Console.WriteLine* from it as well. However, F# also provides a function called *printfn* that provides a more succinct and powerful option. Listing 1-11 shows an example of both of these approaches.

LISTING 1-11 The console output

C# version

```
Console.WriteLine("the sum of odd numbers from 0 to 100 is {0}", sum);
```

F# version

```
// use printfn to output result
printfn "the sum of odd numbers from 0 to 100 is %A" sum

// use Console.WriteLine to output result
System.Console.WriteLine("the sum of odd numbers from 0 to 100 is {0}", sum)
```

F#'s *printfn* is stricter than C#'s *Console.WriteLine*. In C#, *{<number>}* can take anything and you do not have to worry about the type of variable. But F# requires that the placeholder have a format specification indicator. This F# feature minimizes the chance to make errors. Listing 1-12 demonstrates how to use different type-specification indicators to print out the appropriate values. If you really miss the C# way of doing this, you can use *%A*, which can take any type. The way to execute the code will be explained later in this chapter.

LISTING 1-12 The *printfn* function and data types

```
let int = 42
let string = "This is a string"
let char = 'c'
let bool = true
let bytearray = "This is a byte string"B

let hexint = 0x34
let octalint = 0o42
let binaryinteger = 0b101010
let signedbyte = 68y
let unsignedbyte = 102uy

let smallint = 16s
let smalluint = 16us
let integer = 345l
let usignedint = 345ul
let nativeint = 765n
```

```
let unsignednativeint = 765un
let long = 12345678912345789L
let unsignedlong = 12345678912345UL
let float32 = 42.8F
let float = 42.8

printfn "int = %d or %A" int int
printfn "string = %s or %A" string string
printfn "char = %c or %A" char char
printfn "bool = %b or %A" bool bool
printfn "bytearray = %A" bytearray

printfn "hex int = %x or %A" hexint hexint
printfn "HEX INT = %X or %A" hexint hexint
printfn "oct int = %o or %A" octalint octalint
printfn "bin int = %d or %A" binaryinteger binaryinteger
printfn "signed byte = %A" signedbyte
printfn "unsigned byte = %A" unsignedbyte

printfn "small int = %A" smallint
printfn "small uint = %A" smalluint
printfn "int = %i or %A" integer integer
printfn "uint = %i or %A" usignedint usignedint
printfn "native int = %A" nativeint

printfn "unsigned native int = %A" unsignednativeint
printfn "long = %d or %A" long long
printfn "unsigned long = %A" unsignedlong
printfn "float = %f or %A" float32 float32
printfn "double = %f or %A" float float
```

Execution result

```
int = 42 or 42
string = This is a string or "This is a string"
char = c or 'c'
bool = true or true
bytearray = [|84uy; 104uy; 105uy; 115uy; 32uy; 105uy; 115uy; 32uy; 97uy; 32uy; 98uy;
121uy; 116uy; 101uy; 32uy; 115uy; 116uy; 114uy; 105uy; 110uy; 103uy|]
hex int = 34 or 52
HEX INT = 34 or 52
oct int = 42 or 34
bin int = 42 or 42
signed byte = 68y
unsigned byte = 102uy
small int = 16s
small uint = 16us
int = 345 or 345
uint = 345 or 345u
native int = 765n
unsigned native int = 765un
long = 12345678912345789 or 12345678912345789L
unsigned long = 12345678912345UL
float = 42.800000 or 42.7999992f
double = 42.800000 or 42.8
```

The Console has *In*, *Out*, and *Error* standard streams. F# provides *stdin*, *stdout*, *stderr*, which correspond to these three standard streams. For the conversion task, you already have all the building blocks. So let's give it a try in Listing 1-13.

LISTING 1-13 The C# and F# versions of adding odd numbers from 0 to 100

C# version

```
// add all odd numbers from 0 to 100 and print out the result in the console
int sum = 0;
for (int i = 0; i<=100; i++)
{
    if (i%2 != 0)
         sum += i;
}

Console.WriteLine("the sum of odd numbers from 0 to 100 is {0}", sum);
```

F# version

```
let mutable sum = 0
for i = 0 to 100 do
    if i%2 <> 0 then sum <- sum + i
printfn "the sum of odd numbers from 0 to 100 is %A" sum
```

Listing 1-14 shows how to use a list and a *for...in* loop to solve the same problem. Compared to Listing 1-13, this version has the following changes:

- Uses *for...in* to iterate through 0 to 100, where *[1..100]* is an F# list definition
- Uses the *printf* function, which does not output the *"\n"*
- Replaces *%A* with *%d*, which tells the compiler that the *sum* variable must be an integer

LISTING 1-14 Using the F# list in the *for* loop

```
let mutable sum = 0
for i in [0..100] do
    if i%2 <> 0 then sum <- sum + i
printf "the sum of odd numbers from 0 to 100 is %d \n" sum
```

Run Your Program

You can run your program from Visual Studio in two ways: create an F# project, much like you would a C# project, or use the F# Interactive window. F# supports the following project types in Visual Studio 2012:

- *F# Application* is a console-application project template.
- *F# Library* is a class-library template.
- *F# Tutorial* is a console application that contains F# samples. I highly recommend going through all of these samples.
- *F# Portable Library* is a class library for F# libraries that can be executed on Microsoft Silverlight, Windows Phone, and Windows platforms, including Windows 8.
- *F# Silverlight Library* is a Silverlight class-library template.

If you want to execute the sample code shown in this chapter up to this point, the F# application project is a good choice.

Note Microsoft Visual Studio Express 2012 for Web is free. Although its name suggests it is for web development and does not support a portable library, you can use it to create a console application by using the F# tutorial template.

Creating a Console Application

Figure 1-1 shows the project template list. You can select F# Application and accept the default name. This F# console-application template creates a solution with a console-application project that includes a default Program.fs file, as you can see in Figure 1-2. To run the simple summing application we've been referring to throughout this chapter, simply replace the content of Program.fs with the F# code from Listing 1-13. The steps are primarily the same for the creation of other project types, so I'll leave this to you to explore.

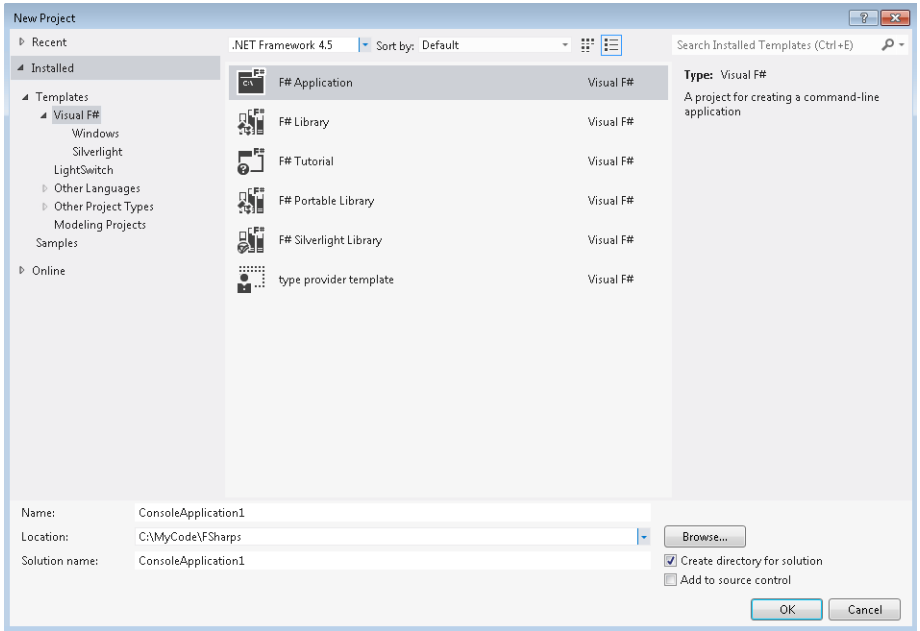

FIGURE 1-1 Creating an F# project

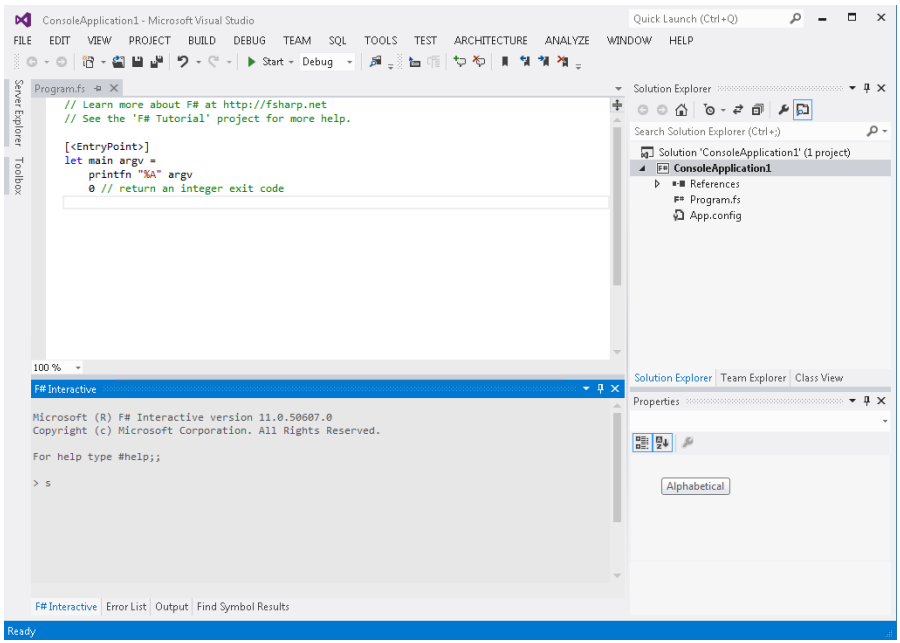

FIGURE 1-2 An F# console application with Program.fs

Using F# Interactive

For simple programs like the one in Listing 1-13, F# ships with an F# Interactive feature (FSI). You can use this to test small F# code snippets. In Visual Studio, the F# Interactive window can be found in the View menu. Depending on the development profile you're using, you can find the F# Interactive window under View, Other Windows, or you can access it directly in the View menu, as shown in Figure 1-3. An example of the FSI window is shown in Figure 1-4.

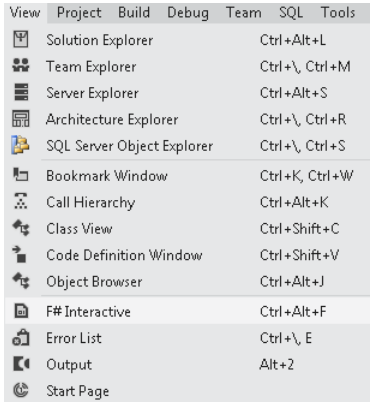

FIGURE 1-3 Accessing F# Interactive from the View menu

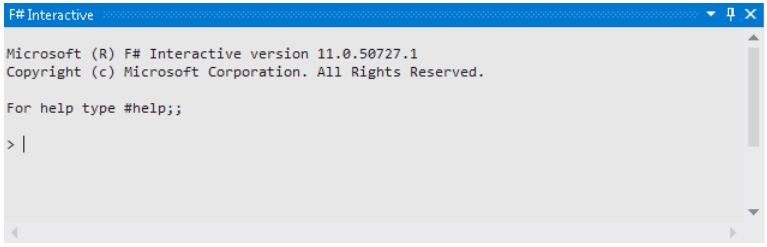

FIGURE 1-4 An F# Interactive window

The FSI window accepts user input, so you can execute your code directly in it. You can use two semicolons (;;) to let FSI know that the statement is finished and can be executed. One major limitation for FSI is that the FSI window does not provide Microsoft IntelliSense. If you don't want to create a full project and still want to use IntelliSense, the F# script file is your best option. You can go to File, New to create a new script file, as shown in Figure 1-5.

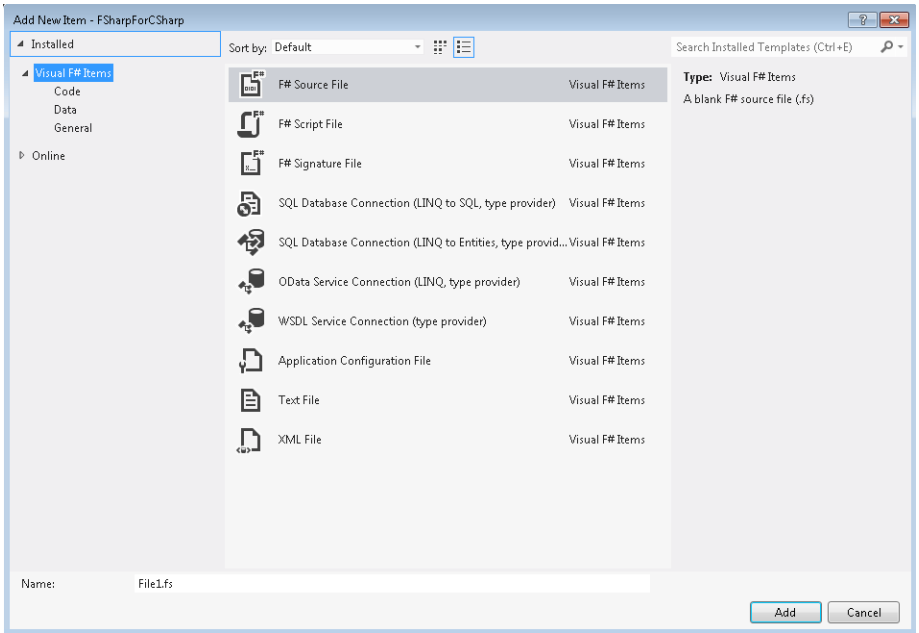

FIGURE 1-5 An F# item template

Note Many item templates are listed in Figure 1-5. I'll introduce them later. For now, you need only an F# source file and an F# script file.

The primary difference between an F# source file and an F# script file is the build action. The F# source file is a file with an extension of .fs, which will be compiled. Its action is set to Compile. The F# script file has an extension of .fsx, and its build action is set to None, which causes it to go into the build process by default. No matter which file type you decide to use, you can always execute the code by selecting it and using the context (that is, right-click) menu option Execute In Interactive. If you prefer using the keyboard, Alt+Enter is the keyboard shortcut as long as the development profile is set to F#. This command sends the selected code to be executed in FSI, as shown in Figure 1-6. There is also another menu option labeled Execute Line In Interactive. As its name suggests, this option is used to send one line of code to the FSI. The shortcut key for Execute Line In Interactive is Alt + '.

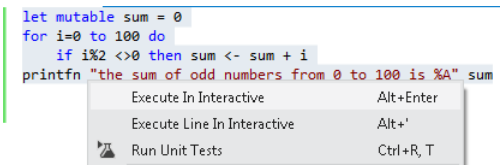

FIGURE 1-6 Executing code in FSI via the context menu

OK, let's put the code in the Program.fs. After that, you can select the code and send it to FSI. The execution result is shown in the FSI window, which then displays the expected result of "the sum of odd numbers from 0 to 100 is 2500," as shown in Figure 1-7. Congratulations! You've got your first F# program running.

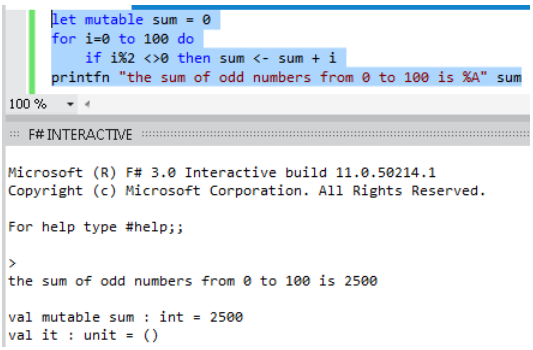

FIGURE 1-7 The execution result in the FSI window

Tip FSI provides a *#time* switch you can use to measure the execution time of your code and Gen 0/1/2 collection numbers. An example of the *#time* switch usage is shown in Listing 1-15. Interested users can perform a long run computation and see how this option works. Other directives can be found in the "FSI Directives" section later in the chapter.

LISTING 1-15 Switching the timing on and off

```
 > #time "on";;

--> Timing now on

> #time "off";;

--> Timing now off
```

After executing the program, FSI's state is changed causing it to become *polluted*. If you need a clean environment, you can use Reset Interactive Session. If you want to clear only the current output, you should select Clear All. The context menu (shown in Figure 1-8) shows all the available options. You can bring it up by right-clicking in the FSI window.

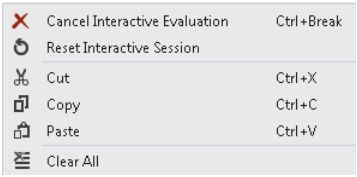

FIGURE 1-8 The FSI context menu

The full list and a description of each command provided in the FSI context menu is shown in Table 1-2.

TABLE 1-2 FSI commands

FSI Command	Description
Cancel Interactive Evaluation	Cancels the current FSI execution.
Reset Interactive Session	Resets the current FSI execution session.
Cut	Cuts the selection in the current editing line to the clipboard. The result from a previous execution or banner cannot be cut.
Copy	Copies the selection to the clipboard.
Paste	Pastes the clipboard text content to the current editing line.
Clear All	Clears all content in the FSI window, including the copyright banner.

FSIAnyCPU

The FSIAnyCPU feature was added with Visual Studio 2012. FSI will be executed as a 64-bit process as long as the current operating system is a 64-bit system. The FSIAnyCPU feature can be enabled by clicking Option, F# Tools, F# Interactive, as shown in Figure 1-9.

