F#

Succinctly

by Robert Pickering

Syncfusion | Technology Resource Portal

F# Succinctly

By
Robert Pickering

Foreword by Daniel Jebaraj

Partially based on *Beginning F#* by Robert Pickering, Apress 2009.

Important licensing information. Please read.

Edited by

This publication was edited by Jay Natarajan, senior product manager, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Robert Pickering was born in Sheffield, in the north of England, but a fascination with computers and the Madchester indie music scene led him to cross the Pennines and study computer science at the University of Manchester.

After finishing his degree, he moved to London to catch the tail end of the dot-com boom. From there he worked on projects in Denmark, Holland, Belgium, and Switzerland, finally settling in Paris, France, where he lives now with his wife and their four cats.

He enjoys tinkering with all things technical, especially F# and other functional programming related things. This has led to blogging and writing about F#, as well as contributing to F# open-source projects and organizing the occasional conference.

Using Code Examples

This book relies heavily on code examples to express F# concepts. The code samples are available at https://bitbucket.org/syncfusion/fsharp-succinctly.

Code samples are provided as individual Visual Studio F# project files. The samples are organized by chapter and named after the sub-headings of their respective chapters.

Most of the samples are console applications. From Visual Studio, if you run your application in debug mode (F5), the console window pops up and closes immediately. To view the sample result, use **Start** without debugging (Ctrl+F5). This will add a *Press any key to continue* prompt at the end of a console application, allowing you to close the console window by pressing any key.

Chapter 1 Introduction

This introductory chapter will address some of the major questions you may have about F# and functional programming (FP).

What Is Functional Programming?

Pure functional programming views all programs as collections of functions that accept arguments and return values. Unlike imperative and object-oriented programming, it allows no side effects and uses recursion instead of loops for iteration. The functions in a functional program are very much like mathematical functions because they do not change the state of the program. In the simplest terms, once a value is assigned to an identifier it never changes, functions do not alter parameter values, and the results that functions return are completely new values. In typical underlying implementations, once a value is assigned to an area in memory, it does not change. To create results, functions copy values and then change the copies, leaving the original values free to be used by other functions and eventually be thrown away when no longer needed. (This is where the idea of garbage collection originated.)

The mathematical basis for pure functional programming is elegant, and FP therefore provides beautiful, succinct solutions for many computing problems, but its stateless and recursive nature makes the other paradigms convenient for handling many common programming tasks. However, one of F#'s great strengths is that you can use multiple paradigms and mix them to solve problems in the way you find most convenient.

Why Is Functional Programming Important?

When people think of functional programming, they often view its statelessness as a fatal flaw without considering its advantages. One could argue that since an imperative program is often 90 percent assignment, and a functional program has no assignment, a functional program could be 90 percent shorter. However, not many people are convinced by such arguments or attracted to the ascetic world of stateless recursive programming, as John Hughes pointed out in his classic paper "Why Functional Programming Matters."

> *The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.*
>
> John Hughes, Chalmers University of Technology
> (http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html)

To see the advantages of functional programming, you must look at what FP permits rather than what it prohibits. For example, functional programming allows you to treat functions themselves as values and pass them to other functions. This might not seem all that important at first glance, but its implications are extraordinary. Eliminating the distinction between data and

function means that many problems can be more naturally solved. Functional programs can be shorter and more modular than corresponding imperative and object-oriented programs.

In addition to treating functions as values, functional languages offer other features that borrow from mathematics and are not commonly found in imperative languages. For example, functional programming languages often offer *curried functions*, where arguments can be passed to a function one at a time and, if all arguments are not given, the result is a residual function waiting for the rest of its parameters. It's also common for functional languages to offer type systems with much better power-to-weight ratios, providing more performance and correctness for less effort.

What Is F#?

Functional programming is the best approach to solving many thorny computing problems, but pure FP often isn't suitable for general-purpose programming. Because of this, FP languages have gradually embraced aspects of the imperative and OO paradigms, remaining true to the FP paradigm but incorporating features needed to easily write any kind of program. F# is a natural successor on this path. It is also much more than just an FP language.

Some of the most popular functional languages, including OCaml, Haskell, Lisp, and Scheme, have traditionally been implemented using custom runtimes, which leads to problems such as lack of interoperability. F# is a general-purpose programming language for .NET, a general-purpose runtime. F# smoothly integrates all three major programming paradigms. With F#, you can choose whichever paradigm works best to solve problems in the most effective way. You can do pure functional programming if you're a purist, but you can easily combine functional, imperative, and object-oriented styles in the same program and exploit the strengths of each paradigm. Like other typed functional languages, F# is strongly typed but also uses inferred typing so programmers don't need to spend time explicitly specifying types unless an ambiguity exists. Further, F# seamlessly integrates with the .NET Framework base class library (BCL). Using the BCL in F# is as simple as using it in C# or Visual Basic (and maybe even simpler).

F# was modeled on Objective Caml (OCaml), a successful object-oriented functional programming language, and then tweaked and extended to mesh well technically and philosophically with .NET. It fully embraces .NET and enables users to do everything that .NET allows. The F# compiler can compile for all implementations of the Common Language Infrastructure (CLI), it supports .NET generics without changing any code, and it even provides for inline Intermediate Language (IL) code. The F# compiler not only produces executables for any CLI, but can also run on any environment that has a CLI, which means F# is not limited to Windows but can run on Linux, Apple's OS X and iOS, as well as Google's Android OS.

The F# 2.0 compiler is distributed with Visual Studio 2012, Visual Studio 2010, and available as a plug-in for Visual Studio 2008. It supports IntelliSense expression completion and automatic expression checking. It also gives tooltips to show what types have been inferred for expressions. Programmers often comment that this really helps bring the language to life. F# 2.0 also has an open source release, licensed under the Apache License and is available from http://github.com/fsharp.

F# was fist implemented by Don Syme at Microsoft Research (MSR) in Cambridge. The project has now been embraced by Microsoft Corporate in Redmond, WA and the implementation of the compiler and Visual Studio integration is now developed by a team located in both Cambridge and Redmond. At the time of writing, the team was focused implementing F# 3.0, which is available in the Visual Studio "dev11" beta.

Although other FP languages run on .NET, F# has established itself as the de facto .NET functional programming language because of the quality of its implementation and its superb integration with .NET and Visual Studio.

No other .NET language is as easy to use and as flexible as F#!

Who Is Using F#?

F# has a strong presence inside Microsoft, both in MSR and throughout the company as a whole. Ralf Herbrich, coleader of MSR's Applied Games Group, which specializes in machine learning techniques, is typical of F#'s growing number of fans:

> *The first application was parsing 110GB of log data spread over 11,000 text files in over 300 directories and importing it into a SQL database. The whole application is 90 lines long (including comments!) and finished the task of parsing the source files and importing the data in under 18 hours; that works out to a staggering 10,000 log lines processed per second! Note that I have not optimized the code at all but written the application in the most obvious way. I was truly astonished as I had planned at least a week of work for both coding and running the application.*
>
> *The second application was an analysis of millions of feedbacks. We had developed the model equations and I literally just typed them in as an F# program; together with the reading-data-from-SQL-database and writing-results-to-MATLAB-data-file, the F# source code is 100 lines long (including comments). Again, I was astonished by the running time; the whole processing of the millions of data items takes 10 minutes on a standard desktop machine. My C# reference application (from some earlier tasks) is almost 1,000 lines long and is no faster. The whole job from developing the model equations to having first real world data results took 2 days.*
>
> Ralf Herbrich, Microsoft Research
> (http://blogs.msdn.com/dsyme/archive/2006/04/01/566301.aspx)

F# usage outside Microsoft is also rapidly growing. I asked Chance Coble, CTO at Cyfeon Solutions, about what F# brought to his work.

> *F# has made its case to me over and over again. The first project I decided to try F# on was a machine vision endeavor, which would identify and extract fingerprints from submitted fingerprint cards and load them into a biometrics system. The project plan was to perform the fingerprint extraction manually, which was growing cumbersome and the automation turned out to be a huge win (with very little code). Later we decided to include that F# work in a larger application that had been written in C#, and*

accomplished the integration with ease. Since then I have used F# in projects for machine learning, domain-specific language design, 3-D visualizations, symbolic analysis, and anywhere performance intensive data processing has been required. The ability to easily integrate functional modules into existing production scale applications makes F# not only fun to work with, but an important addition for project leads. Unifying functional programming with a mature and rich platform like .NET has opened up a great deal of opportunity.

Chance Coble, CTO at Cyfeon Solutions (private email)

Who Is This Book For?

This book is aimed primarily at IT professionals who want to get up to speed quickly on F#. A working knowledge of the .NET Framework and some knowledge of either C# or Visual Basic would be nice, but it's not necessary. All you really need is some experience programming in any language to be comfortable learning F#.

Even complete beginners who've never programmed before and are learning F# as their first computer language should find this book very readable. Though it doesn't attempt to teach introductory programming per se, it does carefully present all the important details of F#.

Chapter 2 First Steps in F#

This chapter will focus on a few general introductory details about the F# language and its programming environment. The next three chapters will focus on fleshing out the details of the language while this chapter will just offer a taste of what can be done. So don't worry if you don't understand all the details of the examples you see in this chapter, the rest of the book will fill them in.

Obtaining and Installing F#

The easiest and quickest way to get going with F# is to use Microsoft's Visual Studio. F# is included with Visual Studio 2012 and 2010. If you do not have a copy of Visual Studio, you can download a free 90-day trial version from http://www.microsoft.com/visualstudio/try.

F# is installed by default with both Visual Studio 2012 and 2010, so just installing Visual Studio with the default options should be enough. If you have Visual Studio installed and F# isn't available, you may have deactivated F# when installing Visual Studio. To activate F#, open **Control Panel** and go to the **Programs** menu.

If you don't want to use F# with Visual Studio you can download a command-line compiler from Microsoft at http://www.microsoft.com/download/en/details.aspx?id=11100 and use your favorite text editor to edit F# source files. As I believe Visual Studio is the best way for beginners to experience F#, the rest of this chapter will assume you're using Visual Studio, though all the examples will work with the command-line version of the compiler.

Hello World

As is traditional, let's start with a "hello world" program in F#. First we need to create a Visual Studio project to host our program. To do this, navigate to **File** > **New** > **Project…** and select an **F# Application**.

Note: F# comes with only four pre-installed application or library templates. However, there are many more templates available online. These online templates have been contributed both by the F# team at Microsoft and the F# community. They can be searched and installed via Visual Studio's New Project dialog.

Delete the contents in the **program.fs** file and enter the following line:

```
System.Console.WriteLine "Hello World"
```

Now press F5 to compile and execute the program and you'll see the console briefly pop up with the "Hello World" greeting. Notice how the program is only one line long—this part of the philosophy of F#, that code should be as free as possible from syntactic clutter, and you'll find

this is a philosophy shared by many functional programming languages. We simply want to be able to call the **`System.Console.WriteLine`** method and pass it a string literal, so these are the only two elements of the program we need.

Since the program exits straight after the greeting is written to the console, the greeting text is probably on the screen too briefly for us to see it. Let's fix that by reading a line from the console so the program will not exit until Enter is pressed:

```
open System
Console.WriteLine "Hello World"

Console.ReadLine()
```

Now press the F5 key again. When the program is executed this time, the greeting will stay until you press Enter and the program exits. Notice how we use the **`open`** keyword to open the **`System`** namespace. This allows us to remove the **`System`** from the beginning of the **`Console`** class' name, and the compiler will still be able to find the class as it will now look for it in the **`System`** namespace. The **`open`** keyword is very similar to the **`using`** keyword in C# when it is used to import namespaces.

Using F# Interactive

Visual Studio comes with an interactive version of F# called F# Interactive. This is sometimes referred to as a read–eval–print loop, or REPL for short. It gives F# the feeling of a dynamic language as the programmer is able to interactively evaluate parts of his or her program and see the results immediately, although it should be noted that F# Interactive dynamically compiles the portions of code you pass to it, so you should see a similar level of performance to compiled F# code. To use F# Interactive, simply highlight the section of code you want to evaluate and press Alt+Enter. You'll then see the results of this code printed in the F# Interactive window, usually located at the bottom of the screen. So if we highlight our initial "hello world" program and press Alt+Enter, we'd see the following results:

```
Hello World
val it : unit = ()
```

The first line is our greeting being output to the console. The second is some details about the program's type—don't worry too much about this for the moment. Types will be explained in a later chapter.