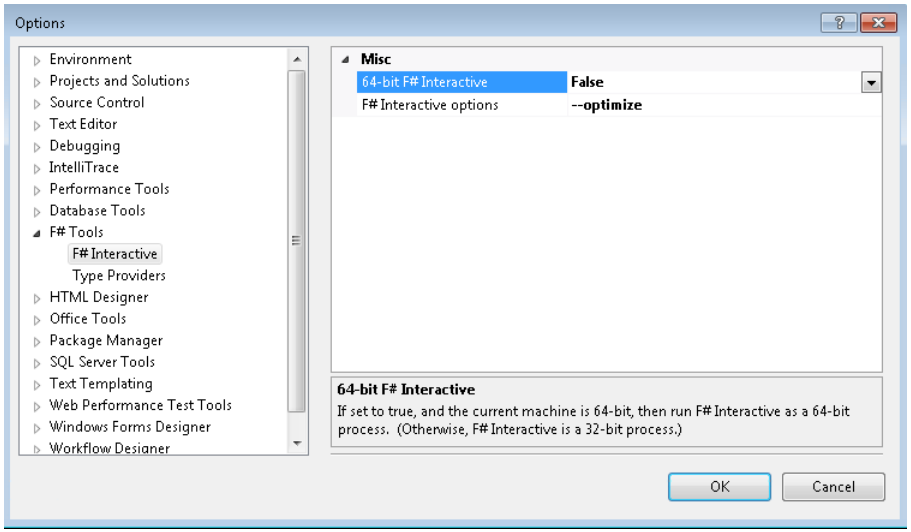

FIGURE 1-9 Enabling or disabling FSIAnyCPU

You can use Process Manager to check whether the FSIAnyCPU process is running, or you can use the *sizeof* operator to check the current *IntPtr* size. The 32-bit machine's *IntPtr* is 32-bit, so the *sizeof* operator returns *4* while the 64-bit machine will return *8*. Listing 1-16 shows the execution result from my 32-bit laptop.

LISTING 1-16 The *sizeof IntPtr* operator in FSI

```
> sizeof<System.IntPtr>;;
val it : int = 4
```

FSI Directives

In addition to the *#time* directive, FSI offers several others:

- *#help* is used to display the help information about available directives.
- *#I* is used to add an assembly search path.
- *#load* is used to load a file, compile it, and run it.
- *#quit* is used to quit the current session. You will be prompted to press Enter to restart. This is how to restart a session from the keyboard.
- *#r* is used to reference an assembly.

The FSI is a great tool that can be used to run small F# snippets of your code for test purposes. If your code is used to perform file I/O operations, the FSI's default directory is the temp folder. Listing 1-17 shows how to get the current FSI folder and change its default folder.

LISTING 1-17 Changing FSI's current folder

```
> System.Environment.CurrentDirectory;;
val it : string = "C:\Users\User\AppData\Local\Temp"

> System.Environment.CurrentDirectory <- "c:\\MyCode";;
val it : unit = ()

> System.Environment.CurrentDirectory;;
val it : string = "c:\MyCode"
```

Note After you reset the FSI session, the current folder will be set back to the temp folder.

Compiler Directives

FSI is a nice feature to have when you want to execute small programs. However, it is not a good choice for building executable binaries. To build binaries, you need to use Visual Studio. We'll use it to create one of the projects previously mentioned in this chapter. The build and execution process and experience is largely the same for both F# and C# applications, though they have different compilers. I already presented the FSI directives, and I will now list the F# compiler directives. The following five directives are supported by F#:

- *if* is used for conditional compilation. Its syntax is *if* <*symbol*>. If the symbol is defined by the compiler, the code after the *if* directive is included in the compilation, as shown in Listing 1-18.
- *else* is used for conditional compilation. If the symbol is not defined, the code after *else* is included in the compilation, as shown in Listing 1-18.
- *endif* is used for conditional compilation, and it marks the end of the conditional compilation. This is also shown in Listing 1-18.
- *line* indicates the original source code line and file name.
- *nowarning* is used to disable one or more warnings. F# tends to be more restrictive and gives more warnings than C# does. If your organization has a zero-warning policy, you can ignore specific warnings by using the *nowarning* directive. Only a number is needed as a suffix to a *nowarning* directive, and you can put multiple warning numbers in one line, as in the following example:

```
nowarning "1" "2" "3"
```

LISTING 1-18 A conditional compilation

```
#if VERSION1

let f1 x y =
   printfn "x: %d y: %d" x y
   x + y

#else

let f1 x y =
   printfn "x: %d y: %d" x y
   x - y

#endif
```

Note There is no *#define* directive. You have to either use a compiler option to define a symbol or define that symbol in the project properties. An example is shown in Figure 1-10.

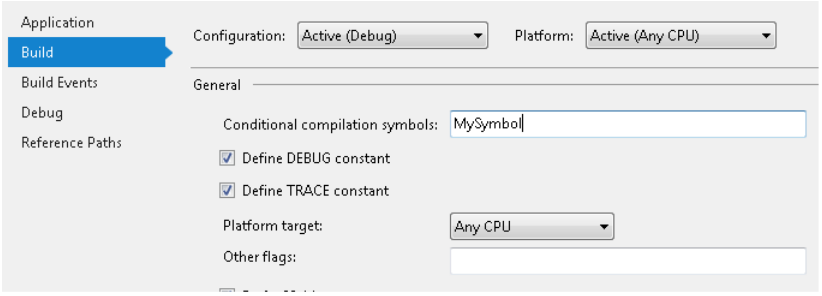

FIGURE 1-10 Defining a compile symbol

The INTERACTIVE compile symbol is a build-in compile symbol. The code wrapped by this symbol will be included in the FSI execution but not in the project build process. Listing 1-19 provides an example. If you are trying to include code only during the project build, the COMPILED symbol can be used, as shown in Listing 1-20.

LISTING 1-19 The INTERACTIVE symbol

```
#if INTERACTIVE

#r "System.Data"
#r "System.Data.Linq"
#r "FSharp.Data.TypeProviders"

#endif
```

LISTING 1-20 The COMPILED symbol

```
#if COMPILED

printfn "this is included in the binary"

#endif
```

Some Useful Add-ins

Visual Studio is a powerful editor with a rich set of editing features. However, some Visual Studio add-ins designed for F# are still recommended as a way to improve your coding experience.

- **F# depth colorizer** Because F# uses space indents to scope the code, you can run into some seemingly weird errors only because an extra space is needed 10 lines earlier. This extension can highlight this type of indentation problem. This add-in is used to help align code blocks by using different colors. I strongly recommend that you install it if your project gets big. You can download it from the Visual Studio gallery at *http://visualstudiogallery.msdn.microsoft.com/0713522e-27e6-463f-830a-cb8f08e467c4*. Figure 1-11 shows an example of the F# depth colorizer in use.

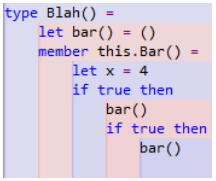

FIGURE 1-11 The F# depth colorizer

- **F# code snippet** The code snippet add-in brings the common code—for example, the class definition—to your fingertips. Unlike the C# snippet, the F# snippet also adds any needed dynamic-link library (DLL) references into the project. You can download the add-in and snippet files from *http://visualstudiogallery.msdn.microsoft.com/d19080ad-d44c-46ae-b65c-55cede5f708b*. An example of the extension in use is shown in Figure 1-12. The configuration options that the tool provides are shown in Figure 1-13.

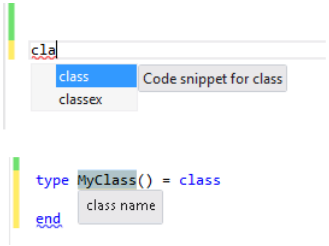

FIGURE 1-12 The F# code snippet

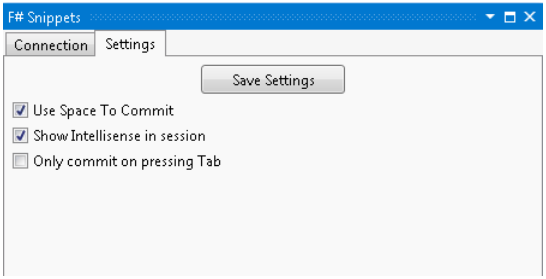

FIGURE 1-13 Configuring the F# code snippet add-in

- **Add reference add-in** Visual Studio's project system provides a nice UI to manage the reference DLLs. This add-in sends reference statements to FSI and adds reference scripts to the Script folder in the current project. Take a look at Figure 1-14.

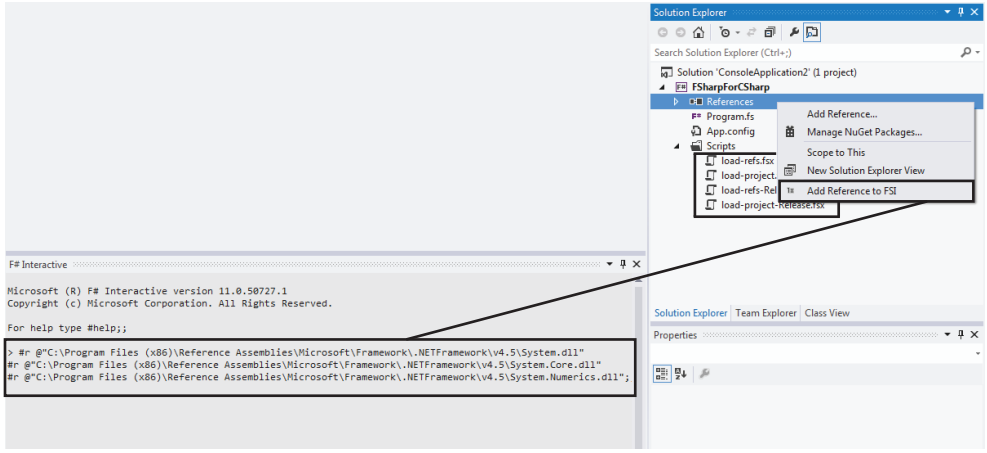

FIGURE 1-14 Adding a reference add-in

List, Sequence, and Array Data Structures

We successfully finished our first task: converting a simple C# program to F#. These days, many C# developers might choose to use LINQ to solve this problem. As I mentioned in this chapter's introduction, long before C# had this LINQ feature, F# had it as a functional programming feature. In this section, you'll learn how to define and use collection data, including the following items: list, sequence, and array. After introducing the list, sequence, and array structures, I'll show you how to use a functional programming style to convert simple C# programs to F#.

Lists

First, we start with the list structure, which you used once in the C#-to-F# conversion task. An F# list is an ordered, immutable series of same-type elements. Listing 1-21 shows different ways to define a list.

LISTING 1-21 Defining an F# list

```
//defines a list with elements from 1 to 10.
let list0 = [1..10]

//defines a list with element 1, 2, and 3.
let list1 = [1;2;3]

//defines a list with elements 0, 1, 4, 9, and 16.
let list2 = [for i=0 to 4 do yield i*i]

//defines an empty list
let emptyList = [ ]
let emptyList2 = List.empty
```

Note The *emptyList* element invokes the *Empty* function, which returns *List.Empty*, while *emptyList2* returns *List.empty* directly.

Unlike C#, F# uses a semicolon to separate the element in an array.

There are two operators that are useful when working with a list:

- **:: (cons) operator** The *::* operator attaches an element to a list. The F# *list* class has a constructor that takes an element and a list. This operator actually invokes this constructor and returns a new F# list. Its time complexity is O(1).

  ```
  let list1With4 = 4::list4
  ```

  Here, *list1With4* is a list defined as = `[4;1,2;3]`.

- **@ operator** The @ operator concatenates two lists and returns a new instance of F# list It time complexity is O(min(M, N)) where M is list0's length and N is list1's length.

  ```
  let list0And1 = list0 @ list1
  ```

  Here, *list0And1* is a list defined as = `[1;2;3;4;5;6;7;8;9;10;1;2;3]`, where *1*, *2*, and *3* are from *list1*.

Lists support indexing. Unlike C#, if you want to use an indexer in F#, you need to use dot notation. This means that you need to put an extra dot between the list variable and the indexer. Listing 1-22 shows this in action. F# list is a linked list and the time complexity is O(i), where *i* is index passed in the statement.

LISTING 1-22 An indexer in an F# list

```
list0[0]  //won't compile
list0.[0] // correct. Using a dot notation
```

F# lists support something called *structural equality*. Structural equality is to check equivalent identity. Listing 1-23 shows that the list can be equal if elements in both lists are equal. Comparisons between incongruous lists (apples-to-oranges comparisons) are not allowed. As an example, Listing 1-24 will not compile because the comparison between *TextBox* and *Button* doesn't make sense. Additionally, comparisons can be performed only if the elements support equality comparisons. Structural comparison is to provide an ordering of values.

LISTING 1-23 A list comparison

List comparison code

```
let l1 : int list = [ 1; 2; 3 ]
let l2 : int list = [ 2; 3; 1 ]

printfn "l1 = l2? %A" (l1 = l2)
printfn "l1 < l2? %A" (l1 < l2)
```

Execution result

```
l1 = l2? False
l1 < l2? True
```

LISTING 1-24 An example of how elements in a list cannot be compared

```
// the following code compiles
open System.Windows.Forms

let l1 = [ new TextBox(); new Button(); new CheckBox() ]
let l2 = [ new Button(); new CheckBox(); new TextBox() ]

// the following code does not compile
// printfn "%A" (l1 < l2) // not compile
```

Note If you want to use the F# list type in a C# project, you need to add a reference to *Microsoft.FSharp.Core.dll*.

The F# list might suggest it has some relationship with the *List<T>* type. Actually, the list type is a *Microsoft.FSharp.Collections.FSharpList<T>* type. It does implement the *IEnumerable<T>* interface, but it is not very similar to the *List<T>* type.

Sequences

According to MSDN documentation, a *sequence* is a logical series of elements of one type. Sequences are particularly useful when you have a large, ordered collection of data but do not necessarily expect to use all the elements. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all of the elements are needed. Any type that implements the *System.IEnumerable* interface can be used as a sequence. Defining a sequence is similar to defining a list. Listing 1-25 shows a few examples of how to define a sequence.

LISTING 1-25 Defining a sequence in F#

```
// defines a sequence with elements from 1 to 10.
let seq0 = seq { 1..10 }

// defines a sequence with elements 0, 1, 4, 9, and 16.
let seq2 = seq { for i=0 to 4 do yield i*i }

// defines a sequence using for…in
let seq1 = seq {
    for i in [1..10] do i * 2
}

// defines an empty sequence
let emptySeq = Seq.empty
```

Note Be aware that `seq { 1; 2; 3 }` is not a valid way to define a sequence. However, you can use the *yield* keyword to define a sequence as shown here: `seq { yield 1; yield 2; yield 3 }`.

A sequence is shown as an *IEnumerable<T>* type when viewed in C#. When you expose a sequence to a C# project, you do not need to add *Microsoft.FSharp.Core.dll*.

Arrays

The definition from MSDN says that *arrays* are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type. Listing 1-26 shows how to define an array.

LISTING 1-26 Defining an F# array

```
// defines an array with elements from 1 to 10.
let array0 = [| 1..10 |]

// defines an array with elements 1, 2, and 3.
let array1 = [| 1;2;3 |]

// defines an array with elements 0, 1, 4, 9, and 16.
let array2 = [| for i=0 to 4 do yield i*i |]

// defines an empty array
let emptyArray = [| |]
let emptyArray2 = Array.empty
```

Note Like the empty case in *Seq*, both *emptyArray* and *emptyArray2* invoke a function that returns *Array.empty*.

Arrays support indexing. Unlike C#, when you use an indexer in F#, you need to use dot notation. Therefore, an extra space is needed between the variable name and indexer. Listing 1-27 shows an example of this.

LISTING 1-27 An indexer in an F# array

```
array0[0]  //won't compile
array0.[0] // correct. Using a dot notation
```

Arrays also have the comparison feature, as shown in Listing 1-23. By changing the list syntax to an array syntax, you can get the same code to perform the structural equality comparison. Listing 1-28 shows an example.

LISTING 1-28 An array comparison

```
let l1 = [| 1; 2; 3 |]
let l2 = [| 2; 3; 1 |]
printfn "l1 = l2? %A" (l1 = l2)
printfn "l1 < l2? %A" (l1 < l2)
```

Another interesting feature provided for working with arrays in F# is *slicing*. You use slicing to take a continuous segment of data from an array. The syntax for slicing is straightforward: `myArray.[lowerBound .. upperBound]`. Listing 1-29 shows how to use slicing.

LISTING 1-29 Slicing an F# array

```
// define an array with elements 1 to 10
let array0 = [| 1 .. 10 |]

// get slice from element 2 through 6
array0.[2..6]

// get slice from element 4 to the end
array0.[4..]

// get the slice from the start to element 6
array0.[..6]

// get all the elements (copy the whole array)
array0.[*]
```

Arrays are the same in both F# and C#. You do not have to reference to *Microsoft.FSharp.Core.dll* to expose an array from an F# library.

Pipe-Forward Operator

Before presenting the F# code used to rewrite the example from the beginning of this chapter in a functional style, I must first explain the pipe-forward operator (|>). If you're familiar with UNIX's pipeline, you can think of this operator as something similar. It gets the output from one function and pipes that output in as input to the next function. For example, if you have the functions *g(x)* and *f(x)*, the *f(g(x))* function can be written by using a pipe-forward operator, as shown in Listing 1-30.

LISTING 1-30 An F# pipe-forward operator

```
x |> g |> f    // equals to f(g(x))
```

The C# program at the beginning of this chapter focused on how to process each single element from 0 to 100 by iterating through the elements. If the element was an odd number, it was added to a predefined variable. Do we really have to think like this?

F# provides a way to think differently. Instead of thinking about how to process each single element, you can think about how to process the whole data set as a single element—a collection of data. The whole process can instead be thought about like this: after being given data from 0 to 100, I get a subset of the given data that contains only odd numbers, sum them up, and print out the result. The C# code to implement this logic is shown in Listing 1-31, as is the equivalent F# code. The use of the pipe-forward operator allows the F# code to become even more succinct and beautiful than the

already beautiful LINQ code. The pipe-forward operator brings the result from *Seq.sum* to *printfn*. Isn't that simple?