Being able to interactively execute code like this is one of my favorite features of F#. I think that being able to quickly try out ideas like this is a real productivity boost. So let's continue by looking at some other things you can do with F# Interactive, like creating interactive charts.

The F# team has created an F#-friendly wrapper for the **`System.Windows.Forms.DataVisualization.Charting.dll`**. The primary aim of this wrapper is to allow you to quickly show the data available in your program, or F# Interactive session, as a chart. It can be downloaded from http://code.msdn.microsoft.com/windowsdesktop/FSharpChart-b59073f5.

Once you unzip the downloaded **FSharpChart** folder, you will find the **FSharpChart.fsx** file inside the **F#** > **Scripts** folder. You'll need to ensure this script is in the same directory as the F# script you're working with, or modify the path to the script accordingly.

Now let's take a look at how we use an F# chart. The following example shows how to create a chart showing a simple linear line:

```
#load "FSharpChart.fsx"

open MSDN.FSharp.Charting

let data = [ 1; 2; 3; 4 ]
FSharpChart.Line data
```

On inputting this program into F# Interactive, again via Alt+Enter, you'll see a window pop up with the following chart:

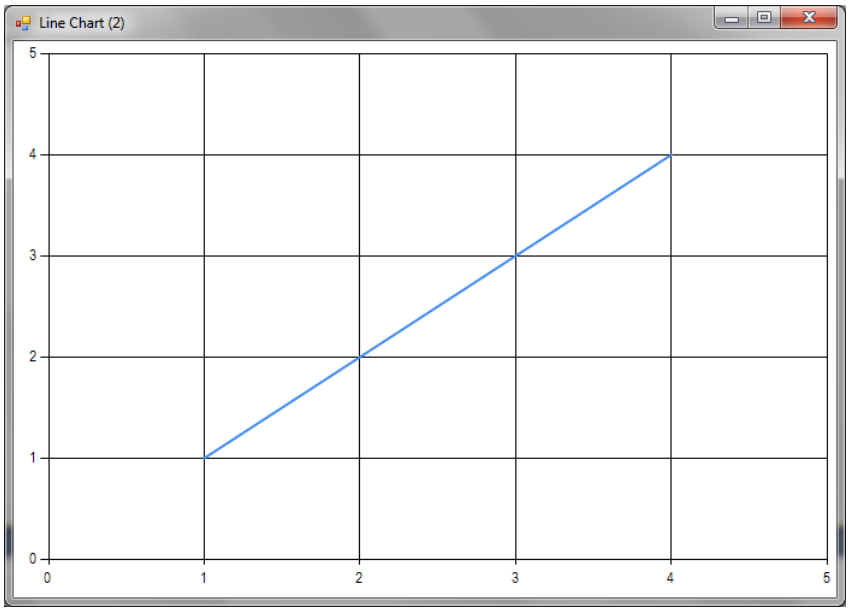

Figure 1: Line Chart in F# Interactive

Let's take a look at how this program works. The first line loads the charting script, a file called **`FSharpChart.fsx`**, into the F# Interactive session. This line can take a few seconds as the charting script is quite large, but you only need to load it once, and the functions will continue to be available in the interactive session. The next line imports the namespace of the charting functions we'll be working with. The following line creates a list of integers and binds them to the

data identifier. Finally, we pass our list to the charting function **FSharpChart.Line**, which draws a line graph. This is not the world's most exciting chart, so let's take a look at another.

The following code sample will create a column chart showing dates and a value at each date:

```
#load "FSharpChart.fsx"

open System
open MSDN.FSharp.Charting

let dateInApril day = new DateTime(2012, 03, day)

let data = [ dateInApril 6, 4; dateInApril 7, 8;
             dateInApril 8, 2; dateInApril 9, 3 ]
FSharpChart.Column data
```

Again, on inputting this program into F# Interactive you'll see a window pop up with the following chart:

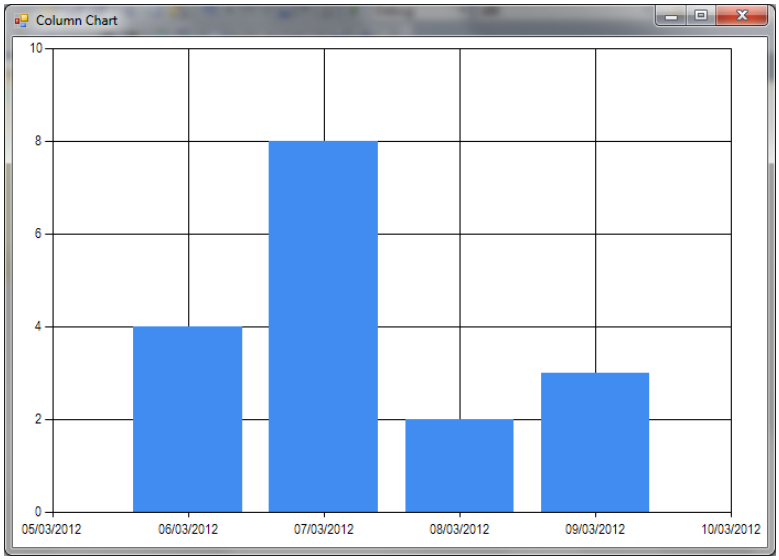

Figure 2: Column Chart in F# Interactive

The top parts of the program, the part loading the **FSharpChart.fsx** script and the **open** statements, are pretty much the same as before. The first major difference is that we define a function, **dateInApril**, to provide a shorthand way to create a **DateTime** object in April 2012. Next you'll notice our list of data is not single values, but pairs of values, referred to as *tuples*. Each pair contains a date object and an integer. Finally we pass our list of tuples to the charting function **FSharpChart.Column** which draws a column chart. While this chart is perhaps a little more interesting than the previous one, the example isn't very realistic because we're more likely to chart data from an external data source such as a text file.

So let's look at how we might load some data from a .csv file and chart it with F#:

```
#load "FSharpChart.fsx"

open System
open System.IO
open MSDN.FSharp.Charting

let treatLine (line: string) =
    let stringParts = line.Split(';')
    DateTime.Parse stringParts.[0], int stringParts.[1]

let data =
    File.ReadAllLines (__SOURCE_DIRECTORY__ + "\\mydata.txt")
    |> Array.map treatLine

FSharpChart.Column data
```

Yet again, the top part of the program changes little. After the **open** statements we define a function called **treatLine** that splits a line into two, parsing the first part as dates and the second as integers. Next we use .NET's **File.ReadAllLines** function to read all the data from a text file called **mydata.txt**. After that we use the **Array.map** function to pass every line in the text file to our **treatLine** function and create a new array—this is very similar to using the LINQ extension method **Select** in C#. Finally, we pass the results to the **FSharpChart.Column** to draw the graph.

Summary

This chapter has given you a very brief introduction to using F#, both to create compiled programs and using F# Interactive to quickly test ideas. The remainder of the book will be a guide on how to program in F# by taking a detailed look at the language's syntax and features.

Chapter 3 Functional Programming

You saw in Chapter 1 that pure functional programming treats everything as a value, including functions. Although F# is not a pure functional language, it does encourage you to program in the functional style; that is, it encourages you to use expressions and computations that return a result, rather than statements that result in some side effect. In this chapter, we'll survey the major language constructs of F# that support the functional programming paradigm and learn how they make it easier to program in the functional style.

Literals

Literals represent constant values and are useful building blocks for computations. F# has a rich set of literals, which we will see in the next example.

In F#, string literals can contain newline characters, and regular string literals can contain standard escape codes. Verbatim string literals use a backslash (\) as a regular character, and two double quotes (**""**) are the escape code for a quote. You can define all integer types using hexadecimal and octal by using the appropriate prefix and postfix indicator. The following example shows some of these literals in action being bound to *identifiers*, which are described in the section Identifiers and let Bindings a little later in this chapter.

```
// Some strings.
let message = "Hello
World\r\n\t!"
let dir = @"c:\projects"

// A byte array.
let bytes = "bytesbytesbytes"B

// Some numeric types.
let xA = 0xFFy
let xB = 0o7777un
let xC = 0b10010UL
```

Functions

In F#, functions are defined using the keyword **fun**. The function's arguments are separated by spaces, and the arguments are separated from the function body by an ASCII arrow (**->**).

Here is an example of a function that takes two values and adds them together:

```
fun x y -> x + y
```

Notice that this function does not have a name; this is a sort of function literal. Functions defined in this way are referred to as *anonymous functions*, *lambda functions*, or just *lambdas*.

The idea that a function does not need a name may seem a little strange. However, if a function is to be passed as an argument to another function, it may not need a name, especially if the task it's performing is relatively simple.

If you need to give the function a name, you can bind it to an identifier, as described in the next section.

Identifiers and let Bindings

Identifiers are the way you give names to values in F# so you can refer to them later in a program. You define an identifier using the keyword **`let`** followed by the name of the identifier, an equal sign, and an expression that specifies the value to which the identifier refers. An expression is any piece of code that represents a computation that will return a value. The following expression shows a value being assigned to an identifier:

```
let x = 42
```

To most people coming from an imperative programming background, this will look like a variable assignment. There are many similarities, but a key difference is that in pure functional programming, once a value is assigned to an identifier, it does not change. This is why I will refer to them throughout this book as *identifiers*, rather than *variables*.

Note: Under some circumstances you can redefine identifiers. This may look a little like an identifier changing value, but it is subtly different. Also, in imperative programming in F#, the value of an identifier can change in some circumstances. In this chapter, we focus on functional programming in which identifiers do not change their values.

An identifier can refer to either a value or a function, and since F# functions are really values in their own right, this is hardly surprising. This means F# has no real concept of a function name or parameter name; these are just identifiers. You can bind an anonymous function to an identifier the same way you can bind a string or integer literal to an identifier:

```
let myAdd = fun x y -> x + y
```

However, as it is very common to need to define a function with a name, F# provides a short syntax for this. You write a function definition the same way as a value identifier, except that a function has two or more identifiers between the **let** keyword and the equal sign, as follows:

```
let raisePowerTwo x = x ** 2.0
```

The first identifier is the name of the function, **raisePowerTwo**, and the identifier that follows it is the name of the function's parameter, **x**. If a function has a name, it is strongly recommended that you use this shorter syntax for defining it.

The syntax for declaring *values* and *functions* in F# is indistinguishable because functions *are* values, and F# syntax treats them both similarly. For example, consider the following code:

```
let n = 10
let add a b = a + b

let result = add n 4
printfn "%i" (result)
```

On the first line, the value **10** is assigned to the identifier **n**. On the second line, an **add** function, which takes two arguments and adds them together, is defined. Notice how similar the syntax is, with the only difference being that a function has parameters that are listed after the function name. Since everything is a value in F#, the literal **10** on the first line is a value, and the result of the expression **a + b** on the next line is also a value that automatically becomes the result of the **add** function. Note that there is no need to explicitly return a value from a function as you would in an imperative language.

Identifier Names

There are some rules governing identifier names. Identifiers must start with an underscore (_) or a letter, and can then contain any alphanumeric character, underscore, or a single quotation mark (**'**). Keywords cannot be used as identifiers. As F# supports the use of a single quotation mark as part of an identifier name, you can use this to represent "prime" to create identifier names for different but similar values, as in this example:

```
let x = 42
let x' = 43
```

F# supports Unicode, so you can use accented characters and letters from non-Latin alphabets as identifier names:

```
let 标识符 = 42
```

If the rules governing identifier names are too restrictive, you can use double tick marks (``) to quote the identifier name. This allows you to use any sequence of characters—as long as it doesn't include tabs, newlines, or double ticks—as an identifier name. This means you could create an identifier that ends with a question mark, for example (some programmers believe it is useful to put a question mark at the end of names that represent Boolean values):

```
let ``more? `` = true
```

This can also be useful if you need to use a keyword as an identifier or type name:

```
let ``class`` = "style"
```

For example, you might need to use a member from a library that was not written in F# and has one of F#'s keywords as its name. Generally, it's best to avoid overuse of this feature, as it could lead to libraries that are difficult to use from other .NET languages.

Scope

The *scope* of an identifier defines where you can use an identifier (or a type, as discussed in the Defining Types section in the next chapter) within a program. It is important to have a good understanding of scope, because if you try to use an identifier that's not in scope, you will receive a compile error.

All identifiers—whether they relate to functions or values—are scoped from the end of their definitions until the end of the sections in which they appear. So, for identifiers that are at the top level (that is, identifiers that are not local to another function or other value), the scope of the identifier is from the place where it's defined to the end of the source file. Once an identifier at the top level has been assigned a value (or function), this value cannot be changed or redefined. An identifier is available only after its definition has ended, meaning that it is not usually possible to define an identifier in terms of itself.

You will have noticed that in F#, you never need to explicitly return a value; the result of the computation is automatically bound to its associated identifier. So, how do you compute intermediate values within a function? In F#, this is controlled by whitespace. An indentation creates a new scope, and the end of this scope is signaled by the end of the indentation. Indentation means that the **`let`** binding is an intermediate value in the computation that is not visible outside this scope. When a scope closes (by the indentation ending) and an identifier is no longer available, it is said to *drop out of scope* or to be *out of scope*.

To demonstrate scope, the following example shows a function that computes the point halfway between two integers. The third and fourth lines show intermediate values being calculated.

```
// Function to calculate a midpoint.
let halfWay a b =
    let dif = b - a
    let mid = dif / 2
    mid + a

printfn "%i" (halfWay 10 20)
```

First, the difference between the two numbers is calculated, and this is assigned to the identifier **`dif`** using the **`let`** keyword. To show that this is an intermediate value within the function, it is indented by four spaces. The choice of the number of spaces is left to the programmer, but the convention is four. After that, the example calculates the midpoint, assigning it to the identifier **`mid`** using the same indentation. Finally, the desired result of the function is the midpoint plus **`a`**, so the code can simply say **`mid + a`**, and this becomes the function's result.

Note: You cannot use tabs instead of spaces for indenting, because these can look different in different text editors, which causes problems when whitespace is significant.

Capturing Identifiers

You have already seen that in F#, you can define functions within other functions. These functions can use any identifier in scope, including definitions that are also local to the function where they are defined. Because these inner functions are values, they could be returned as the result of the function or passed to another function as an argument. This means that although an identifier is defined within a function such that it is not visible to other functions, its actual lifetime may be much longer than the function in which it is defined. Let's look at an example to illustrate this point. Consider the following function, defined as **`calculatePrefixFunction`**:

```
// Function that returns a function to
let calculatePrefixFunction prefix =
    // calculate prefix.
    let prefix' = Printf.sprintf "[%s]: " prefix
    // Define function to perform prefixing.
    let prefixFunction appendee =
        Printf.sprintf "%s%s" prefix' appendee
    // Return function.
    prefixFunction

// Create the prefix function.
let prefixer = calculatePrefixFunction "DEBUG"
```

```
// Use the prefix function.
printfn "%s" (prefixer "My message")
```

This function returns the inner function it defines, **prefixFunction**. The identifier **prefix'** is defined as local to the scope of the function **calculatePrefixFunction**; it cannot be seen by other functions outside **calculatePrefixFunction**. The inner function **prefixFunction** uses **prefix'**, so when **prefixFunction** is returned, the value **prefix'** must still be available. **calculatePrefixFunction** creates the function **prefixer**. When **prefixer** is called, you see that its result uses a value that was calculated and associated with **prefix'**:

```
[DEBUG]: My message
```

Although you should have an understanding of this process, most of the time you don't need to think about it because it doesn't involve any additional work by the programmer. The compiler will automatically generate a *closure* to handle extending the lifetime of the local value beyond the function in which it is defined. The .NET garbage collection will automatically handle clearing the value from memory. Understanding this process of identifiers being captured in closures is probably more important when programming in imperative style where an identifier can represent a value that changes over time. When programming in the functional style, identifiers will always represent values that are constant, making it slightly easier to figure out what has been captured in a closure.

Recursion

Recursion means defining a function in terms of itself; in other words, the function calls itself within its definition. Recursion is often used in functional programming where you would use a loop in imperative programming. Many believe that algorithms are much easier to understand when expressed in terms of recursion rather than loops.

To use recursion in F#, use the **rec** keyword after the **let** keyword to make the identifier available within the function definition. The following example shows recursion in action. Notice how on the fifth line the function makes two calls to itself as part of its own definition.

```
// A function to generate the Fibonacci numbers.
let rec fib x =
    match x with
    | 1 -> 1
    | 2 -> 1
    | x -> fib (x - 1) + fib (x - 2)

// Call the function and print the results.
```

```
printfn "(fib 2) = %i" (fib 2)
printfn "(fib 6) = %i" (fib 6)
printfn "(fib 11) = %i" (fib 11)
```

This function calculates the *n*th term in the Fibonacci sequence. The Fibonacci sequence is generated by adding the previous two numbers in the sequence, and it progresses as follows: 1, 1, 2, 3, 5, 8, 13…. Recursion is most appropriate for calculating the Fibonacci sequence, because the definition of any number in the sequence, other than the first two, depends on being able to calculate the previous two numbers, so the Fibonacci sequence is defined in terms of itself.

Although recursion is a powerful tool, you should be careful when using it. It is easy to inadvertently write a recursive function that never terminates. Although intentionally writing a program that does not terminate is sometimes useful, it is rarely the goal when trying to perform calculations. To ensure that recursive functions terminate, it is often useful to think of recursion in terms of a base case and a recursive case:

- The *recursive case* is the value for which the function is defined in terms of itself. For the function **`fib`**, this is any value other than 1 and 2.
- The *base case* is the non-recursive case; that is, there must be some value where the function is not defined in terms of itself. In the **`fib`** function, 1 and 2 are the base cases.

Having a base case is not enough in itself to ensure termination. The recursive case must tend toward the base case. In the **`fib`** example, if **`x`** is greater than or equal to 3, then the recursive case will tend toward the base case because **`x`** will always become smaller and at some point reach 2. However, if **`x`** is less than 1, then **`x`** will grow continually more negative, and the function will repeat until the limits of the machine are reached, resulting in a stack overflow error (**`System.StackOverflowException`**).

The previous code also uses F# pattern matching, which is discussed in the Pattern Matching section later in this chapter.

Operators

In F#, you can think of *operators* as a more aesthetically pleasing way to call functions.

F# has two different kinds of operators:

- A *prefix* operator is an operator where the operands come after the operator.
- An *infix* operator sits in between the first and second operands.

F# provides a rich and diverse set of operators that you can use with numeric, Boolean, string, and collection types. The operators defined in F# and its libraries are too numerous to be

covered in this section, so rather than looking at individual operators, we'll look at how to use and define operators in F#.

As in C#, F# operators are overloaded, meaning you can use more than one type with an operator. However, unlike in C#, both operands must be the same type, or the compiler will generate an error. F# also allows users to define and redefine operators.

Operators follow a set of rules similar to C#'s for operator overloading resolution; therefore, any class in the BCL or any .NET library that was written to support operator overloading in C# will support it in F#. For example, you can use the **+** operator to concatenate strings, as well as to add a **`System.TimeSpan`** to a **`System.DateTime`**, because these types support an overload of the **+** operator. The following example illustrates this:

```
let rhyme = "Jack " + "and " + "Jill"
printfn "%string" rhyme

open System
let oneYearLater =
    DateTime.Now + new TimeSpan(365, 0, 0, 0, 0)
printfn "%A" oneYearLater
```

Unlike functions, operators are not values, so they cannot be passed to other functions as parameters. However, if you need to use an operator as a value, you can do this by surrounding it with parentheses. The operator will then behave exactly like a function. Practically, this has two consequences:

- The operator is now a function, and its parameters will appear after the operator:

```
let result = (+) 1 1
```

- As it is a value, it could be returned as the result of a function, passed to another function, or bound to an identifier. This provides a very concise way to define the **`add`** function:

```
let add = (+)
```

You'll see how using an operator as a value can be useful later in this chapter when we look at working with lists.

Function Application

Function application, also sometimes referred to as *function composition* or *composing functions*, simply means calling a function with some arguments. The following example shows

the **add** function being defined and then applied to two arguments. Notice that the arguments are not separated with parentheses or commas; only whitespace is needed to separate them.

```
let add x y = x + y

let result = add 4 5

printfn "(add 4 5) = %i" result
```

The results of this example, when compiled and executed, are as follows:

```
(add 4 5) = 9
```

In F#, a function has a fixed number of arguments and is applied to the value that appears next in the source file. You do not necessarily need to use parentheses when calling functions, but F# programmers often use them to define which function should be applied to which arguments. Consider the simple case where you want to add four numbers using the **add** function. You could bind the result of each function call to a new identifier, but for such a simple calculation this would be very cumbersome:

```
let add x y = x + y

let result1 = add 4 5
let result2 = add 6 7

let finalResult = add result1 result2
```

Instead, it is often better to pass the result of one function directly to the next function. To do this, use parentheses to show which parameters are associated with which functions:

```
let add x y = x + y

let result =
    add (add 4 5) (add 6 7)
```

Here, the second and third occurrences of the **add** function are grouped with the parameters **4**, **5** and **6**, **7**, respectively, and the first occurrence of the **add** function will act on the results of the other two functions.

F# also offers another way to compose functions using the *pipe-forward* operator (**|>**). This operator has the following definition:

```
let (|>) x f = f x
```

This simply means it takes a parameter, **x**, and applies it to the given function, **f**, so that the parameter is now given before the function. The following example shows a parameter, **0.5**, being applied to the function **System.Math.Cos** using the pipe-forward operator:

```
let result = 0.5 |> System.Math.Cos
```

This reversal can be useful in some circumstances, especially when you want to chain many functions together. Here is the previous **add** function example rewritten using the pipe-forward operator:

```
let add x y = x + y

let result = add 6 7 |> add 4 |> add 5
```

Some programmers think this style is more readable, as it has the effect of making the code read in a more right-to-left manner. The code should now be read as “add 6 to 7, forward this result to the next function which will add 4, and then forward this result to a function that will add 5.”

This example also takes advantage of the capability to partially apply functions in F#, which is discussed in the next section.

Partial Application of Functions

F# supports the partial application of functions (these are sometimes called *partial* or *curried* functions). This means you don’t need to pass all the arguments to a function at once. Notice that the final example in the previous section passes a single argument to the **add** function, which takes two arguments. This is very much related to the idea that functions are values. So we can create an **add** function, pass one argument to it, and bind the resulting function to a new identifier:

```
let add x y = x + y

let addFour = add 4
```

Because a function is just a value, if it doesn't receive all its arguments at once it returns a value that is a new function waiting for the rest of the arguments. So in the example, passing just the value **4** to the **add** function results in a new function. I named the function **addFour** because it takes one parameter and adds the value **4** to it. At first glance, this idea can look uninteresting and unhelpful, but it is a powerful part of functional programming that you'll see used throughout the book.

Pattern Matching

Pattern matching allows you to look at the value of an identifier and then make different computations depending on its value. It might be compared to the **switch** statement in C++ and C#, but it is much more powerful and flexible. Programs that are written in the functional style tend to be written as series of transformations applied to the input data. Pattern matching allows you to analyze the input data and decide which transformation should be applied to it, so pattern matching fits in well with programming in the functional style.

The pattern matching construct in F# allows you to pattern match over a variety of types and values. It also has several different forms and crops up in several places in the language.

The simplest form of pattern matching is matching over a value. You have already seen this in the Recursion section of this chapter, where it was used to implement a function that generated numbers in the Fibonacci sequence. To illustrate the syntax, the next example shows an implementation of a function that will produce the Lucas numbers, a sequence of numbers as follows: 1, 3, 4, 7, 11, 18, 29, 47, 76…. The Lucas sequence has the same definition as the Fibonacci sequence; only the starting points are different.

```
// Definition of Lucas numbers using pattern matching.
let rec luc x =
    match x with
    | x when x <= 0 -> failwith "value must be greater than 0"
    | 1 -> 1
    | 2 -> 3
    | x -> luc (x - 1) + luc (x - 2)
```

The syntax for pattern matching uses the keyword **match**, followed by the identifier that will be matched, then the keyword **with**, then all the possible matching rules separated by pipes (**|**). In the simplest case, a rule consists of either a constant or an identifier, followed by an arrow (**->**), and then the expression to be used when the value matches the rule. In this definition of the function **luc**, the second and third cases are literals—the values **1** and **2**—and these will be replaced with the values **1** and **3**, respectively. The fourth case will match any value of **x** greater than 2, and this will cause two further calls to the **luc** function.