LISTING 1-31 A functional approach to solve the odd-number summary problem

C# code

```
var sum = dataFrom0to100
    .Where(n=>n%2!=0)    //filter out the odd number
    .Sum()                  //sum up

//output the result
Console.WriteLine("the sum of the odd number from 0 to 100 is {0}", sum);
```

F# code

```
seq { 0..100 }                          //given data from 0 to 100
|> Seq.filter (fun n -> n%2<>0)     //data subset contains only odd number
|> Seq.sum                               //sum them up
|> printfn "the sum of odd number from 0 to 100 is %A"   //print out the result
```

When the F# code is shown side by side with the C# equivalent, it's easy to tell that *Seq.filter* is a built-in function used to filter data and *Seq.sum* is a function used to sum up the elements in a provided sequence. Because *printfn*, which originally needs two parameters, gets its second parameter from the pipe-forward operator (|>), it takes only one explicitly provided parameter. *Seq* module functions are discussed in more detail in the "*Seq/List/Array* Module Functions" section.

From a coding experience and readability perspective, the functional way is much better than the imperative way. F#, as a functional-first language, shows this advantage very clearly. It can chain the functions together more naturally.

One headache for LINQ developers is the debugging of LINQ code. This would also be a headache for F# if FSI was not present. The FSI lets you execute some code to set up the test environment and then send the code that needs to be tested. In the previous example, if you are not sure if the filter gives you the right result, you can select the first two lines and send them to the FSI. After finishing one test, *Reset Interactive Session* is a convenient way to reset your environment. Isn't that nice!

If you're still digesting the pipe-forward operator, you can think of the parameter on the left side of the operator as the suffix to the end of the right part. The two statements shown next in Listing 1-32 are basically the same.

LISTING 1-32 Using the pipe-forward operator

```
mySeq |> Seq.length       // get the length of the sequence
Seq.length mySeq           // the same as the expression above with |>
```

Tip FSI is not only a good approach for debugging a program and running unit tests, it's also a quick way to check a function's definition. For example, you can type **Seq.filter** into the FSI window. FSI then shows you the function definition, which saves you the two seconds of going to the MSDN documentation.

The Sequence, List, and Array Module Functions

Now you must be wondering where someone can find functions like *Seq.filter* and *Seq.sum*. They are located inside three modules: Seq, List, and Array module. Module is a special way for organizing F# code that will be discussed later in this chapter. For the convenience of later discussion, we denote seq, list, and array as *collections*. The functions listed next are the most commonly used ones. Refer to MSDN document *http://msdn.microsoft.com/en-us/library/ee353413* for a complete function list.

length

It is easy to get the length of a list or an array by using the *length* function. The LINQ *Count* extension method provides the same functionality. An example is shown in Listing 1-33.

LISTING 1-33 The *length* function

```
let myList = [1..10]
let listLength = myList |> List.length  // listLength is 10
let myArray = [| 1..10 |]
let arrayLength = myArray |> Array.length //arrayLength is 10
let mySeq = seq { 1..10 }
let seqLength = mySeq |> Seq.length //seqLength is 10
```

Note Seq does have a length function, but keep in mind that a sequence can be of an infinite length. An infinite-length sequence can make many functions not applicable, unsafe, or both.

exists and *exists2*

Seq, list, and array all provide the same functions to check whether an element exists and to see whether two collections contain the same element at the same location. The *exists* function is used to check for a single element, and *exists2* is used for checking two collections. Listing 1-34 shows how to use *Seq.exists* and *Seq.exists2*.

LISTING 1-34 The *exists* and *exists2* functions

```
let mySeq = seq { 1..10 }
let mySeq2 = seq { 10..-1..1 }

// check if mySeq contains 3, which will make "fun n -> n = 3" return TRUE
if mySeq |> Seq.exists (fun n -> n = 3) then printfn "mySeq contains 3"

// more concise version to check if it contains number 3
if mySeq |> Seq.exists ((=) 3) then printfn "mySeq contains 3"

// check if two sequences contain the same element at the same location
if Seq.exists2 (fun n1 n2 -> n1 = n2) mySeq mySeq2 then printfn "two sequences contain
same element"
```

You might have trouble understanding the `((=) 3)` in the code from the previous example. Everything in F# is a function, and the equal sign (=) is no exception. If you want to see the equal sign definition, you can run the FSI code shown in Listing 1-35. The definition is as follows:

```
('a -> 'a -> bool)
```

This definition is a function function that takes an 'a and returns a function ('a -> bool). 'a is something not familiar. It is a type and will be determined by type inference which will be introduced later in this chapter. When an argument is provided to this function, it returns a new function. Back to our sample code of `((=) 3)`: the code generates a function that takes one argument and checks whether the passed-in argument is equal to 3.

LISTING 1-35 An equal function definition

```
let f = (=);;

val f : ('a -> 'a -> bool) when 'a : equality
```

You might be wondering, "What about `((>) 3)`? Does it equal `x > 3` or `3 > x`?" Good question! Again, let us ask FSI. Listing 1-36 shows the result. The first statement defines the function, and the second one passes *4* to the statement. If the parameter *4* is going to the left side of the equation, the final result should be `4>3 = TRUE`. Because the final result is FALSE, the *4* must be on the right side.

LISTING 1-36 An equal function with a fixed parameter

```
> let f = (>) 3;;         // define the function (>) 3
val f : (int -> bool)

> f 4;;                      // pass 4 into the function
val it : bool = false   // result is FALSE
```

Note Using = or > can make your code shorter. However, overuse can make your code less readable.

forall and *forall2*

The *forall* function can be used to check whether all of the elements in a collection meet certain criteria. The LINQ *All* extension method provides the same functionality. Listing 1-37 shows an example of how to use *Seq.forall*.

LISTING 1-37 The *forall* function

```
let myEvenNumberSeq = { 2..2..10 }

// check if all of the elements in the seq are even
myEvenNumberSeq |> Seq.forall (fun n -> n % 2 = 0)
```

Like the *exists2*, *forall2* provides functionality similar to *forall*, but it provides the functionality across two collections. If and only if the user function returns TRUE for the two-element pairs,*forall2* returns TRUE. Listing 1-38 shows this in action.

LISTING 1-38 The *forall2* function

```
let myEvenNumberSeq = { 2..2..10 }
let myEvenNumberSeq2 = { 12..2..20 }

if Seq.forall2 (fun n n2 -> n+10=n2) myEvenNumberSeq myEvenNumberSeq2 then printfn
"forall2 // returns TRUE"
```

find

The *find* function is more like the *First* extension method on *IEnumerable*. It raises *KeyNotFoundException* if no such element exists. See Listing 1-39.

LISTING 1-39 The *find* function

```
// use let to define a function
let isDivisibleBy number elem = elem % number = 0

let result = Seq.find (fun n -> isDivisibleBy 5 n)[ 1 .. 100 ]
printfn "%d " result    //result is 5
```

The *findIndex* function is designed to allow for a quick lookup of an element's index. You can find sample code in the MSDN documentation (*http://msdn.microsoft.com/en-us/library/ee353685*).

map

The *map* function is used to create a new collection based on a given collection by applying a specified function to each element in the provided collection. The LINQ *Select* extension method provides the same functionality. See Listing 1-40.

LISTING 1-40 The *map* function

```
let mySeq = seq { 1..10 }
let result = mySeq |> Seq.map (fun n -> n * 2)  // map each element by multiplying by 2
Seq.forall2 (=) result (seq { 2..2..20 })            // check result
```

filter

The *filter* function returns a new collection containing only the elements of the collection for which the given predicate returns TRUE. The LINQ *Where* extension method provides the same functionality. See Listing 1-41.

LISTING 1-41 The *filter* function

```
let mySeq = { 1..10 }
let result = mySeq |> Seq.filter (fun n -> n % 2 = 0)   //filter out odd numbers
printfn "%A" result
```

fold

The *fold* function aggregates the collection into a single value. Its definition from MSDN shows that *fold* applies a function to each element of the collection and threads an accumulator argument through the computation. The LINQ *Aggregate* extension methods perform the same functionality. Listing 1-42 shows an example that sums all elements in the given sequence.

LISTING 1-42 The *fold* function

```
let mySeq = { 1..10 }
let result = Seq.fold (fun acc n -> acc + n) 0 mySeq
printfn "the sum of 1..10 is %d" result                    //sum = 55
```

One application of the *fold* function is to get the length of a collection. The built-in function *length* supports only 32-bit integers. If the sequence length is a very big number, such as *int64* or even *bigint*, the *fold* function can be used to get the length. The following sample gets the sequence length in *bigint*. See Listing 1-43, which shows the code to accomplish this.

LISTING 1-43 The *fold* function to get a *bigint* length of a sequence

```
let mySeq = { 1..10 }
let length = Seq.fold (fun acc n -> acc + 1I) 0I mySeq
printfn "the bigint length of seq is %A" length
```

collect

The *collect* function applies a user-defined function to each element in the collection and joins results. One good application of this function is the LINQ *SelectMany* extension method. The *SelectMany* method flattens the hierarchy and returns all the elements in the second-level collection. The following sample first generates a list of lists and then combines all the lists. See Listing 1-44.

LISTING 1-44 The *collect* function

```
let generateListTo x = [0..x]

// generates lists [ [0;1]; [0;1;2]; [0;1;2;3] ]
let listOfLists = [1..3] |> List.map generateListTo

// concatenate the result
// seq [0; 1; 0; 1; 2; 0; 1; 2; 3]
let result = listOfLists |> Seq.collect (fun n -> n)
```

Tip If you can use the built-in *id* function, the last line can be rewritten as `Seq.collect id`.

append

This method takes two collections and returns a new collection where the first collection's elements are followed by the second collection's elements. The LINQ *Concat* function provides the same functionality. See Listing 1-45.

LISTING 1-45 The *append* function

```
// the following concatenates two arrays and generates
// a new array that contains 1;2;3;4;5;6
printfn "%A" (Array.append [| 1; 2; 3|] [| 4; 5; 6|])
```

Math Operations

In addition to the transformation functions, F# provides a rich set of functions that perform mathematical operations on collections:

- ***min*** and ***max*** The *min* and *max* functions are used to find the minimum or maximum value in a collection. These functions are just like the LINQ *min* and *max* functions. See Listing 1-46.

  LISTING 1-46 The *min* and *max* functions

```
let myList = [1..10]
let min = myList |> List.min    // min is 1
let max = myList |> List.max     // max is 10
```

- ***average*** The *average* function is used to get the mean of all elements in a collection. Because F# does not have implicit type conversion from integer to float, the following code generates an *int* and does not support *DivideByInt* operator, which means the operand has to be transformed to a data type that supports divide, such as *float* or *float32*. See Listing 1-47.

  LISTING 1-47 The *average* function used with integer

```
let myList = [1..10]

// does not compile because int does not support "DivideByInt"
let myListAverage = myList |> List.average
```

  You can use *map* to change the integer element into a *float* element. Again, the *float* is also a function that converts an integer to a *System.Double*. See Listing 1-48.

  LISTING 1-48 The *average* function in a float sequence

```
let myList = [1..10]

// the average is float type 5.5
let myListAverage = myList
                          |> List.map float
                          |> List.average
```

- ***sum*** The *sum* function returns the sum of elements in a collection. The *sum* result type depends on the input sequence element type. The example in Listing 1-49 showcases this.

  LISTING 1-49 The *sum* function

```
let mySeq = seq { 1..10 }

// sum is 55
let result = mySeq |> Seq.sum
```

zip and *zip3*

The *zip* function combines two sequences into a sequence of pairs. The *zip3* function, as its name suggests, combines three sequences into triples. The *zip* sample is shown in Listing 1-50. You can find a *zip3* sample on MSDN (*http://msdn.microsoft.com/en-us/library/ee370585.aspx*).

LISTING 1-50 The *zip* function

```
let myList = [ 1 .. 3 ]
let myList2 = [ "a"; "b"; "c" ]

// the zip result is [(1, "a"); (2, "b"); (3, "c")]
let result = List.zip myList myList2
```

rev

The *rev* function reverses the elements in a list or array. The sample code is shown in Listing 1-51.

LISTING 1-51 The *rev* function

```
let reverseList = List.rev [ 1 .. 4 ]

// print the reversed list, which is [4;3;2;1]
printfn "%A" reverseList
```

Note Seq does not have a *rev* function implemented.

sort

The *sort* function sorts the given list using *Operators.compare*. If the original element's order is preserved, this is called a *stable sort*. The *sort* function on Seq and List are stable sorts, while *Array.sort* is not a stable sort. See Listing 1-52.

LISTING 1-52 The *sort* function

```
let sortedList1 = List.sort [1; 4; 8; -2]

// print out the sorted list, which is [-2; 1; 4; 8]
printfn "%A" sortedList1
```

Convert to Seq/List/Array

Each type of collection has its unique usage. It is common to convert one type to the other type. It is easy to tell how to convert to a new type by looking at the functions. The sample code is shown in Listing 1-53.

- *Seq.toList* is used to convert a seq to a list.
- *Seq.toArray* is used to convert a seq to an array.
- *List.toSeq* is used to convert a list to a seq.
- *List.toArray* is used to convert a list to an array.
- *Array.toSeq* is used to convert an array to a seq.
- *Array.toList* is used to convert an array to a list.

LISTING 1-53 Some seq, list, and array conversion examples

```
// define a sequence
let mySeq = { 1..5 }

// define a list
let myList = [ 1.. 5 ]

// define an array
let myArray = [| 1..5 |]

// convert seq to a list
let myListFromSeq = mySeq |> Seq.toList

// convert seq to an array
let myArrayFromSeq = mySeq |> Seq.toArray

// convert list to an array
let myArrayFromList = myList |> List.toArray

// convert list to an seq
let mySeqFromList = myList |> List.toSeq

// convert array to a list
let myListFromArray = myArray |> Array.toList

// convert array to a seq
let mySeqFromArray = myArray |> Array.toSeq
```

Convert from Seq/List/Array

Unlike C#, which uses a *from* prefix, F# uses an *of* prefix to represent a function that converts from one collection type to another. See Listing 1-54 for examples.

- *Seq.ofList* is used to convert from a list to a seq.
- *Seq.ofArray* is used to convert from an array to a seq.
- *List.ofSeq* is used to convert from a seq to a list.
- *List.ofArray* is used to convert from an array to a list.
- *Array.ofSeq* is used to convert from a seq to an array.
- *Array.ofList* is used to convert from a list to an array.

LISTING 1-54 Some seq, list, and array examples of conversion

```
let mySeq = { 1..5 }
let myList = [ 1.. 5 ]
let myArray = [| 1..5 |]

// convert from list to a seq
let mySeqFromList = myList |> Seq.ofList

// convert from an array to a seq
let mySeqFromArray = myArray |> Seq.ofArray

// convert from a seq to a list
let myListFromSeq = mySeq |> List.ofSeq

// convert from an array to a list
let myListFromArray = myArray |> List.ofArray

// convert from a seq to an array
let myArrayFromSeq = mySeq |> Array.ofSeq

// convert from a list to an array
let myArrayFromList = myList |> Array.ofList
```

Of the three collection-related data structures discussed so far, sequence is likely the most interesting. A function that takes *seq<'T>* as an input parameter always works with list, array, set, and map in F#. Additionally, if you expose seq to a C# project, a reference to *Microsoft.FSharp.Core.dll* will not be required. Last, C#'s LINQ operations will work on sequences.

Table 1-3 lists all of the functions supported and a description of the performance of each.

TABLE 1-3 The collection of functions

Function	Array	List	Seq	Map	Set	Description
append	*O(m+n)*	*O(min(m,n))*	*O(1)*			Concatenates two collections.
add				*O(lgN)*	*O(lgN)*	Adds a new element, and returns a new collection.
average /averageBy	*O(n)*	*O(n)*	*O(n)*			Gets an average of all elements.
blit	*O(n)*					Returns a slice of the array.
cache			*O(n)*			Caches elements.
cast			*O(n)*			Converts an element to a specified type.
choose	*O(n)*	*O(n)*	*O(n)*			Chooses the element if not return None.
collect	*O(n)*	*O(n)*	*O(n)*			Applies a function to each element, and concatenates the results.
compareWith			*O(n)*			Compares element by element using a given function.
concat	*O(n)*	*O(n)*	*O(n)*			Concatenates two collections.
contains					*O(logN)*	Tests whether contains the specified element.
containsKey				*O(logN)*		Tests whether an element is in the domain of the map.
count					*O(n)*	Counts the number of elements in the set.
countBy			*O(n)*			Counts the generated key numbers. The key is generated from a function.
copy	*O(n)*		*O(n)*			Creates a copy of the collection.
create	*O(n)*					Creates an array.
delay			*O(1)*			Returns a sequence that is built from the given delayed specification of a sequence.
difference					*O(m*lgN)*	Returns a new set with an element in *set1* but not in *set2*.
distinct/ distinctBy			*O(1)*			Returns a new seq with removing duplicated elements.
empty	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*	Creates an empty collection.
exists	*O(n)*	*O(n)*	*O(n)*	*O(logN)*	*O(logN)*	Tests if any element satisfies the condition.
exists2	*O(min(n,m))*		*O(min(n,m))*			Tests whether any pair of corresponding elements of the input sequences satisfies the given predicate.
fill	*O(n)*					Sets the range of element to a specified value.