The rules are matched in the order in which they are defined, and the compiler will issue an error if pattern matching is incomplete; that is, if there is some possible input value that will not match any rule. This would be the case in the **luc** function if you had omitted the final rule,

because any values of **x** greater than 2 would not match any rule. The compiler will also issue a warning if there are any rules that will never be matched, typically because there is another rule in front of them that is more general. This would be the case in the **luc** function if the fourth rule were moved ahead of the first rule. In this case, none of the other rules would ever be matched because the first rule would match any value of **x**.

You can add a **when** guard (as in the first rule in the example) to give precise control over when a rule fires. A **when** guard is composed of the keyword **when** followed by a Boolean expression. Once the rule is matched, the **when** clause is evaluated, and the rule will fire only if the expression evaluates to **true**. If the expression evaluates to **false**, the remaining rules will be searched for another match. The first rule is designed to be the function's error handler. The first part of the rule is an identifier that will match any integer, but the **when** guard means the rule will match only those integers that are less than or equal to zero.

If you want, you can omit the first **|**. This can be useful when the pattern match is small and you want to fit it on one line. You can see this in the next example, which also demonstrates the use of the underscore (_) as a *wildcard*.

```
let booleanToString x =
    match x with false -> "False" | _ -> "True"
```

The **_** will match any value and is a way of telling the compiler that you're not interested in using this value. For example, in this **booleanToString** function, you do not need to use the constant **true** in the second rule, because if the first rule is matched you know that the value of **x** will be **true**. Moreover, you do not need to use **x** to derive the string **"True"**, so you can ignore the value and just use **_** as a wildcard.

Another useful feature of pattern matching is that you can combine two patterns into one rule through the use of the pipe (**|**). The next example, **stringToBoolean**, demonstrates this.

```
// Function for converting a Boolean to a string.
let booleanToString x =
    match x with false -> "False" | _ -> "True"

// Function for converting a string to a Boolean.
let stringToBoolean x =
    match x with
    | "True" | "true" -> true
    | "False" | "false" -> false
    | _ -> failwith "unexpected input"
```

The first two rules have two strings that should evaluate to the same value, so rather than having two separate rules, you can just use **|** between the two patterns.

It is also possible to pattern match over most of the types defined by F#. The next two examples demonstrate pattern matching over tuples, with two functions that implement a Boolean And and Or using pattern matching. Each takes a slightly different approach.

```
let myOr b1 b2 =
    match b1, b2 with
    | true, _ -> true
    | _, true -> true
    | _ -> false

let myAnd p =
    match p with
    | true, true -> true
    | _ -> false
```

The **myOr** function has two Boolean parameters that are placed between the **match** and **with** keywords and separated by commas to form a tuple. The **myAnd** function has one parameter, which is itself a tuple. Either way, the syntax for creating pattern matches for tuples is the same and similar to the syntax for creating tuples.

If it's necessary to match values within the tuple, the constants or identifiers are separated by commas, and the position of the identifier or constant defines what it matches within the tuple. This is shown in the first and second rules of the **myOr** function and in the first rule of the **myAnd** function. These rules match parts of the tuples with constants, but you could use identifiers if you want to work with the separate parts of the tuple later in the rule definition. Just because you're working with tuples doesn't mean you always need to look at the various parts that make up the tuple.

The third rule of **myOr** and the second rule of **myAnd** show the whole tuple matched with a single **_** wildcard character. This, too, could be replaced with an identifier if you want to work with the value in the second half of the rule definition.

Because pattern matching is such a common task in F#, the language provides alternative shorthand syntax. If the sole purpose of a function is to pattern match over something, then it may be worth using this syntax. In this version of the pattern-matching syntax, you use the keyword **function**, place the pattern where the function's parameters would usually go, and then separate all the alternative rules with **|**. The next example shows this syntax in action in a simple function that recursively processes a list of strings and concatenates them into a single string.

```
// Concatenate a list of strings into a single string.
let rec conactStringList =
    function head :: tail -> head + conactStringList tail
           | [] -> ""
```

```
// Test data.
let jabber = ["'Twas "; "brillig, "; "and "; "the "; "slithy "; "toves ";
"..."]
// Call the function.
let completJabber = conactStringList jabber
// Print the result.
printfn "%s" completJabber
```

The results of this example, when compiled and executed, are as follows:

```
'Twas brillig, and the slithy toves ...
```

Pattern matching is one of the fundamental building blocks of F#, and we'll return to it several times in this book. We'll look at pattern matching over lists with record types and union types in the next chapter.

Control Flow

F# has a strong notion of *control flow*. In this way, it differs from many pure functional languages, where the notion of control flow is very loose, because expressions can be evaluated in essentially any order. The strong notion of control flow is apparent in the **`if… then… else…`** expression.

In F#, the **`if… then… else…`** construct is an expression, meaning it returns a value. One of two different values will be returned, depending on the value of the Boolean expression between the **`if`** and **`then`** keywords. The next example illustrates this. The **`if… then… else…`** expression is evaluated to return either **`"heads"`** or **`"tails"`** depending on whether the program is run on an even second or an odd second.

```
let result =
    if System.DateTime.Now.Second % 2 = 0 then
        "heads"
    else
        "tails"

printfn "%A" result
```

It's interesting to note that the **`if… then… else…`** expression is just convenient shorthand for pattern matching over a Boolean value. The previous example could be rewritten as follows:

```
let result =
    match System.DateTime.Now.Second % 2 = 0 with
    | true -> "heads"
    | false ->  "tails"

printfn "%A" result
```

The **if… then… else…** expression has some implications that you might not expect if you are more familiar with imperative-style programming. F#'s type system requires that the values being returned by the **if… then… else…** expression must be the same type, or the compiler will generate an error. So, if in the previous example, you replaced the string **"tails"** with an integer or Boolean value, you would get a compile error. If you really require the values to be of different types, you can create an **if… then… else…** expression of type **obj** (F#'s version of **System.Object**) as shown in the next example, which prints either **"heads"** or **false** to the console.

```
let result =
    if System.DateTime.Now.Second % 2 = 0 then
        box "heads"
    else
        box false

printfn "%A" result
```

Imperative programmers may be surprised that an **if… then… else…** expression must have an **else** if the expression returns a value. This is logical when you consider the examples you've just seen. If the **else** were removed from the code, the identifier **result** could not be assigned a value when the **if** evaluated to false, and having uninitialized identifiers is something that F# (and functional programming in general) aims to avoid.

Lists

F# *lists* are simple collection types that are built into F#. An F# list can be an *empty list*, represented by square brackets (**[]**), or it can be another list with a value concatenated to it. You concatenate values to the front of an F# list using a built-in operator that consists of two colons (**::**), pronounced "cons." The next example shows some lists being defined, starting with an empty list on the first line, followed by two lists where strings are placed at the front by concatenation:

```
let emptyList = []
let oneItem = "one " :: []
let twoItem = "one " :: "two " :: []
```

The syntax to add items to a list by concatenation is a little verbose, so if you just want to define a list, you can use shorthand. In this shorthand notation, you place the list items between square brackets and separate them with a semicolon (**;**), as follows:

```
let shortHand = ["apples "; "pears"]
```

Another F# operator that works on lists is the “at” symbol (**@**), which you can use to concatenate two lists together, as follows:

```
let twoLists = ["one, "; "two, "] @ ["buckle "; "my "; "shoe "]
```

All items in an F# list must be of the same type. If you try to place items of different types in a list—for example, you try to concatenate a string to a list of integers—you will get a compile error. If you need a list of mixed types, you can create a list of type **obj** (the F# equivalent of **System.Object**), as in the following code sample:

```
// The empty list.
let emptyList = []

// List of one item.
let oneItem = "one " :: []

// List of two items.
let twoItem = "one " :: "two " :: []

// List of two items.
let shortHand = ["apples "; "pairs "]

// Concatenation of two lists.
let twoLists = ["one, "; "two, "] @ ["buckle "; "my "; "shoe "]

// List of objects.
let objList = [box 1; box 2.0; box "three"]
```

I discuss types in F# in more detail in the next chapter, Types and Type Inference.

F# lists are *immutable*. In other words, once a list is created, it cannot be altered. The functions and operators that act on lists do not alter them, but they create a new, modified version of the list, leaving the old list available for later use if needed. The next example shows this.

```
// Create a list of one item.
let one = ["one "]
// Create a list of two items.
```

```
let two = "two " :: one
// Create a list of three items.
let three = "three " :: two

// Reverse the list of three items.
let rightWayRound = List.rev three

printfn "%A" one
printfn "%A" two
printfn "%A" three
printfn "%A" rightWayRound
```

An F# list containing a single string is created, and then two more lists are created, each using the previous one as a base. Finally, the **`List.rev`** function is applied to the last list to create a new reversed list.

Pattern Matching against Lists

The regular way to work with F# lists is to use pattern matching and recursion. The pattern-matching syntax for pulling the head item off a list is the same as the syntax for concatenating an item to a list. The pattern is formed by the identifier representing the head, followed by **`::`**, and then the identifier for the rest of the list. You can see this in the first rule of **`concatList`** in the next example. You can also pattern match against list constants; you can see this in the second rule of **`concatList`**, where there is an empty list.

```
// List to be concatenated.
let listOfList = [[2; 3; 5]; [7; 11; 13]; [17; 19; 23; 29]]

// Definition of a concatenation function.
let rec concatList l =
    match l with
    | head :: tail -> head @ (concatList tail)
    | [] -> []

// Call the function.
let primes = concatList listOfList

// Print the results.
printfn "%A" primes
```

Taking the head from a list, processing it, and then recursively processing the tail of the list is the most common way of dealing with lists via pattern matching, but it certainly isn't the only thing you can do with pattern matching and lists. The following example shows a few other uses of this combination of features.

```
// Function that attempts to find various sequences.
let rec findSequence l =
    match l with
    // Match a list containing exactly 3 numbers.
    | [x; y; z] ->
        printfn "Last 3 numbers in the list were %i %i %i"
            x y z
    // Match a list of 1, 2, 3 in a row.
    | 1 :: 2 :: 3 :: tail ->
        printfn "Found sequence 1, 2, 3 within the list"
        findSequence tail
    // If neither case matches and items remain,
    // recursively call the function.
    | head :: tail -> findSequence tail
    // If no items remain, terminate.
    | [] -> ()

// Some test data.
let testSequence = [1; 2; 3; 4; 5; 6; 7; 8; 9; 8; 7; 6; 5; 4; 3; 2; 1]

// Call the function.
findSequence testSequence
```

The first rule demonstrates how to match a list of a fixed length—in this case, a list of three items. Here, identifiers are used to grab the values of these items so they can be printed to the console. The second rule looks at the first three items in the list to see whether they are the sequence of integers 1, 2, 3; and if they are, it prints a message to the console. The final two rules are the standard head and tail treatment of a list, designed to work their way through the list, doing nothing if there is no match with the first two rules.

The results of this example, when compiled and executed, are as follows:

```
Found sequence 1, 2, 3 within the list
Last 3 numbers in the list were 3 2 1
```

Although pattern matching is a powerful tool for the analysis of data in lists, it's often not necessary to use it directly. The F# libraries provide a number of higher-order functions for working with lists that implement the pattern matching for you, so you don't need to repeat the code. To illustrate this, imagine you need to write a function that adds one to every item in a list. You can easily write this using pattern matching:

```
let rec addOneAll list =
    match list with
    | head :: rest ->
        head + 1 :: addOneAll rest
    | [] -> []

printfn "(addOneAll [1; 2; 3]) = %A" (addOneAll [1; 2; 3])
```

The results of this example, when compiled and executed, are as follows:

```
(addOneAll [1; 2; 3]) = [2; 3; 4]
```

However, the code is perhaps a little more verbose than you would like for such a simple problem. The clue to solving this comes from noticing that adding one to every item in the list is just an example of a more general problem: the need to apply some transformation to every item in a list. The F# core library contains a **`map`** function which is defined in the **`List`** module. It has the following definition:

```
let rec map func list =
    match list with
    | head :: rest ->
        func head :: map func rest
    | [] -> []
```

You can see that the **`map`** function has a very similar structure to the **`addOneAll`** function from the previous example. If the list is not empty, you take the head item of the list and apply the function, **`func`**, you are given as a parameter. This is then appended to the results of recursively calling **`map`** on the rest of the list. If the list is empty, you simply return the empty list. The **`map`** function can then be used to implement adding one to all items in a list in a much more concise manner:

```
let result = List.map ((+) 1) [1; 2; 3]
printfn "List.map ((+) 1) [1; 2; 3] = %A" result
```

Also note that this example uses the **`add`** operator as a function by surrounding it with parentheses, as described earlier in this chapter in the Operators section. This function is then partially applied by passing its first parameter but not its second. This creates a function that takes an integer and returns an integer, which is passed to the **`map`** function.

The **`List`** module contains many other interesting functions for working with lists, such as **`List.filter`**, which allows you to filter a list using a predicate, and **`List.fold`**, which is used to create a summary of a list.

Summary

This chapter has given you a short introduction to using F#'s functional features. These provide the programmer with a powerful but flexible way to create programs.

Chapter 4 Types and Type Inference

F# is a *strongly typed* language, which means you cannot use a function with a value that is inappropriate. You cannot call a function that has a string as a parameter with an integer argument; you must explicitly convert between the two. The way the language treats the type of its values is referred to as its *type system*. F# has a type system that does not get in the way of routine programming. In F#, all values have a type, and this includes values that are functions.

Type Inference

Ordinarily, you don't need to explicitly declare types; the compiler will work out the type of a value from the types of the literals in the function and the resulting types of other functions it calls. If everything is okay, the compiler will keep the types to itself; only if there is a type mismatch will the compiler inform you by reporting a compile error. This process is generally referred to as *type inference*. If you want to know more about the types in a program, you can make the compiler display all inferred types with the **`-i`** switch. Visual Studio users get tooltips that show types when they hover the mouse pointer over an identifier.

The way type inference works in F# is fairly easy to understand. The compiler works through the program, assigning types to identifiers as they are defined, starting with the top leftmost identifier and working its way down to the bottom rightmost. It assigns types based on the types it already knows—that is, the types of literals and (more commonly) the types of functions defined in other source files or assemblies.

The next example defines two F# identifiers and then shows their inferred types displayed on the console with the F# compiler's **`-i`** switch.

```
let aString = "Spring time in Paris"
let anInt = 42
```

```
val aString : string
val anInt : int
```

The types of these two identifiers are unsurprising—**`string`** and **`int`**, respectively. The syntax used by the compiler to describe them is fairly straightforward: the keyword **`val`** (meaning "value") and then the identifier, a colon, and finally the type.

The definition of the function **`makeMessage`** in the next example is a little more interesting.

```
let makeMessage x = (Printf.sprintf "%i" x) + " days to spring time"
let half x = x / 2
```

```
val makeMessage : int -> string
val half : int -> int
```

Note that the **makeMessage** function's definition is prefixed with the keyword **val**, just like the two values you saw before; even though it is a function, the F# compiler still considers it to be a value. Also, the type itself uses the notation **int -> string**, meaning a function takes an integer and returns a string. The **->** (ASCII arrow) between the type names represents the transformation of the function being applied. The arrow represents a transformation of the value, but not necessarily the type, because it can represent a function that transforms a value into a value of the same type, as shown in the **half** function on the second line.

The types of functions that can be partially applied and functions that take tuples differ. The following functions, **div1** and **div2**, illustrate this.

```
let div1 x y = x / y
let div2 (x, y) = x / y

let divRemainder x y = x / y, x % y
```

```
val div1 : int -> int -> int
val div2 : int * int -> int
val divRemainder : int -> int -> int * int
```

The function **div1** can be partially applied, and its type is **int -> int -> int**, representing that the arguments can be passed in separately. Compare this with the function **div2**, which has the type **int * int -> int**, meaning a function that takes a pair of integers—a tuple of integers—and turns them into a single integer. You can see this in the function **div_remainder**, which performs integer division and also returns the remainder at the same time. Its type is **int -> int -> int * int**, meaning a curried function that returns an integer tuple.

The next function, **doNothing**, looks inconspicuous enough, but it is quite interesting from a typing point of view.

```
let doNothing x = x
```

```
val doNothing : 'a -> 'a
```

This function has the type **'a -> 'a**, meaning it takes a value of one type and returns a value of the same type. Any type that begins with a single quotation mark (**'**) means a *variable* type. F# has a type, **obj**, that maps to **System.Object** and represents a value of any type—a concept that you will probably be familiar with from other common language runtime (CLR)-based programming languages (and indeed, many languages that do not target the CLR). However, a variable type is not the same. Notice how the type has an **'a** on both sides of the arrow. This means that, even though the compiler does not yet know the type, it knows that the type of the return value will be the same as the type of the argument. This feature of the type system, sometimes referred to as *type parameterization*, allows the compiler to find more type errors at compile time and can help avoid casting.

Note: The concept of a variable type, or type parameterization, is closely related to the concept of* generics *that were introduced in CLR version 2.0 and have now become part of the ECMA specification for CLI version 2.0. When F# targets a CLI that has generics enabled, it takes full advantage of them by using them anywhere it finds an undetermined type. Don Syme, the creator of F#, designed and implemented generics in the .NET CLR before he started working on F#. One might be tempted to infer that he did this so he could create F#!

The function **doNothingToAnInt**, shown in the next sample, is an example of a value being constrained—a *type constraint*. In this case, the function parameter **x** is constrained to be an **int**. It is possible to constrain any identifier, not just function parameters, to be of a certain type, though it is more typical to need to constrain parameters. The list **stringList** here shows how to constrain an identifier that is not a function parameter.

```
let doNothingToAnInt (x: int) = x
let intList = [1; 2; 3]

let (stringList: list<string>) = ["one"; "two"; "three"]
```

```
val doNothingToAnInt _int : int -> int
val intList : int list
val stringList : string list
```

The syntax for constraining a value to be of a certain type is straightforward. Within parentheses, the identifier name is followed by a colon (**:**), followed by the type name. This is also sometimes called a *type annotation*.

The **intList** value is a list of integers, and the identifier's type is **int list**. This indicates that the compiler has recognized that the list contains only integers, and in this case the type of its items is not undetermined, but is **int**. Any attempt to add anything other than values of type **int** to the list will result in a compile error.

The identifier **stringList** has a type annotation. Although this is unnecessary since the compiler can resolve the type from the value, it is used to show an alternative syntax for working

with undetermined types. You can place the type between angle brackets after the type it is associated with instead of just writing it before the type name. Note that even though the type of **stringList** is constrained to be **list<string>** (a list of strings), the compiler still reports its type as **string list** when displaying the type, and they mean exactly the same thing. This syntax is supported to make F# types with a type parameter look like generic types from other .NET libraries.

Constraining values is not usually necessary when writing pure F#, though it can occasionally be useful. It's most useful when using .NET libraries written in languages other than F# and for interoperation with unmanaged libraries. In both cases, the compiler has less type information, so it is often necessary to give it enough information to disambiguate values.

Defining Types

The type system in F# provides a number of features for defining custom types. All of F#'s type definitions fall into two categories:

- *Tuples* or *records*, which are a set of types composed to form a composite type (similar to structs in C or classes in C#).
- *Sum* types, sometimes referred to as *union* types.

Tuple and Record Types

Tuples are a way of quickly and conveniently composing values into a group of values. Values are separated by commas and can then be referred to by one identifier, as shown in the first line of the next example. You can then retrieve the values by doing the reverse, as shown in the second and third lines, where identifiers separated by commas appear on the left side of the equal sign, with each identifier receiving a single value from the tuple. If you want to ignore a value in the tuple, you can use **_** to tell the compiler you are not interested in the value, as in the second and third lines.

```
let pair = true, false
let b1, _ = pair
let _, b2 = pair
```

Tuples are different from most user-defined types in F# because you do not need to explicitly declare them using the **type** keyword. To define a type, you use the **type** keyword, followed by the type name, an equal sign, and then the type you are defining. In its simplest form, you can use this to give an alias to any existing type, including tuples. Giving aliases to single types is not often useful, but giving aliases to tuples can be very useful, especially when you want to use a tuple as a type constraint. The next example shows how to give an alias to a single type and a tuple, and also how to use an alias as a type constraint.

```
type Name = string
type Fullname = string * string

let fullNameToSting (x: Fullname) =
    let first, second = x in
    first + " " + second
```

Record types are similar to tuples in that they compose multiple types into a single type. The difference is that in record types, each *field* is named. The next example illustrates the syntax for defining record types.

```
// Define an organization with unique fields.
type Organization1 = { boss: string; lackeys: string list }
// Create an instance of this organization.
let rainbow =
    { boss = "Jeffrey";
      lackeys = ["Zippy"; "George"; "Bungle"] }

// Define two organizations with overlapping fields.
type Organization2 = { chief: string; underlings: string list }
type Organization3 = { chief: string; indians: string list }

// Create an instance of Organization2.
let (thePlayers: Organization2) =
    { chief = "Peter Quince";
      underlings = ["Francis Flute"; "Robin Starveling";
                    "Tom Snout"; "Snug"; "Nick Bottom"] }

// Create an instance of Organization3.
let (wayneManor: Organization3) =
    { chief = "Batman";
      indians = ["Robin"; "Alfred"] }
```

You place field definitions between braces and separate them with semicolons. A field definition is composed of the field name followed by a colon and the field's type. The type definition **`Organization1`** is a record type where the field names are unique. This means you can use a simple syntax to create an instance of this type where there is no need to mention the type name when it is created. To create a record, you place the field names followed by equal signs and the field values between braces (**`{}`**), as shown in the **`Rainbow`** identifier.

F# does not force field names to be unique, so sometimes the compiler cannot infer the type of a field from the field names alone. In such a case, the compiler cannot infer the type of the record. To create records with nonunique fields, the compiler needs to statically know the type of the record being created. If the compiler cannot infer the type of the record, you need to use a

type annotation as described in the Type Inference section. Using a type annotation is illustrated by the types **Organization2** and **Organization3**, and their instances **thePlayers** and **wayneManor**. You can see the type of the identifier given explicitly just after its name.

Accessing the fields in a record is fairly straightforward. You simply use the syntax **record** identifier name, followed by a dot, followed by field name. The following example illustrates this, showing how to access the **chief** field of the **Organization** record.

```
// Define an organization type.
type Organization = { chief: string; indians: string list }

// Create an instance of this type.
let wayneManor =
    { chief = "Batman";
      indians = ["Robin"; "Alfred"] }

// Access a field from this type.
printfn "wayneManor.chief = %s" wayneManor.chief
```

Records are immutable by default. To an imperative programmer, this may sound like records are not very useful, since there will inevitably be situations where you need to change a value in a field. For this purpose, F# provides a simple syntax for creating a copy of a record with updated fields. To create a copy of a record, place the name of that record between braces followed by the keyword **with**, and then followed by a list of fields to be changed with their updated values. The advantage of this is that you don't need to retype the list of fields that have not changed. The following example demonstrates this approach. It creates an initial version of **wayneManor** and then creates **wayneManor'**, in which **"Robin"** has been removed.

```
// Define an organization type.
type Organization = { chief: string; indians: string list }

// Create an instance of this type.
let wayneManor =
    { chief = "Batman";
      indians = ["Robin"; "Alfred"] }

// Create a modified instance of this type.
let wayneManor' =
    { wayneManor with indians = [ "Alfred" ] }

// Print out the two organizations.
printfn "wayneManor = %A" wayneManor
printfn "wayneManor' = %A" wayneManor'
```

The results of this example, when compiled and executed, are as follows:

```
wayneManor = {chief = "Batman";
 indians = ["Robin"; "Alfred"];}
wayneManor' = {chief = "Batman";
 indians = ["Alfred"];}
```

Another way to access the fields in a record is using pattern matching; that is, you can use pattern matching to match fields within the record type. As you would expect, the syntax for examining a record using pattern matching is similar to the syntax used to construct it. You can compare a field to a constant with *field* **=** *constant.* You can assign the values of fields with identifiers with *field* **=** *identifier.* You can ignore a field with *field* **=** **_**. The **findDavid** function in the following example illustrates using pattern matching to access the fields in a record.

```
// Type representing a couple.
type Couple = { him : string ; her : string }

// List of couples.
let couples =
    [ { him = "Brad" ; her = "Angelina" };
      { him = "Becks" ; her = "Posh" };
      { him = "Chris" ; her = "Gwyneth" };
      { him = "Michael" ; her = "Catherine" } ]

// Function to find "David" from a list of couples.
let rec findDavid l =
    match l with
    | { him = x ; her = "Posh" } :: tail -> x
    | _ :: tail -> findDavid tail
    | [] -> failwith "Couldn't find David"

// Print the results.
printfn "%A" (findDavid couples)
```

The first rule in the **findDavid** function is the one that does the real work, checking the **her** field of the record to see whether it is **"Posh"**, David's wife. The **him** field is associated with the identifier **x** so it can be used in the second half of the rule.