Function	Array	List	Seq	Map	Set	Description
filter	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*	Returns a new collection of elements satisfying the given criteria.
find	*O(n)*	*O(n)*	*O(n)*	*O(lgN)*		Returns the first element satisfying the given criteria.
findIndex	*O(n)*	*O(n)*	*O(n)*			Returns the first element index satisfying the given criteria.
findKey				*O(lgN)*		Evaluates the function on each mapping in the collection, and returns the key for the first mapping where the function returns TRUE.
fold	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*	Applies a function to each element of the collection, threading an accumulator argument through the computation.
fold2	*O(n)*	*O(n)*				Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation.
foldBack	*O(n)*	*O(n)*		*O(n)*	*O(n)*	Applies a function to each element of the collection, threading an accumulator argument through the computation.
foldBack2	*O(n)*	*O(n)*				Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation.
forall	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*	Tests whether all elements meet a condition.
forall2	*O(n)*	*O(n)*	*O(n)*			Tests whether all corresponding elements of the collection satisfy the given predicate pairwise.
get/nth	*O(1)*	*O(n)*	*O(n)*			Returns elements by a given index.
head		*O(1)*	*O(1)*			Returns the first element.
init	*O(n)*	*O(n)*	*O(1)*			Initializes the collection.
initInfinite			*O(1)*			Generates a new sequence which, when iterated, will return successive elements by calling the given function.
isProperSubset/ isProperSuperset					*O(M * log N)*	Tests whether the first set is a proper subset/superset of the second set.
isSubset/ isSuperset					*O(M * log N)*	Tests whether the first set is a subset/superset of the second set.
iter	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*	Applies a function to elements in the collection.
iter2	*O(n)*	*O(n)*	*O(n)*			Applies a function to a collection pairwise.

Function	Array	List	Seq	Map	Set	Description
length	*O(n)*	*O(n)*	*O(n)*			Returns the number of elements.
map	*O(n)*	*O(n)*	*O(1)*			Applies the function to each element.
map2	*O(n)*	*O(n)*	*O(1)*			Applies the function to a collection pairwise.
map3		*O(n)*				Creates a new collection whose elements are the results of applying the given function to the corresponding elements of the three collections simultaneously.
mapi	*O(n)*	*O(n)*	*O(n)*			Builds a new collection from a collection with the index passed in.
mapi2	*O(n)*	*O(n)*				Builds a new collection from two collections pairwise with an index passed in.
max/maxBy min/minBy	*O(n)*	*O(n)*	*O(n)*			Finds the max/min element.
maxElement/ minElement					*O(log N)*	Finds the max/min element in the set.
ofArray		*O(n)*	*O(1)*	*O(n)*	*O(n)*	Gets a collection from an array.
ofList	*O(n)*		*O(1)*	*O(n)*	*O(n)*	Gets a collection from a list.
ofSeq	*O(n)*	*O(n)*		*O(n)*	*O(n)*	Gets a collection from a seq.
pairwise			*O(n)*			Returns a sequence of each element in the input sequence and its predecessor, with the exception of the first element, which is only returned as the predecessor of the second element.
partition	*O(n)*	*O(n)*		*O(n)*	*O(n)*	Splits the collection into two collections, containing the elements for which the given predicate returns *true* and *false*, respectively
permutate	*O(n)*	*O(n)*				Makes a permutation of the collection.
pick	*O(n)*	*O(n)*	*O(n)*	*O(lg N)*		Applies the given function to successive elements, returning the first result where the function returns *Some*.
readonly			*O(n)*			Creates a new sequence object that delegates to the given sequence object.
reduce	*O(n)*	*O(n)*	*O(n)*			Applies a function to each element of the collection, threading an accumulator argument through the computation.
reduceBack	*O(n)*	*O(n)*				Applies a function to each element of the collection, threading an accumulator argument through the computation.

Function	Array	List	Seq	Map	Set	Description
remove				*O(lg N)*	*O(lg N)*	Removes the element.
replicate		*O(n)*				Creates a list of a specified length with every element set to the given value.
rev	*O(n)*	*O(n)*				Reverses the collection.
scan	*O(n)*	*O(n)*	*O(n)*			Applies a function to each element of the collection, threading an accumulator argument through the computation.
scanBack	*O(n)*	*O(n)*				Similar to *foldBack*, but returns both the intermediate and final results.
singleton			*O(1)*		*O(1)*	Returns a collection that contains only one element.
set	*O(1)*					Sets the element value.
skip			*O(n)*			Skips *n* elements.
skipWhile			*O(n)*			Skips an element when it meets the condition.
sort/sortBy	*O(N log N) Worst is O(N^2)*	*O(N lg N)*	*O(N lg N)*			Sorts the collection.
sortInPlace/ sortInPlaceBy	*O(N log N) Worst is O(N^2)*					Sorts the collection by mutating the collection in place.
sortWith	*O(N log N) Worst is O(N^2)*	*O(N lg N)*				Sorts the collection by the given function.
sub	*O(n)*					Gets a sub array.
sum/sumBy	*O(n)*	*O(n)*	*O(n)*			Gets the sum of a collection.
tail		*O(n)*				Returns collection without the first element.
take			*O(n)*			Takes *n* elements from the collection.
takeWhile			*O(1)*			Returns a sequence that, when iterated, yields elements of the underlying sequence while the given predicate returns *true*.
toArray		*O(n)*	*O(1)*	*O(n)*	*O(n)*	Returns an array from the collection.
toList	*O(n)*		*O(1)*	*O(n)*	*O(n)*	Returns a list from the collection.
toSeq	*O(n)*	*O(n)*		*O(n)*	*O(n)*	Returns a seq from the collection.
truncate			*O(1)*			Truncates the collection.
tryFind	*O(n)*	*O(n)*	*O(n)*	*O(lg N)*		Tries to find the element by the given function.

Function	Array	List	Seq	Map	Set	Description
tryFindIndex	*O(n)*	*O(n)*	*O(n)*			Tries to find the element index by the given function.
tryFindKey				*O(lg N)*		Tries to find the key by the given function.
tryPick	*O(n)*	*O(n)*	*O(n)*	*O(lg N)*		Returns the first result when the function returns *Some*.
unfold			*O(n)*			Returns a seq from a given computation.
union					*O(m * lg N)*	Returns a union of two collections.
unionMany					*O(n1 * n2...)*	Returns a union of collections.
unzip/unzip3	*O(n)*	*O(n)*	*O(n)*			Splits the collection into two collections.
windowed			*O(n)*			Returns a sequence that yields sliding windows of containing elements drawn from the input sequence.
zip / zip3	*O(n)*	*O(n)*	*O(n)*			Makes collections into a pair/triple collection.

There are two other primary collection data structures: map and set. *Map* is an immutable dictionary of elements; elements are accessed by key. *Set* is an immutable set based on binary trees; the comparison is the F# structural comparison function. For detailed information on these, refer to MSDN at *http://msdn.microsoft.com/en-us/library/ee353686.aspx* and *http://msdn.microsoft.com/en-us/library/ee353619.aspx*.

What Changed

The sequence operation is very much like the LINQ operation, and these operations will be used often in the rest of this book.

The functional style is very different from the imperative style. Imagine presenting the problem of adding all odd numbers between 0 and 100 to a person without any formal computer background. Most likely, that person would go about solving the problem in the same way that the functional approach presented. They would start by finding all of the odd numbers and then adding them up. This is simply a more straightforward approach that more closely resembles how people think about problems.

On the other hand, a person with a deeply rooted imperative software background will likely lean toward a solution with a *sum* variable, IF statement, and FOR loop. If you look at the history of programming languages—from binary coding to assembly language to modern-day programming

languages such as C#—the trend is that the programming language is more and more like a human language. The more the programming language resembles the human language, the more programmers will adopt it in their daily work and, consequently, make the language successful.

Other F# Types

Our next task is to refactor the F# code. During the refactoring process, more F# types are explored. These types are the foundation on which you will build when I introduce F# classes in Chapter 2, "Using F# for Object-Oriented Programming."

Defining Constants by Using Attributes

C# supports constants through the use of the *const* keyword. The designers of F# decided not to introduce many keywords and thus left them open for use as variable names. Instead, F# uses an attribute to define constant values. In F#, an attribute needs to be put between *[<* and *>]*. When I present more F# features, you will find F# uses more attributes than keywords when defining a data type. See Listing 1-55. In this example, *Literal* is an attribute that indicates to the F# compiler that *myConstant* is a constant.

LISTING 1-55 Defining a constant

```
[<Literal>]
let MyConstant = 99
```

F# uses some compiler tricks to replace a variable with the constant value, as you can see in Listing 1-56.

LISTING 1-56 Defining a constant for an upper limit

```
[<Literal>]
let upperLimit = 100

seq { 0..upperLimit }                //given data from 0 to 100
|> Seq.filter (fun n -> n%2<>0)      //data subset contains only odd numbers
|> Seq.sum                                  //sum them up
|> printfn "the sum of odd number from 0 to 100 is %A"  //print out the result
```

Enumerations

The enumeration type provides a way to define a set of named integral constants. Listing 1-57 shows how to define an enumeration type. Each field must be a unique value. Listing 1-58 shows how to use the enumeration value once it is defined.

LISTING 1-57 An F# enumeration definition

```
Enumeration definition using integers
type Card =
    |  Jack = 11
    |  Queen = 12
    |  King = 13
    |  Ace = 14


Enumeration definition using a binary integer format
type OptionEnum =
    | DefaultOption = 0b0000
    | Option1 = 0b0001
    | Option2 = 0b0010
    | Option3 = 0b0100
    | Option4 = 0b1000
```

Note Each item in the enumeration must have an integral value assigned to it; you cannot specify only a starting value.

LISTING 1-58 An access enumeration value

```
let option1 = OptionEnum.Option1
let opton1Value = int OptionEnum.Option1    //option1 value is integer 1
```

Like C#, F# does not allow integral type values to be directly set to an enumeration variable. Instead, a conversion with type is needed, as shown in Listing 1-59.

LISTING 1-59 Converting an integer to *OptionEnum*

```
let option1 = enum<OptionEnum>(0b0001)
```

The bitwise operation can use the optimized algorithm to check the odd number. You can use enumeration and a bit operation, as shown in Listing 1-60. F# supports five bitwise operators, which are listed in Table 1-4.

LISTING 1-60 Using a bitwise operation

```
type EvenOddFlagEnum =
    |  Odd = 0x1
    |  Even = 0x0

seq { 0..100 }                              //given data from 0 to 100
|> Seq.filter (fun n -> n &&& (int EvenOddFlagEnum.Odd) <> 0 ) //data subset contains only
odd numbers
|> Seq.sum                                 //sum them up
|> printfn "the sum of odd number from 0 to 100 is %A"   //print out the result
```

TABLE 1-4 F# bitwise operators

Bitwise operation	C# operation	F# operation	Expected result
bitwise AND	0x1 & 0x1	0x1 &&& 0x1	0x1
bitwise OR	0x1 | 0x2	0x1 ||| 0x2	0x3
bitwise XOR	0x1 ^ 0x1	0x1 ^^^ 0x1	0
left shift	0x2 << 1	0x2 <<< 1	0x4
right shift	0x2 >> 1	0x2 >>> 1	0x1

Tuples

A *tuple* is a grouping of unnamed but ordered items. The items in a tuple can have different types. Listing 1-61 demonstrates how to define a tuple.

LISTING 1-61 A tuple definition

```
// Tuple of two integers: int * int
( 1, 5 )

// Tuple with three strings: string * string * string
( "one", "two", "three" )

// mixed type tuple: string * int * float
( "one", 1, 2.0 )

// Tuple can contain non-primitive type values
( a + 1, b + 1)

//tuple which contains tuples: (int * int) * string
((1,2), "good")
```

The tuple introduces an interesting phenomenon. How many parameters are there in the function *F(1,2,3)*? *Three* is the wrong answer. The function *F* takes only one parameter, whose type is a tuple, and the tuple is of type *int*int*int*. A function that takes three parameters is defined as *F 1 2 3*. A tuple can be used to group parameters together to make sure that related parameters are always passed into a function. For example, if you always need to pass the first name and last name together into a function named *g*, it is better to declare *g* like this:

```
// indicate first name and last name be provided together
g (firstName, lastName)
```

That way is better than the following approach:

```
// indicate first name and last name can be passed separately
g firstName lastName
```

The tuple supports structural equality. This means that the code shown in Listing 1-62 returns *true*. This concise syntax can make your code more readable.

LISTING 1-62 The tuple structural equality

```
(1,2) = (1,2)
(1,2) = (1, 1+1)
(1,2) < (2, 4)
```

Forming a tuple can be as simple as putting all elements between a pair of parentheses, although parentheses are optional if omitting them does not introduce confusion. Retrieving the elements requires two functions: the *fst* and *snd* functions are used to retrieve the first and the second elements, respectively, from a tuple. See Listing 1-63.

LISTING 1-63 The *fst* and *snd* functions in a tuple

```
// define tuple without parentheses
let a = 1, 2
let b = 1, "two"

// get first element in a tuple
let isOne = fst a

// get second element in a tuple
let isTwo = snd b
```

If the third or fourth element is needed, the functions defined in Listing 1-64 can be used. The underscore (_) stands for a placeholder where the value can be ignored. You'll find the underscore used in other places as well. It's a way to tell the F# compiler that this is something you don't care about.

LISTING 1-64 The third and fourth functions in a tuple

```
// get the third value of the triple
let third (_,_,c) = c

// get the fourth value of the quadruple
let fourth (_,_,_,d) = d

third (1,2,3) // return 3
fourth (1,2,3,4) //return 4
```

There is another way to retrieve the embedded elements without using these functions. Listing 1-65 shows that the F# compiler can figure out that the element in *l* is a triple. The variables *a*, *b*, and *c* are used to hold the element values in the triple.

LISTING 1-65 Use let and iterating through a triple list and

```
let tripleVariable = 1, "two", "three"
let a, b, c = tripleVariable

let l = [(1,2,3); (2,3,4); (3,4,5)]
for a,b,c in l do
    printfn "triple is (%d,%d,%d)" a b c
```

Functions

If you want to refactor the F# code in the conversion task, one possible way is to define a function that checks whether a given number is odd. Defining a function is a simple task for an experienced C# developer. Most likely, you already figured out how to write an F# function. One thing I want to point out is that F# does not have a *return* keyword. As a result, the value from the last expression is always the returned value. The following function defines an operation that increments a given integer by one:

```
let increaseOne x = x + 1
```

If you run the code in FSI, the result shows that the function takes an integer as input and returns an integer as output. The result might look strange, but it's still understandable:

```
val increaseOne : int -> int
```

Things start to get more interesting when you try to define a *sum* function that takes two parameters. Listing 1-66 shows the code definition and its execution result in FSI.

LISTING 1-66 Defining a *sum* function that takes two parameters

```
// define a sum function that takes two parameters
let sum x y = x + y

// FSI execution result
val sum : int -> int -> int
```

sum is a curried function. You can envision this function as something that takes *x* as a parameter and returns a new function that takes a parameter called *y*. The beauty of this approach is that it enables you to define a new function based on an existing partially applied function. For example, the *increaseOne* function is really a *sum* function where *y* is always *1*. So you can rewrite the *increaseOne* function as follows:

```
let increaseOne2 = sum 1
```

If you give *4* to this *increaseOne2* function, the return value will be *5*. It's like passing *4* and *1* into the *sum* function.

```
increaseOne2 4    // returns 5
sum 4 1             // pass 4 and 1 into the sum function and yield 5
```

Note You might be tempted to invoke this *sum* function with the following syntax: `sum(4,1)`. If you do this, an error message will point out that the expression expects a type of *int*int*. However, *(4,1)* is an F# type called a *tuple*. The code *sum(4,1)* is trying to pass a *tuple* type into *sum*, which confuses the F# compiler.

Now you can define your own function and refactor the conversion-task F# code, as shown in Listing 1-67.

LISTING 1-67 Defining an F# function in the odd-number sum program

```
// define an enum
type EvenOddFlagEnum =
            |  Odd = 0x1
            |  Even = 0x0

// define a function to check if the given number is odd or not
let checkOdd n = n &&& (int EvenOddFlagEnum.Odd) <> 0
```

```
seq { 0..100 }                          //given data from 0 to 100
|> Seq.filter checkOdd                 //data subset contains only odd numbers
|> Seq.sum                               //sum them up
|> printfn "the sum of odd number from 0 to 100 is %A"   //print out the result
```

As the code is getting cleaner, somebody might notice the *Seq.filter* function takes a function as input. Yes, F# treats values and functions the same. If a function takes a function as input or returns a function, that function is called a *higher-order function*. In Listing 1-67, *Seq.filter* is a higher-order function. The *Seq.filter* function provides the skeleton of a filter algorithm. You can then implement your special filter mechanism and pass your function into the skeleton function. Higher-order functions provide an extremely elegant way to reuse code. The higher-order function is a light-weight Strategy design pattern.

We have finished our refactoring work. You might already be eager to see the class definition in F#, but before I start introducing how F# handles object-oriented concepts such as classes, we need to spend a little more time on some basics.

Recursive Functions

The keyword *rec* is needed when defining a recursive function. Listing 1-68 shows how to use the *rec* keyword to define a function to compute Fibonacci numbers.

LISTING 1-68 Using the *rec* keyword to define a recursive function

```
let rec fib n =
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

Note This recursive version does not use a tail call (which is defined later in this section), so this version can generate a stack overflow error.

Sometimes a function is *mutually recursive* because the calls form a circle. Listing 1-69 shows the mutually recursive *Even* and *Odd* functions. The F# variable and function resolution is from top to bottom and from left to right. All the functions and variables must be declared first before they can be referenced. In this case, you have to use the *and* keyword to let the compiler know.

LISTING 1-69 A mutually recursive function definition

```
let rec Even x =              //Even calls Odd
   if x = 0 then true
   elif x = 1 then false
   else Odd (x - 1)
and Odd x =                     //Odd calls Even
   if x = 1 then true
   elif x = 0 then false
   else Even (x - 1)
```

In C# code, the stack overflow exception can happen when you use a recursive function. A small amount of memory is allocated when doing each recursive function call, and this allocation can lead to a stack overflow exception when a large amount of recursion is needed. A *tail call* is a function call whose result is immediately treated as the output of the function. Thanks to the tail call in F#, the F# compiler generates a tail-call instruction to eliminate the stack overflow problem when possible. Figure 1-15 shows the project setting's Build tab, which is where you can select (or deselect) the Generate Tail Calls check box.

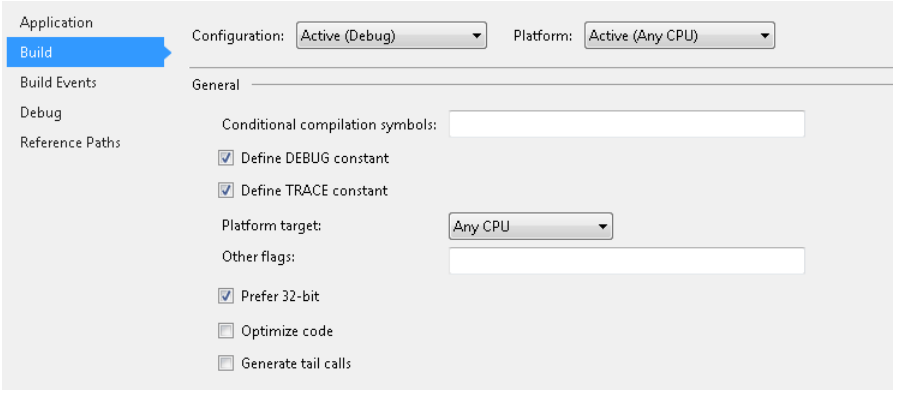

FIGURE 1-15 The Generate Tail Calls check box on the project setting's Build tab

Pipe/Composite Operators

Because functional programming languages see no difference between data and functions, you might be wondering if there are any operators specifically for functions. The pipe-forward operator was introduced already. The real function operators are forward and backward composite operators. See Listing 1-70.

LISTING 1-70 Composing two functions into a new function using the forward composite operator

```
let f0 x = x * 2
let f1 x = x + 7

//composite f and g to get a new function
// g(x) = x + 11
// the f0 function is executed first and then f1
let g = f0 >> f1

// result is 2*2+7 = 11
let result = g 2
```

The forward composite operator executes the function from left to right. The backward composition operator executes the function in the opposite direction, from right to left. Listing 1-71 demonstrates these two operators.

LISTING 1-71 Composing two functions into a new function using the backward composite operator

```
let f0 x = x * 2
let f1 x = x + 7

// composite f and g to get a new function
// g(x) = x + 11
// the f1 function is executed first and then f0
let g = f0 << f1

// result is (2+7)*2 = 18
let result = g 2
```

Similar to the backward composite operator, F# also has a backward pipe operator (<|). The backward pipe operator uses left associativity. The code `f <| g <| x` is parsed as `(f <| g) <| x`. It is not parsed as `f <| (g <| x)`, which is equivalent to `x |> g |> f`. See Listing 1-72.

LISTING 1-72 Comparing forward and backward pipe operators

Pipe-forward operator

```
let f x = x * 2
let g x = x + 5

// forwardPipeResult is 3*2 + 5 = 11
let forwardPipeResult = 3 |> f |> g
```

Pipe-backward operator

```
let f x = x * 2
let g x = x + 5

// forwardPipeResult is 2*(3 + 5) = 16
let forwardPipeResult = f <| (g <| 3)
```

Note Pipe operators and composite operators do not affect function execution performance. You can choose either of them according to your preference.

If you are curious about how to allow a C# method to hook into the pipe operation, Listing 1-73 shows an example. The code takes an integer list as input, converts each item into a string, and joins them by using "*" as a separator. Because *String.Join* has many overloaded functions, you have to tell the compiler which overload format you prefer. We use *FuncConvert.FuncFromTupled* to convert the .NET function to a curried format, which is easier to work with when using the pipe operator.

LISTING 1-73 Assigning a function and converting function to a curried format

```
let input = [ 1..10 ]

// assign String.Join to join with the string*string list as function signature
let join : string*string list -> string = System.String.Join

// convert join to curry format
let curryFormatJoin = FuncConvert.FuncFromTupled join

input
|> List.map (fun number -> string(number))  //convert int to string
|> curryFormatJoin "*"      // join the string list to a string string using "*"
```

In functional programming, the function is a first-class citizen. Some readers might try to explore mathematical operations with functions. The pipeline operator and the composite operator can be viewed as the *add* operation, which combines two functions. Listing 1-74 does not compile because you cannot compare two functions.

LISTING 1-74 A function does not support the equality comparison

```
let f1 = fun () -> ()
let f2 = fun () -> ()
f1 = f2  //does not compile
```

Unit Types

If a C# method does not have a return value, you use *void* to tell the compiler. In F#, the *unit* keyword is used to accomplish this. There are two ways to tell the compiler that a function returns *void*:

- Specify the return type in the function. See Listing 1-75.

  LISTING 1-75 A function with return unit type and input parameter type specified

```
//define a function return unit
let f (x) : unit = ()

//define a function with int parameter and unit return type
let f (x:int) : unit = ()
```

- Make the "*()*" be the last statement. As I mentioned earlier in this chapter, F# functions do not use the *return* keyword. Instead, F# takes the type from the last statement that the function returns. So the following function returns *unit* as well. You can use the *ignore* operator to throw away the computation result, which makes the function return nothing.

```
// define a function returns unit
let f x = () // f : 'a -> unit

// use ignore to throw away the keyboard input and f2 returns unit
let f2 () = System.Console.ReadKey() |> ignore
```

Note When a function can be passed in as a parameter to another function, the passed-in function is of type *Microsoft.FSharp.Core.FSharpFunc*. The function, which is passed in as a parameter, can be called by calling the *Microsoft.FSharp.Core.FSharpFunc.Invoke* method. When this is the case, the Invoke a function returning *unit* actually returns the *Microsoft.FSharp.Core.Unit* type, which is not void at all. Fortunately, the F# Compiler handles this so that you do not have to be aware of the subtle difference when coding.

Type Aliases

Another nice F# feature allows you to give a type an alias. Listing 1-76 shows how to give the built-in system type *int* a different name. This feature is more like *typedef* in C++. From the execution result, you can see that the *int* type can now be referenced by using *I*.

LISTING 1-76 A type alias and the FSI execution result

```
> type I = int
let f (a:I) = a + 1;;

type I = int
val f : I -> I
```

Type Inferences

At this point, you might still be wondering why you don't have to specify type information when writing the F# code shown in Listing 1-13. You might get the wrong impression that F# is a scripting language or a dynamic language, which usually do not emphasize the type. Actually, this is not true; F# is a strongly typed language. F# provides the ability to leave off type definitions through a feature called *type inference*, which can identify type information based on how code is used. Listing 1-77 shows the function *f* and how its definition is shown in FSI. The type for *s* can be determined by the function call *System.String.IsNullOrEmpty*, which takes a string and returns a Boolean. This is how F# identifies the type information.

LISTING 1-77 A type inference sample

```
> let f s = System.String.IsNullOrEmpty s;;

val f : string -> bool
```

Type inference works great, but it will need help sometimes, such as when processing overloaded methods, because overloaded methods distinguish themselves from each other by parameter types. For the example, in Listing 1-78, the *LastIndexOf* method can get both *char* and *string* as input. You did not expect that the compiler could read your mind, did you? Because the variable *ch* can be either the *string* or *char* type, you have to specify the type for the variable *ch*. If you do not tell the compiler the parameter type, there is no way for F# to figure out which overloaded method to use. The correct code is shown in Listing 1-79.

LISTING 1-78 The type inference when using overloaded methods

```
> let f ch = "abc".LastIndexOf(ch);;

  let f ch = "abc".LastIndexOf(ch);;
  -----------^^^^^^^^^^^^^^^^^^^^

stdin(10,12): error FS0041: A unique overload for method 'LastIndexOf' could not be
determined based on type information prior to this program point. A type annotation may
be needed. Candidates: System.String.LastIndexOf(value: char) : int, System.String.
LastIndexOf(value: string) : int
```

LISTING 1-79 The type inference with a type specified

```
> let f (ch:string) = "abc".LastIndexOf(ch);;

val f : string -> int
```

It's good to understand how type inference process work. It is performed from top to bottom and from left to right. It does not start from the program's *main* function. So the code in Listing 1-80 does not compile. The string type cannot be inferred from the *str.Length* because there are tons of types in .NET that have a *Length* property.

LISTING 1-80 The type inference from top to bottom

```
let f str = str.Length  // str type cannot be determined

[<EntryPoint>]
let main argv =
    let result = f "abc"
    0 // return an integer exit code
```

Note It's a best practice to write the code as you go and let the F# compiler figure out the type for you. If there is an error, look at the code above the line that the error is pointing to. Those errors can often be easily fixed by providing type information. Type inference is processed from top to bottom, which could not be the order of how code is executed. The program entry point is usually located at the end of the file, but decorating the variable at entry point does not help.

IntelliSense uses a different way to provide type information for users. Therefore, sometimes IntelliSense shows the type information but the code will not compile. A simple code example is shown next. The type for *str* on the second line is unknown, although IntelliSense shows that *str* is of the *System.String* type.

```
let f str =
    let len = str.Length // str type is unknown
    str + "aa"
```

Type inference tends to make code more general. This simplicity not only saves you typing time, it also improves the readability by making the code more intuitive and friendly, as I think you can see in Listing 1-81.

LISTING 1-81 F# and C# side by side

F# code

```
let printAll aSeq =
    for n in aSeq do
        <your function call>
```

C# equivalent

```
public static void printAll<a>(IEnumerable<a> aSeq)
{
    foreach (a i in aSeq)
    {
        <your function call>
    }
}
```

If you're not convinced yet, Listing 1-82 demonstrates how to use type inference and a tuple to create a general *swap* function. The function can handle any data type. If you try to use C# to implement the same function, you'll find the F# code is more clean and elegant. If you use Visual Studio or MonoDevelop, type information can be shown when hovering over the function.

LISTING 1-82 A tuple and *swap* function

```
// swap a and b
let swap(a,b) = (b,a)
```

Interop and *Function* Parameters

Some F# users start to use F# as a library authoring tool. The F# library is then referenced and invoked from some C# library or application. Because F# is a .NET language, using *interop* with other .NET languages is seamless. Adding a reference to an F# project is exactly same as adding one to a C# project. The only thing that needs to be explained here is how to add a reference to FSI:

1. Use *#r* to reference a DLL.
2. Open the namespace.

Listing 1-83 shows how to reference *System.Core.dll*, open the *System.Collections.Generic* namespace, and use the *HashSet<T>* type.

LISTING 1-83 Creating a *HashSet* in an F# script file or in FSI

```
#r "System.Core.dll"
open System.Collections.Generic
let a = HashSet<int>()
```

Note F# does not need *new* to instantiate an object. However, the *new* keyword is recommended when the class type implements the *IDisposable* interface.

Compared to C#, there is little difference in the approach for invoking a .NET method in F#. Here's how to use a .NET method to convert a string to an integer:

```
System.Convert.ToInt32("234")
```

Passing a tuple to a C# function seems to work well until you have a function that requires an *out* parameter. Tuples do not provide a way to say that a value is *ref* or *out*, but a tuple does help to solve this problem. If you have a C# function like the following

```
public int CSharpFunction(int k, out int n)
```

the function actually returns two values to the invoker. So the solution is to declare a tuple to hold the returned values:

```
let returnValue, nValue = CSharpFunction(1)
```

This approach works for the method, which is not externally declared. A call to an externally declared method requires the *&* operator. Listing 1-84 shows how to handle an *out* parameter in an *extern* function.

LISTING 1-84 Using F# to invoke a C# *extern* function that has an *out* parameter

C# function definition

```
[DllImport("MyDLL.dll")]
public static extern int ASystemFunction(IntPtr inRef, out IntPtr outPtr);
```

Define and invoke the function in F#

```
[<System.Runtime.InteropServices.DllImport("something.dll")>]
extern int ASystemFunction(System.IntPtr inRef, System.IntPtr& outPtr);

let mutable myOutPtr = nativeint 1
let n = ASystemFunction (nativeint 0, &myOutPtr)
```

If you want to expose an *out* parameter to a C# project, you need a special attribute named *System.Runtime.InteropServices.Out*. By decorating a parameter with this attribute in F#, the C# invoker can see that the parameter is intended to be an *out* parameter. See the sample in Listing 1-85. The *byref* keyword is used to declare the variable as being passed in as a reference type.

LISTING 1-85 Using F# to expose an *out* parameter to C#

```
let f ([<System.Runtime.InteropServices.Out>]a: int byref) =  a <- 9
```

I covered the *out* parameter, so now let's shift our attention to *ref*. It's easy to handle the C# *ref* definition as well. Listing 1-86 shows how to handle *ref* parameters in F#. There is a *ref* keyword in

the sample code, which I'll introduce in Chapter 6, "Other Unique Features." For now, think of it as a reference variable.

LISTING 1-86 Using F# to invoke a C# function that has a *ref* parameter

C# function definition

```
namespace CSharpProject
{
    public class Chapter1Class
    {
        public static int ParameterFunction(int inValue, ref int refValue)
        {
            refValue += 3;
            return inValue + 7;
        }

        public static void ReturnSample(out int x)
        {
            x = 8;
        }
    }
}
```

F# code invoking the C# function

```
//declare a mutable variable
let mutable mutableValue = 2

// declare a reference cell
let refValue = ref 2

// pass mutable variable into the function
let refFunctionValue = CSharpProject.Chapter1Class.ParameterFunction(1, &mutableValue)

// pass reference cell into the function
let refFunctionValue2 = CSharpProject.Chapter1Class.ParameterFunction(1, refValue)

// out parameter
let mutable x = Unchecked.defaultof<_>
CSharpProject.Chapter1Class.ReturnSample(&x)
```

If you want to expose a *ref* parameter from an F# function to a C# project, removing the *Out* attribute from the function definition will do the trick. The use of the *byref* keyword indicates that this is a reference parameter. The code is shown in Listing 1-87.

LISTING 1-87 Exposing a *ref* parameter to a C# project

```
let f (a: int byref) =  a <- 9
```

Module, Namespace, and Program Entry Points

Now that the small F# program is clean, you might want to build an executable. Listing 1-1 is a code snippet, and you need to use Visual Studio to write, debug, and publish a product. This section will cover how to create an F# console application using Visual Studio. Figure 1-16 shows how to create an F# console application.

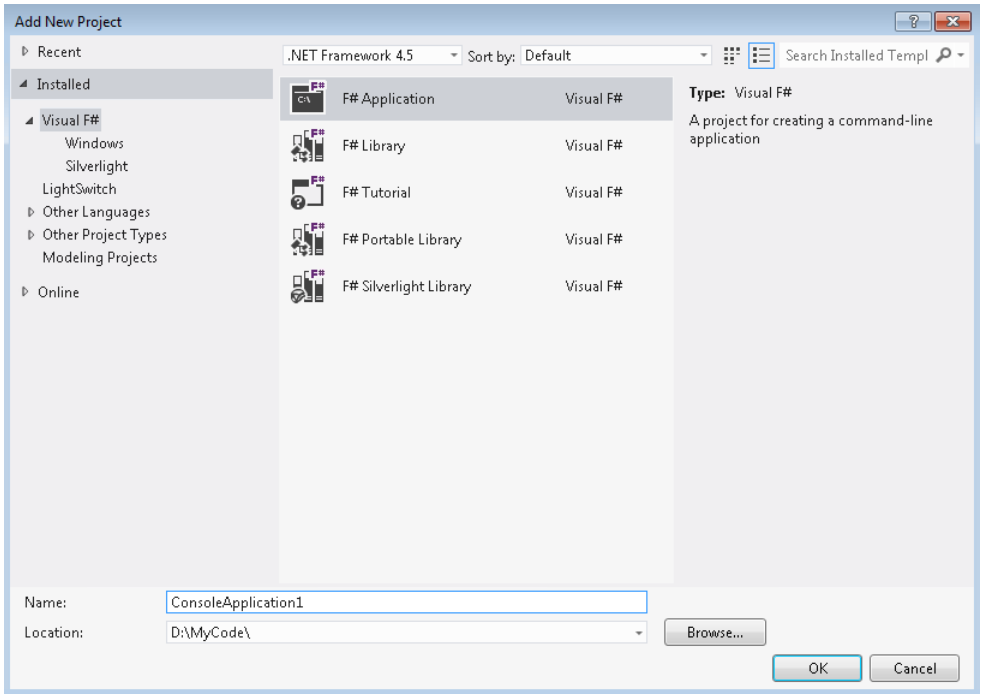

FIGURE 1-16 Creating an F# console application

The default Program.fs file contains a *main* function with an *EntryPoint* attribute; see Figure 1-17. Actually, F# does not need an *entry* function to be defined. For a single file project, the content is executed from the first line to the last.

```
// Learn more about F# at http://fsharp.net
// See the 'F# Tutorial' project for more help.

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

FIGURE 1-17 The default *main* function for an F# console application

If using the *EntryPoint* attribute is the preferred way, the function must meet the following criteria:

- The function must be decorated with the *[<EntryPoint>]* attribute.
- The function must be in the last file in the project.
- The function must return an integer as exit code.
- The function must take a string array, *string[]*, as input.

Tip In an F# DLL, modules are initialized by a static constructor. This guarantees the initialization occurs before any of the module's values are referenced. On the other hand, in an F# executable, the initialization is performed in the application's entry point, which is defined by the *EntryPoint* attribute. It is recommended that you explicitly define an entry point for an F# executable by using *EntryPoint*.

When you add the F# item in your console project, it can immediately make your code unable to compile. The newly added item is the last item in the project system. This breaks the second requirement in the preceding list. You can use the context menu—see Figure 1-18—or Alt+Up/Down Arrow keys, if the development profile is set to F#, to move your item up and down the project system.