The results of this example, when compiled and executed, are as follows:

```
Becks
```

It's important to note that you can use only literal values when you pattern match over records like this. So, if you wanted to generalize the function to allow you to change the person you are searching for, you would need to use a **when** guard in your pattern matching:

```
let rec findPartner soughtHer l =
    match l with
    | { him = x ; her = her } :: tail when her = soughtHer -> x
    | _ :: tail -> findPartner soughtHer tail
    | [] -> failwith "Couldn't find him"

// Print the results.
printfn "%A" (findPartner "Angelina" couples )
```

Field values can also be functions, which can occasionally be useful to provide object-like behavior as each record instance of the record could have a different implementation of the function.

Union or Sum Types

Union types, sometimes called *sum types* or *discriminated unions*, are a way of bringing together data that may have a different meaning or structure.

You define a union type using the **type** keyword, followed by the type name, followed by an equal sign—the same as with all type definitions. Next are the definitions of the different *constructors* separated by pipes. The first pipe is optional.

A constructor is composed of a name that must start with a capital letter, which is intended to avoid the common bug of getting constructor names mixed up with identifier names. The name can optionally be followed by the keyword **of** and then the types that make up that constructor. Multiple types that make up a constructor are separated by asterisks. The names of constructors within a type must be unique. If several union types are defined, then the names of their constructors can overlap; however, you should be careful when doing this, because it is possible that further type annotations are required when constructing and consuming union types.

The next example defines a type **Volume** whose values can have three different meanings: liter, US pint, or imperial pint. Although the structure of the data is the same and is represented by a float, the meanings are quite different. Mixing up the meaning of data in an algorithm is a common cause of bugs in programs, and the **Volume** type is, in part, an attempt to avoid this.

```
type Volume =
    | Liter of float
    | UsPint of float
    | ImperialPint of float
```

```
let vol1 = Liter 2.5
let vol2 = UsPint 2.5
let vol3 = ImperialPint (2.5)
```

The syntax for constructing a new instance of a union type is the constructor name followed by the values for the types, with multiple values separated by commas. Optionally, you can place the values in parentheses. You use the three different **`Volume`** constructors to construct three different identifiers: **`vol1`**, **`vol2`**, and **`vol3`**.

To deconstruct the values of union types into their basic parts, you always use pattern matching. When pattern matching over a union type, the constructors make up the first half of the pattern matching rules. You don't need a complete list of rules, but if the list is incomplete, there must be a default rule using either an identifier or a wildcard to match all remaining rules. The first part of a rule for a constructor consists of the constructor name followed by identifiers or wildcards to match the various values within it. The following **`convertVolumeToLiter`**, **`convertVolumeUsPint`**, and **`convertVolumeImperialPint`** functions demonstrate this syntax:

```
// Type representing volumes.
type Volume =
    | Liter of float
    | UsPint of float
    | ImperialPint of float

// Various kinds of volumes.
let vol1 = Liter 2.5
let vol2 = UsPint 2.5
let vol3 = ImperialPint 2.5

// Some functions to convert between volumes.
let convertVolumeToLiter x =
    match x with
    | Liter x -> x
    | UsPint x -> x * 0.473
    | ImperialPint x -> x * 0.568
let convertVolumeUsPint x =
    match x with
    | Liter x -> x * 2.113
    | UsPint x -> x
    | ImperialPint x -> x * 1.201
let convertVolumeImperialPint x =
    match x with
    | Liter x -> x * 1.760
    | UsPint x -> x * 0.833
```

```
    | ImperialPint x -> x

// A function to print a volume.
let printVolumes x =
    printfn "Volume in liters = %f,
in us pints = %f,
in imperial pints = %f"
        (convertVolumeToLiter x)
        (convertVolumeUsPint x)
        (convertVolumeImperialPint x)
// Print the results.
printVolumes vol1
printVolumes vol2
printVolumes vol3
```

An alternative solution to this problem is to use F#'s units of measure, which allow types to be applied to numeric values.

Type Definitions with Type Parameters

Both union and record types can be parameterized. Parameterizing a type means leaving one or more of the types within the type being defined to be determined later by the consumer of the types. This is a similar concept to the variable types discussed earlier in this chapter. When defining types, you must be a little more explicit about which types are variable.

To create a type parameter or parameters, place the types being parameterized in angle brackets after the type name, as follows:

```
type BinaryTree<'a> =
| BinaryNode of 'a BinaryTree * 'a BinaryTree
| BinaryValue of 'a

let tree1 =
    BinaryNode(
        BinaryNode ( BinaryValue 1, BinaryValue 2),
        BinaryNode ( BinaryValue 3, BinaryValue 4) )
```

Like variable types, the names of type parameters always start with a single quote (') followed by an alphanumeric name for the type. Typically, just a single letter is used. If multiple parameterized types are required, you separate them with commas. You can then use the type parameters throughout the type definition.

The syntax for creating and consuming an instance of a parameterized type does not change from that of creating and consuming a nonparameterized type. This is because the compiler will

automatically infer the type parameters of the parameterized type. You can see this in the following construction of **tree1**, and their consumption by the function **printBinaryTreeValues**:

```
// Definition of a binary tree.
type BinaryTree<'a> =
    | BinaryNode of 'a BinaryTree * 'a BinaryTree
    | BinaryValue of 'a

// Create an instance of a binary tree.
let tree1 =
    BinaryNode(
        BinaryNode ( BinaryValue 1, BinaryValue 2),
        BinaryNode ( BinaryValue 3, BinaryValue 4) )

// Function to print the binary tree.
let rec printBinaryTreeValues x =
    match x with
    | BinaryNode (node1, node2) ->
        printBinaryTreeValues node1
        printBinaryTreeValues node2
    | BinaryValue x ->
        printf "%A, " x

// Print the results.
printBinaryTreeValues tree1
```

The results of this example, when compiled and executed, are as follows:

```
1, 2, 3, 4,
```

You may have noticed that although I've discussed defining types, creating instances of them, and examining these instances, I haven't discussed updating them. It is not possible to update these kinds of types, because the idea of a value that changes over time goes against the idea of functional programming.

Summary

This has been a brief tour of how to create types in F#. You'll find that F#'s type system provides a flexible way to represent data in your programs.

Chapter 5 Object-Oriented Programming

Object-oriented programming is the third major programming paradigm. There has been a tendency to try and show the functional paradigm and the object-oriented paradigm as competing, but I believe them to be complementary techniques that work well together, which I will try to demonstrate in this chapter. At its heart, object-oriented programming has a few simple ideas, sometimes referred to as the tenets of object-oriented programming: encapsulation, polymorphism, and inheritance.

Possibly the most important tenet is *encapsulation*—the idea that the implementations and state should be *encapsulated*, or hidden behind well-defined boundaries. This makes the structure of a program easier to manage. In F#, you hide things by using signatures for modules and type definitions, as well as by simply defining them locally to an expression or class construction (you'll see examples of both in this chapter).

The second tenet, *polymorphism*, is the idea that you can implement abstract entities in multiple ways. You've met a number of simple abstract entities already, such as function types. A function type is abstract because you can implement a function with a specific type in many different ways. For example, you can implement the function type **`int -> int`** as a function that increments the given parameter, a function that decrements the parameter, or any one of millions of mathematical sequences. You can also build other abstract entities out of existing abstract components, such as the interface types defined in the .NET BCL. You can also model more sophisticated abstract entities using user-defined interface types. Interface types have the advantage that you can arrange them hierarchically; this is called *interface inheritance*. For example, the .NET BCL includes a hierarchical classification of collection types, available in the **`System.Collections`** and **`System.Collections.Generic`** namespaces.

In OOP, you can sometimes arrange implementation fragments hierarchically. This is called *implementation inheritance*, and it tends to be less important in F# programming because of the flexibility that functional programming provides for defining and sharing implementation fragments. However, it is significant for domains such as graphical user interface (GUI) programming.

While the tenets of object-oriented programming are important, object-oriented programming has also become synonymous with organizing your code around the values of the system *nouns* and then providing operations on those values as members, functions, or methods that operate on this value. This is often as simple as taking a function written in the style where the function is applied to a value (such as **`String.length s`**) and rewriting it using the dot notation (such as **`s.Length`**). This simple act can often make your code a good deal clearer. You'll see in this chapter how F# allows you to attach members to any of its types, not just its classes, enabling you to organize all your code in an object-oriented style if you wish.

F# provides a rich object-oriented programming model that allows you to create classes, interfaces, and objects that behave similarly to those created by C# and VB.NET. Perhaps more importantly, the classes you create in F# are indistinguishable from those that are created in

other languages when packaged in a library and viewed by a user of that library. However, object-oriented programming is more than simply defining objects, as you'll see when you start looking at how you can program in an object-oriented style using F# native types.

F# Types with Members

It is possible to add functions to both F#'s record and union types. You can call a function added to a record or union type using dot notation, just as you can a member of a class from a library not written in F#. It also proves useful when you want to expose types you define in F# to other .NET languages. Many programmers prefer to see function calls made on an instance value, and this technique provides a nice way of doing this for all F# types.

The syntax for defining an F# record or union type with members is the same as the syntax you learned in Chapter 4, except here it includes member definitions that always come at the end, after the **`with`** keyword. The definition of the members themselves starts with the keyword **`member`**, followed by an identifier that represents the parameter of the type the member is being attached to, followed by a dot, the function name, and then any other parameters the function takes. After this comes an equal sign followed by the function definition, which can be any F# expression.

The following example defines a record type, **`Point`**. It has two fields, **`Left`** and **`Top`**; and a member function, **`Swap`**. The **`Swap`** function is a simple function that creates a new point with the values of **`Left`** and **`Top`** swapped. Note how to use the **`x`** parameter, given before the function name **`Swap`**, within the function definition to access the record's other members:

```
// A point type.
type Point =
    { Top: int;
      Left: int }
    with
        // The swap member creates a new point
        // with the left/top coords reversed.
        member x.Swap() =
            { Top = x.Left;
              Left = x.Top }

// Create a new point.
let myPoint =
    { Top = 3;
      Left = 7 }

let main() =
    // Print the inital point.
    printfn "%A" myPoint
```

```
    // Create a new point with the coordinates swapped.
    let nextPoint = myPoint.Swap()
    // Print the new point.
    printfn "%A" nextPoint

// Start the app.
do main()
```

When you compile and execute this example, you get the following results:

```
{Top = 3;
 Left = 7;}
{Top = 7;
 Left = 3;}
```

You might have noticed the **x** parameter in the definition of the function **Swap**:

```
member x.Swap() =
    { Top = x.Left;
      Left = x.Top }
```

This is the parameter that represents the object on which the function is being called. Now look at the case where you call a function on a value:

```
let nextPoint = myPoint.Swap()
```

The value you call the function on is passed to the function as an argument. This is logical when you consider that the function needs to be able to access the fields and methods of the value on which you call it. Some OO languages use a specific keyword for this, such as **this** or **Me**, but F# lets you choose the name of this parameter by specifying a name for it after the keyword member—**x**, in this case.