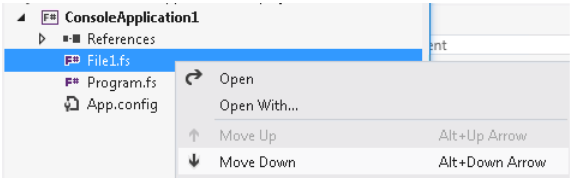

FIGURE 1-18 The context menu for moving an item up and down

It's always a good coding practice to divide the code into different units when the project has multiple files. F# provides the *module* or *namespace* to divide the code into small units. They have subtle differences, which I will discuss later in this chapter.

If you delete all content in the Program.fs file and paste the code shown in Listing 1-33, which does not have a module definition, the F# compiler will create an anonymous module. Its name is the file name with the first letter capitalized. For example, if your code is in file1.fs, the module name will be File1. You can also explicitly provide module names at the first line of the file, as shown in Listing 1-88.

LISTING 1-88 Module definition

```
module MyModule

printfn "Hello World!"
```

A module can contain other modules, type definitions, expressions, values, or any combination of those, but namespaces cannot be inside a module. The nested module needs to be indented and must have an equal sign (=) as a suffix. See Listing 1-89.

LISTING 1-89 Defining a nested module

```
module Print

// general print function
let print x = printf "%A" x

// define sub module NumberPrint
module NumberPrint =
    // function to print int
    let printInt = printf "%d"

    // function to print float
    let printFloat = printf "%f"

// define sub module StringPrint
module StringPrint =
    // function to print string
    let printString = printf "%s"

    // function to print char
    let printChar = printf "%c"

// define sub module with new type
module NewTypes =
    // define a 2D point
    type Point2D = float32*float32

// invoke print functions
NumberPrint.printFloat 4.5
NumberPrint.printInt 2
StringPrint.printChar 'c'
StringPrint.printString "abc"
```

If you want a scope to hold the functions and variables, the namespace and module are the same from F#'s point of view. If the F# project is going to be a library opened from a C# project, the namespace is a better choice. The module is a static class when it's viewed from the C# side. By using a module name and function name together, you can reference a function defined in a different module. If you want to reference a function inside a module, you can use the *open* keyword to open that module or decorate the module with the *AutoOpen* attribute, which can open the module automatically, essentially causing the function to be placed in the global namespace. See Listing 1-90.

LISTING 1-90 Opening a module

```
module Print

// general print function
let print x = printf "%A" x

// auto open  NumberPrint module
[<AutoOpen>]
module NumberPrint =
    // function to print int
    let printInt = printf "%d"

    // function to print float
    let printFloat = printf "%f"

module StringPrint =
    // function to print string
    let printString = printf "%s"

    // function to print char
    let printChar = printf "%c"

// invoke print functions
printFloat 4.5
printInt 2

// use module name to reference the function defined in the module
StringPrint.printChar 'c'
StringPrint.printString "abc"

// open StringPrint module
open StringPrint

printChar 'c'
printString "abc"
```

Note The *open* statement must be located somewhere after the module is defined.

The *RequireQualifiedAccess* attribute indicates that a reference to an element in that module must specify the module name. Listing 1-91 shows how to decorate the attribute on a module and reference a function within.

LISTING 1-91 Using the *RequireQualifiedAccess* attribute

```
[<RequireQualifiedAccessAttribute>]
module MyModule =
    let f() = printfn "f inside MyModule"

//reference to f must use MyModule
MyModule.f()
```

Modules can be extended by creating a new module with the same name. All the items—such as functions, types, expressions, and values—will be accessible as long as the new module is opened. See Listing 1-92. To invoke the extended method, you can open the namespace *MyCollectionExtensions*. Listing 1-93 opens the namespace, and the extension method is shown.

LISTING 1-92 Extending a module

```
Namespace MyCollectionExtensions

open System.Collection.Generic

// extend the array module by adding a lengthBy function
module Array =
    let lengthBy filterFunction array =
        array |> Array.filter fiterFunction |> Array.length
```

Note A module inside a namespace also needs an equal sign (=) as a suffix.

LISTING 1-93 Invoking the function in the extended module

```
open MyCollectionExtensions

let array = [| 1..10 |]
let evenNumber = array |> Array.lengthBy (fun n -> n % 2 = 0 )
```

As mentioned previously, if the F# code needs to be referenced from other .NET languages, a namespace is the preferred way to organize the code. Namespaces can contain only modules and type definitions. You cannot put namespaces, expressions, or values inside of a namespace. See Listing 1-94.

LISTING 1-94 Using a namespace

```
namespace MySpace

// define a 3D point
type Point3D = Point of float32 * float32 * float32

// define module inside namespace
module HelloWorld =
    let msg = @"Hello world"
    printfn "%s" msg

// the following line won't compile because a namespace cannot contain value or expression
// let a = 2
```

Note Namespaces are not supported in FSI because it's difficult to find out where the namespace ends in FSI.

Empty files do not compile in a multifile project. Source code must have either a namespace or a module at the beginning if the project contains multiple files.

CHAPTER 3

F# and Design Patterns

I constantly hear people say that F# is a cool cutting-edge technology, but they do not know how to use it. In previous chapters, I showed how C# developers can pretty much map their existing imperative, LINQ, and data-structure knowledge to F#. However, this is not enough know-how to design or implement a component or a system. In this chapter, I use well-known design patterns to introduce performing system design by using F#. The samples in this chapter use unique F# language features to implement well-known design patterns. These samples will help you start to think of F# as something other than a niche language.

I do not see a huge difference between computer language and human language. Both languages are used to convey human thinking, only the audiences are different. One is the computer, and the other is a human. If you want to master a language and use it to write a beautiful article, having knowledge of only the basic words of that language would definitely not be enough. Likewise, if people really want to use F# fluently in their daily programming work, they need to know more than how to write a float type and a FOR loop.

In this chapter, a number of design patterns are implemented in F#. These implementations should help you gain more insight about how our team designed the language and, consequently, how to use these features to solve system-design problems. Ultimately, my goal is to help you start to really think in F# terms.

There are some design patterns that are easily implemented with more advanced F# language features, such as F# object expressions. I am not going to discuss every aspect of these features. More detailed information about these special language features will be presented in Chapter 5, "Write Your Own Type Provider." If any aspects of this chapter are not clear, I encourage you to refer to Chapter 5, where F# unique features are introduced in detail.

Using Object-Oriented Programming and Design Patterns

Like many well-studied concepts, *design pattern* has many definitions. In this book, I borrow the definition from the Wikipedia page on the topic (*http://en.wikipedia.org/wiki/Software_design_pattern*). My quick definitions of the design patterns in this chapter are also largely based on Wikipedia.

The design pattern is the reusable solution template for a problem. It can speed up the development process by providing tested, proven development paradigms. The effective software design requires considering problems that may not become obvious until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

From the preceding statements, you can see that design patterns are not necessarily tied to specific languages or programming paradigms. Given that the object-oriented programming (OOP) paradigm is the most used, most design-pattern implementations and discussions are based on languages that target OOP—for example, C#. Some people from the functional programming community have suggested that design patterns are merely a means to address flaws in OOP languages. I will not go into the details of this topic; instead, I will cover how to use F# to implement design patterns.

First, I'll cover three basic concepts in programming languages that primarily target OOP:

- *Encapsulation* is a construct that facilitates the bundling of data with methods (or other functions) that operate on that data.
- *Inheritance* is a way to compartmentalize and reuse code. It creates a subtype based on an existing type.
- *Polymorphism: subtype polymorphism*, which is almost universally called just *polymorphism* in the context of object-oriented programming, is the ability to create a variable, a function, or an object that has more than one form.

The typical C# implementations of design patterns often use all three of these concepts. In the rest of the chapter, you will see how F# can use both OOP and functional features to implement most common design patterns.

Before demonstrating these design patterns, I'd like to remind you that a design pattern can have more than one implementation. Each of the implementations in the following examples show different F# language features in practice. Additionally, they provide a better way to apply F# in component or system design than what would be achieved by simply porting over a C# implementation.

Working with F# and Design Patterns

Let's start by looking at some of the design patterns that will be discussed in this chapter along with the definitions of each. Note that the following definitions are from an OOP perspective, so the definitions occasionally still use object-oriented terminology:

- The *chain of responsibility pattern* avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. It chains the receiving objects and passes the request along the chain until an object handles it.

- The *decorator pattern* attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- The *observer pattern* defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- The *proxy pattern* provides a surrogate or placeholder for another object to control access to it.
- The *strategy pattern* defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern lets the algorithm vary independently from clients that use it.
- The *state pattern* allows an object to alter its behavior when its internal state changes.
- The *factory pattern* lets a class defer instantiation to subclasses.
- The *adapter pattern* and *bridge pattern* are both used to convert the interface of a class into another interface. The adapter pattern lets classes work together that couldn't otherwise because of incompatible interfaces. If we don't focus on interfaces or classes, we can rephrase the definition to a shorter one: These are patterns that provide a way to allow incompatible types to interact.
- The *singleton pattern* ensures a class has only one instance and provides a global point of access to it.
- The *command pattern* is used to allow an object to store the information needed to execute some other functionality at a later time. For example it can help implement a redo-undo scenario.
- The *composite pattern* describes a group of objects that are to be treated in the same way as a single instance of an object. The intent of a composite is to *compose* objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly. The *visitor pattern* separates the algorithm implementation from the data structure. These two patterns can work together. The composite pattern forms a tree structure, and the visitor pattern applies a function to the tree structure and brings the result back.
- The *template pattern* is, as its name suggests, a program or algorithm skeleton.
- The *private data class pattern* is used to encapsulate fields and methods that can be used to manipulate the class instance.
- The *builder pattern* provides abstract steps of building objects. Using this pattern allows a developer to pass different implementations of abstract steps.
- The *façade pattern* allows you to create a higher level interface that can be used to make it easier to invoke underlying class libraries.
- The *memento pattern* saves an object's internal state for later use.

Working with the Chain of Responsibility Pattern

The chain of responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains a set of logic that describes the types of command objects it can handle and how to pass off those it cannot handle to the next processing object in the chain. The sample in Listing 3-2 shows a physical check process that needs to make sure that a person's age is between 18 and 65, that their weight is no more than 200 kilograms, and that they are taller than 120 centimeters.

The type in Listing 3-2 is called a *Record*. Listing 3-2 uses a *Record* to store a patient's medical data. It has several named fields that are used to hold the patient's data. It is very much like a database record. Listing 3-1 shows how to define a *Record* type and create a *record* object. The sample code creates a point record that has its *X* and *Y* fields set to *(1, 1)*.

LISTING 3-1 Defining a record type and creating a *Record* object

```
// define a point record
type Point2D = {
    X : float
    Y : float
}

// create original point record
let originalPoint = { X = 0.0; Y = 0.0 }

// create (1,1) point record
let onePoint = { X = 1.0; Y = 1.0 }
```

The record object implicitly forces data initialization; therefore, initial values are not optional when creating a *Record* type. The invoker must define the patient with some data, and this eliminates any possible initialization problems.

LISTING 3-2 Chain of responsibility pattern

```
// define a record to hold a person's age and weight
type Record = {
    Name : string;
    Age : int
    Weight: float
    Height: float
}

// Chain of responsibility pattern
let chainOfResponsibility() =
```

```
// function to check that the age is between 18 and 65
let validAge record =
    record.Age < 65 && record.Age > 18

// function to check that the weight is less than 200
let validWeight record =
    record.Weight < 200.

// function to check that the height is greater than 120
let validHeight record =
    record.Height > 120.

// function to perform the check according to parameter f
let check f (record, result) =
    if not result then record, false
    else record, f(record)

// create chain function
let chainOfResponsibility = check validAge >> check validWeight >> check validHeight

// define two patients' records
let john = { Name = "John"; Age = 80; Weight = 180.; Height = 180. }
let dan = { Name = "Dan"; Age = 20; Weight = 160.; Height = 190. }

printfn "John's result = %b" (chainOfResponsibility (john, true) |> snd)
printfn "Dan's result = %b" (chainOfResponsibility (dan, true) |> snd)
```

Execution result from the chain of responsibility sample

```
John's result = false
Dan's result = true
```

Note You have to execute the *chainOfResponsibility* function to get the result shown.

In the implementation in Listing 3-2, three functions (responsibilities) are composed into a chain and the data is passed along the chain when it is being processed. The parameter passed in contains a Boolean variable that decides whether the data can be processed. In Listing 3-2, all the functions are in effect AND-ed together. The parameter passed into the first function contains a Boolean value. The successive function can be invoked only if the Boolean value is *true*.

The other implementation is used for pipelining, as shown in Listing 3-3, rather than function composition. The *chainTemplate* higher-order function takes a process and *canContinue* function. The *canContinue* function always returns true, and the process function is a simple "increase one" function. The execution result is *2*.

LISTING 3-3 Chain of responsibility sample using pipelining

```
// chain template function
let chainTemplate processFunction canContinue s =
    if canContinue s then
        processFunction s
    else s

let canContinueF _ = true
let processF x = x + 1

//combine two functions to get a chainFunction
let chainFunction = chainTemplate processF canContinueF

// use pipeline to form a chain
let s = 1 |> chainFunction |> chainFunction

printfn "%A" s
```

The other chain of responsibility implementation uses the partial pattern feature in F#. I introduced the unit of measure to make the code readable. The process goes from the first case and stops when the condition is met. The sample code is listed in Listing 3-2. The sample code checks the height and weight value for some predefined criteria. The person's data is checked against *NotPassHeight* and then *NotPassWeight* if his height passes the validation criteria. The code also demonstrates how to use the F# unit-of-measure feature, which avoids possible confusion because of the unit of measure used. The parameter for *makeCheck* is *#Person*, which means that any object of type *Person* or derived from a *Person* type can be passed in.

Listing 3-4 uses units-of-measure language constructs within a calculation. Only the number with the same unit of measure can be involved in the same calculation. Listing 3-4 shows how to define a kilogram (*kg*) unit and decorate it with a number.

LISTING 3-4 Defining and using a *kg* unit of measure

```
// define unit-of-measure kg
[<Measure>] type kg

// define 1kg and 2kg variables
let oneKilo = 1<kg>
let twoKilo = 1<kg> + 1<kg>
```

The *None* and *Some(person)* syntax in the sample code in Listing 3-6 represents a *Nullable*-type-like data structure called an *option*. You can think of *None* as NULL. The special function `let (| NotPassHeight | _ |)` is called an *active pattern*. It takes a *person* parameter and decides whether the person meets certain criteria. If the person meets the criteria, the function returns *Some(person)* and triggers the match statement. Listing 3-5 shows how to use the *Some()/None* syntax to check for an odd number. This sample introduced several new concepts. I will come back to these concepts in detail in Chapter 5.

LISTING 3-5 Using active pattern, *option*, and *match* to check for an odd number

```
// define an active pattern function to check for an odd number
let (| Odd | _ |) x = if x % 2 = 0 then None else Some(x)

// define a function to check for an odd number
let findOdd x =
    match x with
    | Odd x -> printfn "x is odd number"
    | _ -> printfn "x is not odd number"

// check odd number
findOdd 3
findOdd 4
```

Execution result

```
x is odd number
x is not odd number
```

LISTING 3-6 Chain of responsibility pattern using partial pattern matching

```
// define two units of measure: cm and kg
[<Measure>] type cm
[<Measure>] type kg

// define a person class with its height and weight set to 0cm and 0kg
type Person() =
    member val Height = 0.<cm> with get, set
    member val Weight = 0.<kg> with get, set

// define a higher order function that takes a person record as a parameter
let makeCheck passingCriterion (person: #Person) =
    if passingCriterion person then None  //if passing, say nothing, just let it pass
    else Some(person)   //if not passing, return Some(person)

// define NotPassHeight when the height does not meet 170cm
let (| NotPassHeight | _ |) person = makeCheck (fun p -> p.Height > 170.<cm>) person

// define the NotPassWeight when weight does not fall into 100kg and 50kg range
let (| NotPassWeight | _ |) person =
    makeCheck (fun p -> p.Weight < 100.<kg> && p.Weight > 50.<kg>) person

// check incoming variable x
let check x =
    match x with
    | NotPassHeight x -> printfn "this person is not tall enough"
    | NotPassWeight x -> printfn "this person is out of weight range"
    | _ -> printfn "good, this person passes"
```

```
// create a person with 180cm and 75kg
let p = Person(Height = 180.<cm>, Weight = 75.<kg>)

// perform the chain check
check p
```

Execution result

```
good, this person passes
```

Working with the Adapter Pattern

The adapter pattern is a design pattern that translates one interface for a type into an interface that is compatible with some other type. An adapter allows classes to work together that normally could not because of incompatible types. In Listing 3-8, we use the *Generic Invoke(GI)* function as an adapter or bridge to invoke two methods of incompatible types. By using the GI function, a common interface is no longer needed and the function can still be invoked. The GI function is a static type *constraint* function, it requires that type *T* define a certain member function. For example, in Listing 3-7, it requires that the type *T* has a *canConnect* function that takes *void (unit)* and returns a Boolean. (Note that F# requires you to declare a function as "inline" when arguments of the function are *statically resolved type parameters* such as those in the following code listing.)

LISTING 3-7 GI function

```
// define a GI function
let inline canConnect (x : ^T) = (^T : (member CanConnect : unit->bool) x)
```

The interesting thing about the design pattern implementation in Listing 3-8 is that *Cat* and *Dog* do not have any common base class or interface. However, they can still be processed in a unified function. This implementation can be used to invoke the legacy code, which does not share any common interface or base class. (You should note, by the way, that this is a sloppy way of solving the problem and should be considered only when no other option is available.)

Imagine that you have two legacy systems that need to be integrated and that you do not have access to the source code. It would be difficult to integrate the systems in other languages, but it's possible and even easy in F# using the *generic invok*e technique.

LISTING 3-8 The adapter pattern (bridge pattern)

```
//define a cat class
type Cat() =
    member this.Walk() = printfn "cat walks"

// define a dog class
type Dog() =
    member this.Walk() = printfn "dog walks"

// adapter pattern
let adapterExample() =
    let cat = Cat()
    let dog = Dog()

    // define the GI function to invoke the Walk function
    let inline walk (x : ^T) = (^T : (member Walk : unit->unit) x)

    // invoke GI and both Cat and Dog
    walk(cat)
    walk(dog)
```

Execution result from adapter pattern sample

```
cat walks
dog walks
```

Note The implementation in Listing 3-8 can also be viewed as a bridge pattern.

Working with the Command Pattern

The command pattern is a design pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time. Listing 3-10 shows how to use the command pattern to implement a redo-undo framework. This is an example of typical usage of the command pattern in the OOP world.

Listing 3-9 defines a result using the *ref* keyword. The *ref* keyword defines a reference type that points to the value *7*. The result is a reference cell. You can think of the *ref* keyword as a way to define a mutable variable.

LISTING 3-9 Reference cell

```
// define a reference cell to value 0
let a = ref 0

// define a function to increase a's value by 1
let increaseA() =
    a := !a + 1

// increase a's value and print out result
increaseA()
printfn "a = %A" !a
```

Execution result

```
a = 1
```

Note F# provides *incr* and *decr* to increase or decrease reference cell values by 1. When using the *incr* function, the *increaseA* function becomes `let increaseA() = incr a`.

Note The := operator is used to assign a new value to the content of the reference cell. The *!* (pronounced *bang*) operator is used to retrieve the reference cell content.

LISTING 3-10 Command pattern

```
// define a command record
type Command = { Redo: unit->unit; Undo: unit->unit }

let commandPatternSample() =

    // define a mutable storage
    let result = ref 7

    // define the add command
    let add n = {
        Redo = (fun _ -> result := !result + n)
        Undo = (fun _ -> result := !result - n) }

    // define the minus command
    let minus n = {
        Redo = (fun _ -> result := !result - n)
        Undo = (fun _ -> result := !result + n) }

    // define an add 3 command
    let cmd = add 3
    printfn "current state = %d" !result
```

```
    // perform add 3 redo operation
    cmd.Redo()
    printfn "after redo: %d" !result

    // perform an undo operation
    cmd.Undo()
    printfn "after undo: %d" !result
```

Execution result from the command pattern sample obtained by invoking the *commandPatternSample* function

```
current state = 7
after redo: 10
after undo: 7
```

Note There is no storage structure for command history; however, adding such a storage structure is trivial.

Note According to the MSDN documentation (*http://msdn.microsoft.com/en-us/library/dd233186.aspx*), a mutable variable should be used instead of a reference cell whenever possible. The preceding code uses a reference cell just for demo purposes. You can convert this code to use a mutable variable.

There is another implementation that emphasizes that the command can be treated like data. The code defines two types of commands: *deposit* and *withdraw*. The *Do* and *Undo* functions are used to perform the do and undo actions. See Listing 3-12.

To implement this *Do* and *Undo* functionality, it is helpful to understand the F# discriminated union (DU) feature. Listing 3-11 demonstrates how to use a DU to check whether or not the given time is a working hour. Note how the first DU, *DayOfAWeek*, looks a lot like an enum, but without the default numeric value. In the second example, *TWorkingHour*, the DU case *Hour* has a tuple value, where the first element of the tuple is a *DayOfAWeek* and the second element is an integer.

LISTING 3-11 Using DU to check whether the given time is a working hour

```
// define day of the week
type DayOfAWeek =
    | Sunday
    | Monday
    | Tuesday
    | Wednesday
    | Thursday
    | Friday
    | Saturday
```

```
// define working hour
type TWorkingHour =
    | Hour of DayOfAWeek * int

// check that the working hour is Monday to Friday 9:00 to 17:00
let isWorkingHour day =
    match day with
    | Hour(Sunday, _) -> false
    | Hour(Saturday, _) -> false
    | Hour(_, time) -> time >= 9 && time <= 17

// check if Sunday is working hour
let sunday = Hour(Sunday, 9)
printfn "%A is working hour? %A" sunday (isWorkingHour sunday)

// check if Monday 10:00 is working hour
let monday = Hour(Monday, 10)
printfn "%A is working hour? %A" monday (isWorkingHour monday)
```

Execution result

```
Hour (Sunday,9) is working hour? false
Hour (Monday,10) is working hour? true
```

Now that you understand discriminated unions, you can apply them to the command pattern.

LISTING 3-12 Command pattern implementation II

```
// define two command types
type CommandType =
    | Deposit
    | Withdraw

// define the command format, which has a command type and an integer
type TCommand =
    | Command of CommandType * int

// mutable variable result
let result = ref 7

// define a deposit function
let deposit x = result := !result + x

// define a withdraw function
let withdraw x = result := !result - x

// do function to perform a do action based on command type
let Do = fun cmd ->
    match cmd with
    | Command(CommandType.Deposit, n) -> deposit n
    | Command(CommandType.Withdraw,n) -> withdraw n
```

```
// undo function to perform an undo action based on command type
let Undo = fun cmd ->
    match cmd with
    | Command(CommandType.Deposit, n) -> withdraw n
    | Command(CommandType.Withdraw,n) -> deposit n

// print the current balance
printfn "current balance %d" !result

// deposit 3 into the account and print the balance
let depositCmd = Command(Deposit, 3)
Do depositCmd
printfn "after deposit: %d" !result

// undo the deposit command and print the balance
Undo depositCmd
printfn "after undo: %d" !result
```

Execution result

```
current balance 7
after deposit: 10
after undo: 7
```

Working with the Observer Pattern

The observer pattern is a pattern in which a *subject* object maintains a list of its observer dependents. The *subject* automatically notifies its dependents of any changes by calling one of the dependent's methods. The implementation in Listing 3-13 passes the function into the *subject*, and the *subject* notifies its changes by calling this function along with some parameters.

LISTING 3-13 Observer pattern

```
// define a subject
type Subject() =
    // define a default notify function
    let mutable notify = fun _ -> ()

    // subscribe to a notification function
    member this.Subscribe notifyFunction =
        let wrap f i = f i; i
        notify <- wrap notifyFunction >> notify

    // reset notification function
    member this.Reset() = notify <- fun _ -> ()

    // notify when something happens
    member this.SomethingHappen k =
        notify k
```

```
// define observer A
type ObserverA() =
    member this.NotifyMe i = printfn "notified A %A" i

// define observer B
type ObserverB() =
    member this.NotifyMeB i = printfn "notified B %A" i

// observer pattern
let observer() =
    // create two observers
    let a = ObserverA()
    let b = ObserverB()

    // create a subject
    let subject = Subject()

    // let observer subscribe to subject
    subject.Subscribe a.NotifyMe
    subject.Subscribe b.NotifyMeB

    // something happens to the subject
    subject.SomethingHappen "good"
```

Execution result from the observer pattern sample obtained by invoking the *observer* function

```
notified B "good"
notified A "good"
```

F#'s *Observable* module can be used to implement this pattern as well. In Listing 3-14, an event is defined along with three observers of the event. Compared to the version in Listing 3-13, this version is much more lightweight. The *myEvent* value is bound to an instance of the F# *event* type. For the *Observable* module to subscribe to the event, you have to publish the event. After the event is published, the *Observable.add* function is used to add the event-handler function to this event. When the event is fired by using *Trigger*, all the event-handler functions will be notified.

LISTING 3-14 Using the *Observable* module to implement the observer pattern

```
// define an event
let myEvent = Event<_>()

// define three observers
let observerA = fun i -> printfn "observer A noticed something, its value is %A" i
let observerB = fun i -> printfn "observer B noticed something, its value is %A" i
let observerC = fun i -> printfn "observer C noticed something, its value is %A" i

// publish the event and add observerA
myEvent.Publish
|> Observable.add observerA
```

```
// publish the event and add observerA
myEvent.Publish
|> Observable.add observerB

// publish the event and add observerA
myEvent.Publish
|> Observable.add observerC

//fire event with value 1
myEvent.Trigger 1
```

Execution result

```
observer A noticed something, its value is 1
observer B noticed something, its value is 1
observer C noticed something, its value is 1
```

Working with the Decorator Pattern

The decorator pattern can be used to extend (a.k.a. *decorate*) the functionality of an object at run-time. In Listing 3-15, the decorator pattern is used along with the composite operator to add new logic to the existing function. As the function is passed dynamically into a structure, the run-time behavior can be easily changed. The sample code defines a property that exposes a function. This function can then be changed at runtime.

LISTING 3-15 Decorator pattern

```
// define the Divide class
type Divide() =
    // define basic divide function
    let mutable divide = fun (a,b) -> a / b

    // define a property to expose the function
    member this.Function
        with get() = divide
        and set(v) = divide <- v

    // method to invoke the function
    member this.Invoke(a,b) = divide (a,b)

// decorator pattern
let decorate() =

    // create a divide instance
    let d = Divide()

    // set the check zero function
    let checkZero (a,b) = if b = 0 then failwith "a/b and b is 0" else (a,b)
```

```
    // invoke the function without check zero
    try
        d.Invoke(1, 0) |> ignore
    with e -> printfn "without check, the error is = %s" e.Message

    // add the check zero function and then invoke the divide instance
    d.Function <- checkZero >> d.Function
    try
        d.Invoke(1, 0) |> ignore
    with e -> printfn "after add check, error is = %s" e.Message
```

Execution result from the decorator pattern sample obtained by invoking the *decorate* function

```
without check, the error is = Attempted to divide by zero.
after add check, error is = a/b and b is 0
```

Working with the Proxy Pattern

The proxy pattern uses a class that acts as a placeholder or interface for another object or function. It's often used for caching, to control access, or to delay the execution or creation of an object that is costly in the form of time or resources. See Listing 3-16. The *CoreComputation* class hosts two calculation functions, named *Add* and *Sub*. The class also exposes a proxy class from which a user can get access to the computation.

LISTING 3-16 Proxy pattern

```
// define core computation
type CoreComputation() =
    member this.Add(x) = x + 1
    member this.Sub(x) = x - 1
    member this.GetProxy name =
        match name with
        | "Add" -> this.Add, "add"
        | "Sub" -> this.Sub, "sub"
        | _ -> failwith "not supported"

// proxy implementation
let proxy() =
    let core = CoreComputation()

    // get the proxy for the add function
    let proxy = core.GetProxy "Add"

    // get the compute from proxy
    let coreFunction = fst proxy

    // get the core function name
    let coreFunctionName = snd proxy
```

```
    // perform the core function calculation
    printfn "performed calculation %s and get result = %A" coreFunctionName (coreFunction 1)
```

Execution result from the proxy pattern sample obtained by invoking the *proxy* function

```
performed calculation add and get result = 2
```

Working with the Strategy Pattern

The strategy pattern is a software design pattern whereby algorithms can be selected and used at runtime. Listing 3-17 uses a function to hold different strategies. During runtime, the strategy can be modified.

LISTING 3-17 Strategy pattern

```
// quick sort algorithm
let quicksort l =
    printfn "quick sort"

// shell short algorithm
let shellsort l =
    printfn "shell short"

// bubble short algorithm
let bubblesort l =
    printfn "bubble sort"

// define the strategy class
type Strategy() =
    let mutable sortFunction = fun _ -> ()
    member this.SetStrategy f = sortFunction <- f
    member this.Execute n = sortFunction n

let strategy() =
    let s = Strategy()

    // set strategy to be quick sort
    s.SetStrategy quicksort
    s.Execute [1..6]

    // set strategy to be bubble sort
    s.SetStrategy bubblesort
    s.Execute [1..6]
```

Execution result from the strategy pattern sample obtained by invoking the *strategy* function

```
quick sort
bubble sort
```

Note The sample code does not really implement three sorting algorithms. Instead, the code simply outputs the name of the algorithm that would be used.

Listing 3-17 shows how to implement this pattern using the OOP paradigm. However, the strategy pattern can be implemented more succinctly with a functional approach. Listing 3-18 shows how to use the higher-order function named *executeStrategy* to implement this pattern using a functional paradigm.

LISTING 3-18 Strategy pattern using a higher-order function

```
// quick sort algorithm
let quicksort l =
    printfn "quick sort"

// shell short algorithm
let shellsort l =
    printfn "shell short"

// bubble short algorithm
let bubblesort l =
    printfn "bubble sort"

let executeStrategy f n = f n

let strategy() =
    // set strategy to be quick sort
    let s = executeStrategy quicksort
    // execute the strategy against a list of integers
    [1..6] |> s

    // set strategy to be bubble sort
    let s2 = executeStrategy bubblesort
    // execute the strategy against a list of integers
    [1..6] |> s2
```

Working with the State Pattern

The state pattern is used to represent the ability to vary the behavior of a routine depending on the state of an object. This is a clean way for an object to partially change its type at runtime. Listing 3-19 shows that the interest rate is decided by the internal state: account balance. The higher the balance is, the higher the interest is that a customer will receive. In the sample, I also demonstrate how to use the unit-of-measure feature.

LISTING 3-19 State pattern

```
// define account state
type AccountState =
    | Overdrawn
    | Silver
    | Gold

// define unit of measure as US dollar
[<Measure>] type USD

// define an account that takes the unit of measure
type Account<[<Measure>] 'u>() =
    // field to hold the account balance
    let mutable balance = 0.0<_>

    // property for account state
    member this.State
        with get() =
            match balance with
            | _ when balance <= 0.0<_> -> Overdrawn
            | _ when balance > 0.0<_> && balance < 10000.0<_> -> Silver
            | _ -> Gold

    // method to pay the interest
    member this.PayInterest() =
        let interest =
            match this.State with
                | Overdrawn -> 0.
                | Silver -> 0.01
                | Gold -> 0.02
        interest * balance

    // deposit into the account
    member this.Deposit x =
        let a = x
        balance <- balance + a

    // withdraw from account
    member this.Withdraw x =
        balance <- balance - x

// implement the state pattern
let state() =
    let account = Account()

    // deposit 10000 USD
    account.Deposit 10000.<USD>

    // pay interest according to current balance
    printfn "account state = %A, interest = %A" account.State (account.PayInterest())

    // deposit another 2000 USD
    account.Withdraw 2000.<USD>
```

```
    // pay interest according to current balance
    printfn "account state = %A, interest = %A" account.State (account.PayInterest())
```

Execution result from the state pattern sample obtained by invoking the *state* function

```
account state = Gold, interest = 200.0
account state = Silver, interest = 80.0
```

In F#, one way to implement a state machine is with a *MailboxProcessor*. The F# *MailboxProcessor* can be viewed as a message queue. It takes an asynchronous workflow as the processing logic. The asynchronous workflow will be introduced in the next chapter, and it can be thought of as a simple function being executed on a background thread. The *Post* method is used to insert a message into the queue, and the *Receive* method is used to get the message out of the queue. In Listing 3-20, the variable *inbox* represents the message queue. When the state machine starts, it goes to *state0*, which is represented by the *state0()* function, and waits for user input. The state machine will transition to another state according to the user's input.

LISTING 3-20 State pattern with F# *MailBoxProcessor*

```
open Microsoft.FSharp.Control

type States =
    | State1
    | State2
    | State3

type StateMachine() =
    let stateMachine = new MailboxProcessor<States>(fun inbox ->
                let rec state1 () = async {
                    printfn "current state is State1"
                    // <your operations>

                    //get another message and perform state transition
                    let! msg = inbox.Receive()
                    match msg with
                        | State1 -> return! (state1())
                        | State2 -> return! (state2())
                        | State3 -> return! (state3())
                    }
                and state2() = async {
                    printfn "current state is state2"
                    // <your operations>

                    //get another message and perform state transition
                    let! msg = inbox.Receive()
                    match msg with
                        | State1 -> return! (state1())
                        | State2 -> return! (state2())
                        | State3 -> return! (state3())
                    }
```

```
                and state3() = async {
                    printfn "current state is state3"
                    // <your operations>

                    //get another message and perform state transition
                    let! msg = inbox.Receive()
                    match msg with
                        | State1 -> return! (state1())
                        | State2 -> return! (state2())
                        | State3 -> return! (state3())
                    }
                and state0 () =
                    async {

                        //get initial message and perform state transition
                        let! msg = inbox.Receive()
                        match msg with
                            | State1 -> return! (state1())
                            | State2 -> return! (state2())
                            | State3 -> return! (state3())
                    }
                state0 ())

    //start the state machine and set it to state0
    do
        stateMachine.Start()

    member this.ChangeState(state) = stateMachine.Post(state)

let stateMachine = StateMachine()
stateMachine.ChangeState(States.State2)
stateMachine.ChangeState(States.State1)
```

Execution result in FSI

```
current state is state2
current state is State1
```

Note If the preceding code is executed in Microsoft Visual Studio debug mode, *Thread.Sleep* is needed because the main process (thread) needs to give CPU cycles to the background execution.

Working with the Factory Pattern

The factory pattern in Listing 3-21 is an object-oriented design pattern used to implement the concept of *factories*. It uses the *function* keyword as shortcut to the *match* statement. It can create an object without specifying the exact class of object that will be created. Listing 3-22 shows an example that uses the *object* expression to implement the factory pattern.

LISTING 3-21 Using the *function* keyword

```
// define two types
type Type =
  | TypeA
  | TypeB

// check with function keyword
let checkWithFunction = function
    | TypeA -> printfn "type A"
    | TypeB -> printfn "type B"

// check with match keyword
let checkWithMatch x =
    match x with
    | TypeA -> printfn "type A"
    | TypeB -> printfn "type B"
```

In Listing 3-22, the *factory* inside *factoryPattern* is actually a function. It is a shortcut for a *match* statement. The *checkWithFunction* and *checkWithMatch* functions in Listing 3-21 are equivalent.