Union types can have member functions, too. You define them in the same way that you define record types. The next example shows a union type, **DrinkAmount**, which has a function added to it:

```
// A type representing the amount of a specific drink.
type DrinkAmount =
    | Coffee of int
    | Tea of int
    | Water of int
```

```
    with
        // Get a string representation of the value.
        override x.ToString() =
            match x with
            | Coffee x -> Printf.sprintf "Coffee: %i" x
            | Tea x -> Printf.sprintf "Tea: %i" x
            | Water x -> Printf.sprintf "Water: %i" x

// Create a new instance of DrinkAmount.
let t = Tea 2

// Print out the string.
printfn "%s" (t.ToString())
```

When you compile and execute this code, you get the following results:

```
Tea: 2
```

Note how this uses the keyword **`override`** in place of the keyword **`member`**. This has the effect of replacing, or *overriding*, an existing function of the base type. This is not a very common practice with function members associated with F# types because only four methods are available to be overridden: **`ToString`**, **`Equals`**, **`GetHashCode`**, and **`Finalize`**. Every .NET type inherits these from **`System.Object`**. Because of the way some of these methods interact with the CLR, the only one I recommend overriding is **`ToString`**. Only four methods are available for overriding because record and union types can't act as base or derived classes, so you cannot inherit methods to override (except from **`System.Object`**).

Defining Classes

You have already seen quite a few examples of using classes defined in the .NET BCL library; next, you'll learn how to define your own classes. In object-oriented programming, a class should model some concept used within the program or library you are creating. For example, the **`String`** class models a collection of characters, and the **`Process`** class models an operating-system process.

A class is a type, so a class definition starts with the **`type`** keyword, followed by the name of the class and the parameters of the class' constructor between parentheses. Next comes an equal sign, followed by a class' member definitions. The most basic member of a class is called a *method*, which is a function that has access to the parameters of the class.

The next example shows a class that represents a user. The user class' constructor takes two parameters: the user's name and a hash of the user's password. Your class provides two

member methods: **Authenticate**, which checks whether the user's password is valid; and **LogonMessage**, which gets a user-specific logon message:

```
open System.Web.Security
// Give shorter name to password hashing method.
let hash = FormsAuthentication.HashPasswordForStoringInConfigFile

// A class that represents a user.
// Its constructor takes two parameters: the user's
// name and a hash of his or her password.
type User(name, passwordHash) =
    // Hashes the user's password and checks it against
    // the known hash.
    member x.Authenticate(password) =
        let hashResult = hash (password, "sha1")
        passwordHash = hashResult

    // Gets the user's logon message.
    member x.LogonMessage() =
        Printf.sprintf "Hello, %s" name

// Create a new instance of our user class.
let user = User("Robert", "AF73C586F66FDC99ABF1EADB2B71C5E46C80C24A")

let main() =
    // Authenticate user and print appropriate message.
    if user.Authenticate("badpass") then
        printfn "%s" (user.LogonMessage())
    else
        printfn "Logon failed"

do main()
```

The second half of the example demonstrates how to use the class. It behaves exactly like other classes you've seen from the .NET BCL. You can create a new instance of **User** using the **new** keyword, and then call its member methods.

It's often useful to define values that are internal to your classes. Perhaps you need to pre-calculate a value that you share between several member methods, or maybe you need to retrieve some data for the object from an external data source. To enable this, objects can have **let** bindings that are internal to the object, but shared between all members of the object. You place the **let** bindings at the beginning of the class definition, after the equal sign, but before the first member definition. These **let** bindings form an implicit construct that executes when the object is constructed; if the **let** bindings have any side effects, these too will occur when the

object is constructed. If you need to call a function that has the **unit** type, such as when logging the object's construction, you must prefix the function call with the **do** keyword.

The next example demonstrates private **let** bindings by taking your **User** class and modifying it slightly. Now the class constructor takes a **firstName** and **lastName**, which you use in the **let** binding to calculate the user's **fullName**. To see what happens when you call a function with a side effect, you can print the user's **fullName** to the console:

```
// A class that represents a user.
// Its constructor takes three parameters: the user's
// first name, last name, and a hash of his or her password.
type User(firstName, lastName, passwordHash) =
    // Calculate the user's full name and store for later use
    let fullName = Printf.sprintf "%s %s" firstName lastName
    // Print the user's full name as object is being constructed.
    do printfn "User: %s" fullName

    // Hashes the user's password and checks it against
    // the known hash.
    member x.Authenticate(password) =
        let hashResult = hash (password, "sha1")
        passwordHash = hashResult

    // Retrieves the user's full name.
    member x.GetFullname() = fullName
```

Notice how the members also have access to the class' **let** bindings, and how the member **GetFullName** returns the pre-calculated **fullName** value.

It's common to need to be able to change values within classes. For example, you might need to provide a **ChangePassword** method to reset the user's password in the **User** class. F# gives you two approaches to accomplish this. You can make the object immutable—in this case, you copy the object's parameters, changing the appropriate value as you go. This method is generally considered to fit better with functional-style programming, but it can be a little inconvenient if the object has a lot of parameters or is expensive to create. For example, doing this might be computationally expensive, or it might require lots of I/O to construct it. The following example illustrates this approach. Notice how in the **ChangePassword** method you call the **hash** function on the **password** parameter, passing this to the **User** object's constructor along with the user's name:

```
// A class that represents a user.
// Its constructor takes two parameters: the user's
// name and a hash of his or her password.
type User(name, passwordHash) =
```

```
    // Hashes the user's password and checks it against
    // the known hash.
    member x.Authenticate(password) =
        let hashResult = hash (password, "sha1")
        passwordHash = hashResult

    // Gets the user's logon message.
    member x.LogonMessage() =
        Printf.sprintf "Hello, %s" name

    // Creates a copy of the user with the password changed.
    member x.ChangePassword(password) =
        new User(name, hash password)
```

The alternative to an immutable object is to make the value you want to change mutable. You do this by binding it to a mutable **`let`** binding. You can see this in the next example, where you bind the class's parameter **`passwordHash`** to a mutable **`let`** binding of the same name:

```
// A class that represents a user.
// Its constructor takes two parameters: the user's
// name and a hash of his or her password.
type User(name, passwordHash) =
    // Store the password hash in a mutable let
    // binding so it can be changed later.
    let mutable passwordHash = passwordHash

    // Hashes the user's password and checks it against
    // the known hash.
    member x.Authenticate(password) =
        let hashResult = hash (password, "sha1")
        passwordHash = hashResult

    // Gets the user's logon message.
    member x.LogonMessage() =
        Printf.sprintf "Hello, %s" name

    // Changes the user's password.
    member x.ChangePassword(password) =
        passwordHash <- hash password
```

This means you are free to update the **`passwordHash`** to a **`let`** binding, as you do in the **`ChangePassword`** method.

Defining Interfaces

Interfaces can contain only abstract methods and properties, or members that you declare using the keyword **abstract**. Interfaces define a *contract* for all classes that implement them, exposing those components that clients can use while insulating clients from their actual implementation. A class can inherit from only one base class, but it can implement any number of interfaces. Because any class implementing an interface can be treated as being of the interface type, interfaces provide similar benefits to multiple-class inheritance while avoiding the complexity of that approach.

You define interfaces by defining a type that has no constructor and where all the members are abstract. The following example defines an interface that declares two methods: **Authenticate** and **LogonMessage**. Notice how the interface name starts with a capital **I**. This is a naming convention that is strictly followed throughout the .NET BCL and you should follow it in your code too. It will help other programmers distinguish between classes and interfaces when reading your code:

```
// An interface "IUser".
type IUser =
    // Hashes the user's password and checks it against
    // the known hash.
    abstract Authenticate: evidence: string -> bool
    // Gets the user's logon message.
    abstract LogonMessage: unit -> string

let logon (user: IUser) =
    // Authenticate user and print appropriate message.
    if user.Authenticate("badpass") then
        printfn "%s" (user.LogonMessage())
    else
        printfn "Logon failed"
```

The second half of the example illustrates the advantages of interfaces. You can define a function that uses the interface without knowing the implementation details. You define a **logon** function that takes an **IUser** parameter and uses it to perform a logon. This function will then work with any implementations of **IUser**. This is extremely useful in many situations; for example, it enables you to write one set of client code that you can reuse with several different implementations of the interface.

Implementing Interfaces

To implement an interface, use the keyword **interface**, followed by the interface name, the keyword **with**, and then the code to implement the interface members. You prefix member definitions with the keyword **member**, but they are otherwise the same as the definition of any

method or property. You can implement interfaces by either classes or structs; you can learn how to create classes in some detail in the following sections.

The next example defines, implements, and uses an interface. The interface is the same **IUser** interface you implemented in the previous section; here you implement it in a class called **User**:

```
open System.Web.Security
// Give shorter name to password hashing method.
let hash = FormsAuthentication.HashPasswordForStoringInConfigFile
// An interface "IUser".
type IUser =
    // Hashes the user's password and checks it against
    // the known hash.
    abstract Authenticate: evidence: string -> bool
    // Gets the user's logon message.
    abstract LogonMessage: unit -> string

// A class that represents a user.
// Its constructor takes two parameters: the user's
// name and a hash of his or her password
type User(name, passwordHash) =
    interface IUser with
        // Authenticate implementation.
        member x.Authenticate(password) =
            let hashResult = hash (password, "sha1")
            passwordHash = hashResult

        // LogonMessage implementation.
        member x.LogonMessage() =
            Printf.sprintf "Hello, %s" name

// Create a new instance of the user.
let user = User("Robert", "AF73C586F66FDC99ABF1EADB2B71C5E46C80C24A")
// Cast to the IUser interface.
let iuser = user :> IUser
// Get the logon message.
let logonMessage = iuser.LogonMessage()

let logon (iuser: IUser) =
    // Authenticate the user and print the appropriate message.
    if iuser.Authenticate("badpass") then
        printfn "%s" logonMessage
    else
        printfn "Logon failed"
```

```
do logon user
```

Notice how in the middle of the example you see *casting* for the first time; you can find a more detailed explanation of casting at the end of the chapter in the Casting section. But for now here's a quick summary of what happens: the identifier **`user`** is cast to the interface **`IUser`** via the downcast operator, **`:>`**:

```
// Create a new instance of the user.
let user = User("Robert", "AF73C586F66FDC99ABF1EADB2B71C5E46C80C24A")
// Cast to the IUser interface.
let iuser = user :> IUser
```

This is necessary because interfaces are explicitly implemented in F#. Before you can use the method **`GetLogonMessage`**, you must have an identifier that is of the type **`IUser`** and not just of a class that implements **`IUser`**. Toward the end of the example, you will work around this in a different way. The function **`logon`** takes a parameter of the **`IUser`** type:

```
let logon (iuser: IUser) =
```

When you call **`logon`** with a class that implements **`IUser`**, the class is implicitly downcast to this type.

Casting

Casting is a way of explicitly altering the static type of a value by either throwing information away, which is known as *upcasting*; or rediscovering it, which is known as *downcasting*. In F#, upcasts and downcasts have their own operators. The type hierarchy starts with **`obj`** (or **`System.Object`**) at the top, with all its descendants below it. An upcast moves a type up the hierarchy, while a downcast moves a type down the hierarchy.