LISTING 3-22 Example of the factory pattern

```
// define the interface
type IA =
  abstract Action : unit -> unit

// define two types
type Type =
  | TypeA
  | TypeB

let factoryPattern() =
    // factory pattern to create the object according to the input object type
    let factory = function
      | TypeA -> { new IA with
                     member this.Action() = printfn "I am type A" }
      | TypeB -> { new IA with
                     member this.Action() = printfn "I am type B" }

    // create type A object
    let obj1 = factory TypeA
    obj1.Action()

    // create type B object
    let obj2 = factory TypeB
    obj2.Action()
```

Execution result from the factory pattern sample obtained by invoking the *factoryPattern* function

```
I am type A
I am type B
```

The *factory* function returns an object that is not familiar. Actually, the return type is something called an *object expression*, and this lightweight syntax can simplify your code significantly. If the object is not involved in inheritance, you can pretty much use an object expression to replace a class definition completely. Listing 3-23 shows how to create an instance of interface *IA* using object expression syntax.

LISTING 3-23 Using object expression

```
// define the interface
type IA =
  abstract Action : unit -> unit

let a = { new IA with
            member this.Action() =
                printfn "this is from object expression" }
```

Working with the Singleton Pattern

The singleton pattern is a design pattern used to implement the mathematical concept of a *singleton*. It restricts the instantiation of a class to a single instance. This is useful when exactly one object is needed to coordinate actions across the system. One example of a singleton in F# is a *value*. An F# value is immutable by default, and this guarantees there is only one instance. Listing 3-24 shows how to make sure that an F# class instance is a singleton. The sample declares a private constructor and ensures that the class has only one instance in memory.

LISTING 3-24 An example of the singleton pattern

```
// define a singleton pattern class
type A private () =
    static let instance = A()
    static member Instance = instance
    member this.Action() = printfn "action from type A"

// singleton pattern
let singletonPattern() =
    let a = A.Instance
    a.Action()
```

Working with the Composite Pattern

The composite pattern is a partitioning design pattern. The composite pattern describes a group of objects that are to be treated in the same way as a single instance of that object. The typical application is a tree structure representation. Listing 3-25 demonstrates a tree structure. The sample focuses more on how to access this tree structure and bring back the result.

The dynamically generated wrapper object can be treated like a visitor to the tree. The visitor accesses the node and brings the result back to the invoker. In the sample code, the *CompositeNode* structure not only defines the tree but also defines three common ways to traverse the tree. It does the heavy lifting by encapsulating the tree traversal algorithm. The visitor defines how to process the single node and is responsible for bringing the result back to the invoker. In this sample, the visitor adds the value in the tree nodes and brings back the sum.

LISTING 3-25 An example of the composite pattern

```
// define visitor interface
type IVisitor<'T> =
    abstract member Do : 'T -> unit

// define a composite node
type CompositeNode<'T> =
    | Node of 'T
    | Tree of 'T * CompositeNode<'T> * CompositeNode<'T>
    with
        // define in-order traverse
        member this.InOrder f =
            match this with
            | Tree(n, left, right) ->
                left.InOrder f
                f n
                right.InOrder(f)
            | Node(n) -> f n

        // define pre-order traverse
        member this.PreOrder f =
            match this with
            | Tree(n, left, right) ->
                f n
                left.PreOrder f
                right.PreOrder f
            | Node(n) -> f n

        // define post order traverse
        member this.PostOrder f =
            match this with
            | Tree(n, left, right) ->
                left.PostOrder f
                right.PostOrder f
                f n
            | Node(n) -> f n

let invoke() =
    // define a tree structure
    let tree = Tree(1, Tree(11, Node(12), Node(13)), Node(2))
```

```
    // define a visitor, it gets the summary of the node values
    let wrapper =
        let result = ref 0
        ({ new IVisitor<int> with
                member this.Do n =
                    result := !result + n
        }, result)

    // pre-order iterates the tree and prints out the result
    tree.PreOrder (fst wrapper).Do
    printfn "result = %d" !(snd wrapper)
```

Execution result from the composite pattern sample obtained by calling the *invoke* function

```
result = 39
```

Working with the Template Pattern

The template pattern is, as its name suggests, a program or algorithm skeleton. It is a behavior-based pattern. In F#, we have higher-order functions that can serve as a template to generate other functions. It is natural to use higher-order functions to implement this pattern. Listing 3-26 defines a three-stage database operation function named *TemplateF*. The actual implementation is provided outside of this skeleton function. I do not assume the database connection and query are all the same, so three functions are left outside of the class definition, and the user can define and pass in their own version of each.

LISTING 3-26 An example of the template pattern

```
// the template pattern takes three functions and forms a skeleton function named
TemplateF
type Template(connF, queryF, disconnF) =
    member this.Execute(conStr, queryStr) =
        this.TemplateF conStr queryStr
    member this.TemplateF =
            let f conStr queryStr =
                connF conStr
                queryF queryStr
                disconnF ()
            f

// connect to the database
let connect conStr =
    printfn "connect to database: %s" conStr

// query the database with the SQL query string
let query queryStr =
    printfn "query database %s" queryStr
```

```
// disconnect from the database
let disconnect ()  =
    printfn "disconnect"

let template() =
    let s = Template(connect, query, disconnect)
    s.Execute("<connection string>", "select * from tableA")

template()
```

Execution result from the template pattern sample obtained by invoking the *template* function

```
connect to database: <connection string>
query database select * from tableA
disconnect
```

Note The *connect*, *query*, and *disconnect* functions can be implemented as private functions in a class.

The class definition is convenient for C# projects that need to reference the implementation of this design pattern in an F# project. However, the class is not necessary in an F#-only solution. Listing 3-27 shows how to use higher-order functions to implement the template pattern.

LISTING 3-27 Template pattern with a higher-order function

```
// connection, query, and disconnect functions
let connect(conStr ) = printfn "connect using %s" conStr
let query(queryStr) = printfn "query with %s" queryStr
let disconnect() = printfn "disconnect"

// template pattern
let template(connect, query, disconnect) (conStr:string) (queryStr:string)=
    connect(conStr)
    query(queryStr)
    disconnect()

// concrete query
let queryFunction = template(connect, query, disconnect)

// execute the query
do queryFunction "<connection string>" "select * from tableA"
```

Working with the Private Data Class Pattern

The private data class pattern is a design pattern that encapsulates class properties and associated data manipulation. The purpose of the private accessibility is to prevent the modification of these values. C# uses the *readonly* property, which does not have a *setter* function, to solve this problem. F# values are immutable by default, so implementing this *readonly* type of behavior is supported inherently. In the following example, I use an F# record type to implement the pattern by extending the record type. The *with* keyword in the code shown in Listing 3-28 is a way to tell the compiler that some property, method, or both will be added to the *record* type. In the sample code, the circle data remains the same once it is created. Some object-oriented implementations even implement another class so that there is little chance to modify the values. The immutability of *record* types eliminates the needs of a second class, as well as the need for explicitly defining *getter*-only properties with a keyword.

LISTING 3-28 An example of the private data class pattern

```
type Circle = {
    Radius : float;
    X : float;
    Y : float }



with
    member this.Area = this.Radius**2. * System.Math.PI
    member this.Diameter = this.Radius * 2.
let myCircle = {Radius = 10.0; X = 5.0; Y = 4.5}
printfn "Area: %f Diameter: %f" myCircle.Area myCircle.Diameter
```

Working with the Builder Pattern

The builder pattern provides abstract steps of building objects. This allows you to pass different implementations of specific abstract steps. Listing 3-29 demonstrates the abstract steps of making a pizza. The invoker can pass in different implementation steps to the *cook* function to generate different pizzas.

LISTING 3-29 An example of the builder pattern sample

```
// pizza interface
type IPizza =
    abstract Name : string with get
    abstract MakeDough : unit->unit
    abstract MakeSauce : unit->unit
    abstract MakeTopping: unit->unit
```

```
// pizza module that defines all recipes
[<AutoOpen>]
module PizzaModule =
    let makeNormalDough() = printfn "make normal dough"
    let makePanBakedDough() = printfn "make pan baked dough"
    let makeCrossDough() = printfn "make cross dough"

    let makeHotSauce() = printfn "make hot sauce"
    let makeMildSauce() = printfn "make mild sauce"
    let makeLightSauce() = printfn "make light sauce"

    let makePepperoniTopping() = printfn "make pepperoni topping"
    let makeFiveCheeseTopping() = printfn "make five cheese topping"
    let makeBaconHamTopping() = printfn "make bacon ham topping"

// define a pepperoni pizza recipe
let pepperoniPizza =
        {   new IPizza with
                member this.Name = "Pepperoni Pizza"
                member this.MakeDough() = makeNormalDough()
                member this.MakeSauce() = makeHotSauce()
                member this.MakeTopping() = makePepperoniTopping() }

// cook takes pizza recipe and makes the pizza
let cook(pizza:IPizza) =
    printfn "making pizza %s" pizza.Name
    pizza.MakeDough()
    pizza.MakeSauce()
    pizza.MakeTopping()

// cook pepperoni pizza
cook pepperoniPizza
```

Execution result from the builder pattern sample

```
making pizza Pepperoni Pizza
make normal dough
make hot sauce
make pepperoni topping
```

The pizza interface and object expression give the program a good structure, but it makes things unnecessarily complicated. The builder pattern requires the actual processing function or functions be passed in, which is a perfect use of higher-order functions. Listing 3-30 uses a higher-order function to eliminate the interface and object expression.

LISTING 3-30 Builder pattern implementation using a higher-order function

```
// pizza module that defines all recipes
[<AutoOpen>]
module PizzaModule =
    let makeNormalDough () = printfn "make normal dough"
    let makePanBakedDough () = printfn "make pan baked dough"
    let makeCrossDough() = printfn "make cross dough"

    let makeHotSauce() = printfn "make hot sauce"
    let makeMildSauce() = printfn "make mild sauce"
    let makeLightSauce() = printfn "make light sauce"

    let makePepperoniTopping() = printfn "make pepperoni topping"
    let makeFiveCheeseTopping() = printfn "make five cheese topping"
    let makeBaconHamTopping() = printfn "make bacon ham topping"

// cook takes the recipe and ingredients and makes the pizza

let cook pizza recipeSteps =
    printfn "making pizza %s" pizza
    recipeSteps
    |> List.iter(fun f -> f())

[ makeNormalDough; makeMildSauce
  makePepperoniTopping ]
|> cook "pepperoni pizza"
```

Working with the Façade Pattern

The façade pattern provides a higher-level interface that makes invoking an underlying class library easier, more readable, or both. Listing 3-31 shows how to perform an employment background check.

LISTING 3-31 An example of the façade pattern

```
// define Applicant record
type Applicant = { Name : string }

// library to perform various checks
[<AutoOpen>]
module SubOperationModule =
    let checkCriminalRecord (applicant) =
        printfn "checking %s criminal record..." applicant.Name
        true

    let checkPastEmployment (applicant) =
        printfn "checking %s past employment..." applicant.Name
        true

    let securityClearance (applicant, securityLevel) =
        printfn "security clearance for %s ..." applicant.Name
        true
```

```
// façade function to perform the background check
let isBackgroundCheckPassed(applicant, securityLevel) =
    checkCriminalRecord applicant
    && checkPastEmployment applicant
    && securityClearance(applicant, securityLevel)

// create an applicant
let jenny = { Name = "Jenny" }

// print out background check result
if isBackgroundCheckPassed(jenny, 2) then printfn "%s passed background check" jenny.Name
else printfn "%s failed background check" jenny.Name
```

Execution result from the façade pattern sample

```
checking Jenny criminal record...
checking Jenny past employment...
security clearance for Jenny ...
Jenny passed background check
```

Working with the Memento Pattern

The memento pattern saves an object's internal state so that it can be used later. In Listing 3-32, the *particle* class saves its location information and later restores that information back to the saved location. If the state data is relatively small, a list storage can easily turn the memento pattern into a redo-undo framework.

LISTING 3-32 An example of the memento pattern

```
// define location record
type Location = { X : float; Y : float }

// define a particle class with a location property
type Particle() =
    let mutable loc = {X = 0.; Y = 0.}
    member this.Loc
        with get() = loc
        and private set v = loc <- v
    member this.GetMemento() = this.Loc
    member this.Restore v = this.Loc <- v
    member this.MoveXY(newX, newY) = loc <- { X = newX; Y = newY }


// create a particle
let particle = Particle()
```

```
// save current state
let currentState = particle.GetMemento()
printfn "current location is %A" particle.Loc

// move particle to new location
particle.MoveXY(2., 3.)
printfn "current location is %A" particle.Loc

// restore particle to previous saved location
particle.Restore currentState
printfn "current location is %A" particle.Loc
```

Writing Design Patterns: Additional Notes

As I mentioned in the beginning of this chapter, design patterns have been criticized since their birth. Many functional programmers believe that design patterns are not needed when programming in a functional style. Peter Norvig, in his paper "Design Patterns in Dynamic Languages," claims that design patterns are just missing language features and demonstrates that design patterns can be simplified or eliminated completely when using a different language. I am not planning to be part of these discussions. Design patterns are a way to represent a system or idea. It is really a de facto and concise way for many computer professionals to describe system design. If the program is simple and small, design patterns are often unnecessary. For these scenarios, the use of basic data and flow-control structure is enough. However, when a program becomes large and complicated, a tested approach is needed to organize thinking and avoid possible design flaws or bugs. If the basic data structure is analogous to a word in a sentence, design patterns can be viewed as the idea to organize an article.

As a functional-first programming language, F# is adept at creating code with a functional style. For example, the *pipeline* and *function* composition operators make function operation much easier. Instead of being confined to a class, the function can be freely passed and processed like data in F#. If the design pattern is mainly about how to pass an action/operation or coordinate the flow of an operation, the pipeline and function composition operators can definitely simplify the implementation. The chain of responsibility pattern is an example. The biggest change from C# is that a function in F# is no longer auxiliary to the data; instead, it can be encapsulated, stored, and manipulated in a class. The data (field and property) in a class can actually be provided as a function or as method parameters and remain auxiliary to the function. Additionally, the presence of a class is optional if the class only serves as an operation container. The builder pattern demonstrates a way to eliminate the class while still implementing the same functionality.

Functional programming can still have a structure to encapsulate logic into a unit. Functions, which can be treated like data, can be encapsulated in a class or inside a closure and, more importantly, the application of object expressions provides an even simpler way to organize the code. Listing 3-33 shows different ways to encapsulate the data.

LISTING 3-33 Data encapsulation

F# closure

```
let myFunction () =
    let constValue = 100
    let f () = [1..constValue] |> Seq.iter (printfn "%d")
    f()
```

Object expression

```
let obj =
    let ctor = printfn "constructor code"
    let field = ref 8
    { new IA with
        member this.F() = printfn "%A" (!field)
      interface System.IDisposable with
        member this.Dispose() = ()}
```

Object expressions are great, because the type is created on the fly by the compiler. Instead of inventing a permanent boilerplate class to hold the function and data, you can use object expressions to quickly organize functions and data into a unit and get the job done. Imagine an investment bank with a bunch of mathematicians who lack a computer background: object expressions can let them quickly transform their knowledge into code without worrying about programmers complaining about their inability to implement complex inheritance hierarchies. The flattened structure from the object expression is a straightforward and suitable approach for quick prototyping and agile development. The command pattern is a good sample for demonstrating how to use object expressions to simplify the design.

Both functional programming and object-oriented programming have their own way of reusing the code. Object-oriented programming uses inheritance, while functional programming uses higher-order functions. Both approaches have loyal followers, and you might already be convinced that one is superior to the other. I say that both approaches have their own advantages under certain circumstances. Unfortunately, neither is a silver bullet that can be used to solve all problems. Using the right tool for the right job is the key. F#, which supports both OO and functional programming, provides both approaches, and this gives the developer the liberty to use the best way to perform the system design.

F# provides the alternative to encapsulation (object expressions) and inheritance (higher order functions): polymorphism. It can also be implemented by higher-order functions when given different parameters. This is yet another example of how F# provides a wide set of tools for developers to implement their components and systems.

In addition, the adapter pattern introduces the GI function, which breaks class encapsulation and makes possible communication between objects that do not share a common base class. It is not a recommended way to use the original object-oriented design; however, it is a feasible approach to wrap legacy code because of inaccessibility to the source code. It is not fair to blame a gun for causing crime and not blame the criminal. Likewise, F# provides this approach, but I'll leave the decision to you regarding when and how to use it.

It is totally fine to copy a standard object-oriented approach when doing system design, especially when someone is new to a language. If you are motivated to use F# to write design patterns, here are some principles that I used to implement the design patterns in this chapter. If the design pattern is a behavior design pattern, its main focus is on how to organize the function, so consider using the function composition and pipeline operators. If the function needs to be organized into a unit, put the function into a module and use object expressions to organize the function. If the design pattern is a structural design pattern, I always question why extra structure is needed. If the extra structure is a placeholder for functionality, higher-order functions most likely will do the same job. If the extra structure is needed to make two unrelated objects work together, the GI function could be a good candidate to simplify the design.

F# is a young language and how to properly apply its language feature into the system design is still a new topic. Keep in mind that F# provides the OOP way of implementing class encapsulation, inheritance, and polymorphism. This chapter is only a small step to explore how to use F# in system design.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

TAO LIU is a VP leading a team of .NET and Java developers in the Credit Technology group at Citi. He was a Software Design Engineer in Test (SDET) on the Microsoft F# team. A leader in the F# user community and organizer of the Seattle, Washington F# user group, Liu gives video talks on F# design patterns for Microsoft Channel 9 and he's the main contributor to the F# 3.0 sample package on Codeplex.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press