Upcasts change a value's static type to one of its ancestor types. This is a safe operation. The compiler can always tell whether an upcast will work because it always knows all the ancestors of a type, so it's able to use static analysis to determine whether an upcast will be successful. An upcast is represented by a colon, followed by the greater-than sign (**`:>`**). The following code shows you how to use an upcast to convert a **`string`** to an **`obj`**:

```
let myObject = ("This is a string" :> obj)
```

Generally, you must use upcasts when defining collections that contain disparate types. If you don't use an upcast, the compiler will infer that the collection has the type of the first element and give a compile error if elements of other types are placed in the collection. The next example demonstrates how to create an array of controls, a common task when working with

Windows Forms. Notice that you upcast all the individual controls to their common base class, **`Control`**:

```
open System.Windows.Forms

let myControls =
    [| (new Button() :> Control);
       (new TextBox() :> Control);
       (new Label() :> Control) |]
```

An upcast also has the effect of automatically boxing any value type. Value types are held in memory on the program stack, rather than on the managed heap. Boxing means that the value is pushed onto the managed heap, so it can be passed around by reference. The following example demonstrates how to box a value:

```
let boxedInt = (1 :> obj)
```

A downcast changes a value's static type to one of its descendant types; thus, it recovers information hidden by an upcast. Downcasting is dangerous because the compiler doesn't have any way to determine statically whether an instance of a type is compatible with one of its derived types. This means you can get it wrong, and this will cause an invalid cast exception (**`System.InvalidCastException`**) to be issued at run time. Due to the inherent danger of downcasting, many developers prefer to replace it with pattern matching over .NET types. Nevertheless, a downcast can be useful in some places, so a downcast operator—composed of a colon, question mark, and greater-than sign (**`:?>`**)—is available. The next example shows you how to use downcasting:

```
open System.Windows.Forms

let moreControls =
    [| (new Button() :> Control);
       (new TextBox() :> Control) |]

let control =
    let temp = moreControls.[0]
    temp.Text <- "Click Me!"
    temp

let button =
    let temp = (control :?> Button)
    temp.DoubleClick.Add(fun e -> MessageBox.Show("Hello") |> ignore)
    temp
```

This example creates an array of two Windows control objects, upcasting them to their base class, **`Control`**. Next, it binds the first control to the **`control`** identifier; downcasts this to its specific type, **`Button`**; and adds a handler to its **`DoubleClick`** event—an event not available on the **`Control`** class.

Summary

You've now seen how to use two of the three major programming paradigms in F# and how flexible F# is for coding in any mix of styles.

Chapter 6 Simulations and Graphics

I like to break the art of putting pixels on the screen into two parts:

- Low-level graphics programming, in which the programmer is responsible for the algorithms that control the geometry of what appears on the screen. An example of this might be creating a game, where the programmer is directly responsible for drawing the various elements that appear on the screen.

- High-level user interface creation, in which the programmer is responsible for composing together common elements, such as labels and text boxes, to quickly create practical user interfaces.

F# can be used to do both of these parts. In this chapter we'll look at how to create graphics in F#, and in the next chapter we'll look at how to put together forms and other higher-level user interfaces.

I've paired the creation of graphics in F# with the creation of simulations. This is because it's likely that the graphics you'll want to show will be the result of some kind of simulation. In this case, to keep the examples simple, I've chosen to create a simulation of a bouncing ball. There are many other simulations that might interest you, from Conway's game of life to many kinds of fractals. These could be implemented in a way similar to the bouncing ball simulation discussed in this chapter. Even the implementation of games in F# could borrow heavily from the techniques described in this chapter.

The bouncing ball simulation itself will be a short F# module that just generates the raw data of the simulation. We'll then show how this can be rendered in both WPF and Silverlight.

By creating the application in this way we have good separation of concerns. The simulation is responsible for the mathematical part of the application—although the math is very simple in this case—and the graphics layer is just responsible for rendering the data produced by the simulation.

The Bouncing Ball Simulation

The bouncing ball simulation we're going to look at is probably the simplest simulation you could imagine. Our aim is simply to model a ball that bounces infinitely between an area defined by four walls. To start our ball simulation, we need to choose a way of representing the ball. To do this, we must decide what information about the ball is important for the simulation. In this case we care about two things: the position of the ball and its velocity. As this is a 2-D simulation, these values can be represented by four numbers in X and Y coordinates: two to represent the position, and two to represent the ball's velocity. We could include many other details about the ball, for example a friction coefficient to help calculate how much drag the ball experiences, or details about the ball's spin direction and speed to help determine how the ball would move

when it hits an object, but our simulation is simple enough that these details just aren't relevant, so we don't include them. The ball can be represented by a simple F# record:

```
/// Represents a ball by its position and its velocity.
type Ball =
    { X: float
      Y: float
      XVel: float
      YVel: float }
```

It's often useful to provide a static method that constructs a new instance of the object. This provides a convenient shorthand for creating a new instance of an object. In this case this would look like:

```
        /// Convenient function to create a new instance of a ball.
        static member New(x: float, y: float, xVel: float, yVel: float) =
            { X = x
              Y = y
              XVel = xVel
              YVel = yVel }
```

Now that we have a representation of our ball, it's time to think about how the ball will move. We'll implement this as a function that creates a new updated version of the ball: the X and Y coordinates will be updated according the ball's direction of travel, and the velocity values will be updated if the ball hits a wall. To do this we need one additional set of data: the dimensions of the area the ball is travelling around. These will be passed as parameters to the function that calculates the ball's movement.

The calculation we need to make is very simple. We update the ball's position, and if the ball is outside of the bounds we reverse the direction of travel and again update the position. Otherwise, we just move the ball to its new position. I have implemented this algorithm as a member method of the **`Ball`** type and called the **`Move`** method. The **`Move`** method accepts four parameters: **`xMin`**, **`yMin`**, **`xMax`**, and **`yMax`**. These define the area in which the ball is moving. It's convenient to implement it as a member function with information about the environment passed as individual numbers since the simulation is simple. A more complex simulation would probably change in two ways. First, an environment type which grouped the information about the environment in which the ball was moving would be passed to the functions instead of individual numbers. Secondly, it might be necessary to have a centralized function in a module for coordinating the different elements of the simulation. As the simulation is simple, we'll stick to the convenience of a member and individual parameters:

```
        /// Creates a new ball that has moved one step by its velocity within
        /// the given bounds. If the bounds are crossed, the ball velocity
```

```
        /// is inversed.
        member ball.Move(xMin: float, yMin: float, xMax: float, yMax: float)
=
            // Local helper to implement the logic of the movement.
            let updatePositionAndVelocity pos vel min max =
                let pos' = pos + vel
                if min < pos' && pos' < max then
                    pos', vel
                else
                    pos - vel, -vel
            // Get the new position and velocity.
            let newX, newXVel =
                updatePositionAndVelocity ball.X ball.XVel xMin xMax
            let newY, newYVel =
                updatePositionAndVelocity ball.Y ball.YVel yMin yMax

            // Create the next ball.
            Ball.New(newX, newY,newXVel, newYVel)
```

Notice how there is really only one calculation to make, but we need to make it twice, for both the X and Y coordinates. To achieve this without repetition, we define a local helper function **`updatePositionAndVelocity`** which encapsulates the calculation of updating a coordinate. We can then call this function, pass it the relevant details for the X and Y coordinates, and the function will return the updated position and velocity. The final thing the function needs to do is create the new ball definition and return it.

This nicely illustrates the use of some simple but important features of F#, namely using local functions to avoid repetition and using tuples to return multiple values. In a language without local functions it may be tempting to just repeat the calculation of the position since it's fairly short and simple. However, any repetition can lead to the definitions diverging over time as the code is maintained, and this is likely to result in bugs. The ability to return multiple values via tuples also helps in this process. If we weren't able to do this, the alternative would be to directly update the variables from within our calculations. This would make the code less flexible as it would be coupled to the variable definitions that it was updating.

Now that we have a model which defines how the ball will move, we need to test the model to ensure we have the behavior we expect.

Testing the Model

One of the advantages of this model-based approach is that it makes testing very simple. The model consumes and produces data, so it's very easy to create the required inputs and verify that the model outputs the values we expect. Using F# provides the added convenience of using F# Interactive to create test scripts that we can execute with a single key stroke.

There are four cases we would like to test:

- The ball does not hit the wall in the X or the Y direction so no bounce occurs.
- The ball hits the wall in the X direction but not the Y direction so a bounce occurs in the X direction.
- The ball does not hit the wall in the X direction but does in the Y direction so a bounce occurs in the Y direction.
- The ball hits the wall in both the X and Y directions so a bounce in both directions occurs.

All tests will share the same structure; only the data and assertions will change, so we can write one generic test and use it to test all four scenarios. But first, to start off our testing we need to add a new F# script and add a couple of commands to load the code to be tested:

```
#load "BallModel.fs"
open FsSuccinctly.BouncingBall
```

The first line loads the file containing the ball model into F# Interactive, and the next opens the namespace defined in the **`BallModel.fs`** file to give easy access to the types and functions contained within it. Now that we have access to our types and functions that need testing, we can write this generic test function:

```
let genericBallTest name x y shouldBounceX shouldBounceY =
    // Create a ball.
    let ball = Ball.New(x, y, 1., 1.)
    // Move the ball.
    let ball' = ball.Move(0., 0., 100., 100.)
    // Verify we have the movement we expect.
    let xOp = if shouldBounceX then (-) else (+)
    let yOp = if shouldBounceY then (-) else (+)
    if xOp ball.X ball.XVel <> ball'.X then failwith "X value not updated
correctly"
    if yOp ball.Y ball.YVel <> ball'.Y then failwith "Y value not updated
correctly"
    // Notify the user that the test has passed.
    printfn "passed - %s" name
```

This function takes five parameters: **`name`**, which defines the name of the test and allows us to write a meaningful message when the test passes; **`x`** and **`y`**, which define the position of the ball for the test; and finally **`shouldBounceX`** and **`shouldBounceY`**, which define whether we expect the ball to bounce and are used during the test assertions. After that, the structure of the test is very simple. First we create an instance of our ball object to be tested, and then we call our

move function to create the updated ball. Next, we verify that the ball has moved correctly, and finally we print a success message if the ball has moved as expected. The only complicated bit of the test is verifying if the ball has moved correctly. To do this, we test the **shouldBounceX** and **shouldBounceY** parameters to see whether the ball's velocity should be added or subtracted, and then we perform the addition or subtraction operation ourselves and test it against the actual result. If the result is not as we expect, we call the **failwith** function—a nice shorthand way of creating an exception with an appropriate error message—otherwise we continue. The calculation of whether the ball is in the correct position is easy to make as we have access to both the original ball, which is immutable and therefore unchanged, and the updated ball instance.

Now that we have the function for running our tests, it's easy to create the tests we need for each of the four cases we are interested in. We know that each test takes place on a 100-by-100 grid, so we just need to start the ball in the appropriate position. Here are the four functions that will test each of the cases:

```
let testNoneBounce() =
    genericBallTest "testNoneBounce" 10. 10. false false

let testBounceX() =
    genericBallTest "testBounceX" 99. 10. true false

let testBounceY() =
    genericBallTest "testBounceY" 10. 99. false true

let testBounceBoth() =
    genericBallTest "testBounceBoth" 99. 99. true true
```

To execute these tests, we need to make a call to each function from the top level:

```
testNoneBounce()
testBounceX()
testBounceY()
testBounceBoth()
```

When these tests are executed interactively, the results we would expect to see are:

```
passed - testNoneBounce
passed - testBounceX
passed - testBounceY
passed - testBounceBoth
```

Testing in this basic way has a few disadvantages when compared to using a unit testing framework, such as NUnit. The two main disadvantages are the need to explicitly call the function you want to test and that all tests will stop at the first error. However the main advantage of this testing method is that there's no need to take a dependency on an external framework. Plus, the two approaches are not incompatible—I often find I start off testing a project just using F# Interactive directly and then migrate my tests to a unit testing framework as the project matures.

Drawing the Simulation's Results

Now that we're sure our simulation behaves as expected, it's time to draw the results on the screen. We're going to do this using WPF, and then using Silverlight. Although the controls available in WPF and Silverlight are similar, the low-level APIs differ significantly, so this will be an interesting exercise in porting the simulation between GUI libraries.

WPF has a nice set of methods for drawing shapes. To get access to these methods, the programmer simply needs to create a type that derives from **`System.Windows.FrameworkElement`** and override the **`OnRender`** method which is passed a **`DrawingContext`** parameter. We'll create a type, **`BallRender`**, which will be responsible for running the simulation and rendering the results. This is how we'll declare the type:

```
type BallRender() as br =
    inherit FrameworkElement()
```

To render the results of our simulation we need to scale them. We have chosen to simulate the ball bouncing on a 100-by-100 grid with the ball having a width of just one point. If we tried to render this without scaling, even on today's lowest resolution screens it would appear tiny. This is exactly the sort of geometric calculation that the UI should be taking care of. The simulation should be free to use values and representations that are convenient for making the simulation. It is the GUI's responsibility to translate this into something that would be appealing to the end user. To make these calculations we'll need some constants relating to the layout of the control, so we'll declare these next in the class' implicit constructor directly after the type definition:

```
    // Ball size and limits.
    let size = 5.0
    let xMax = 100.0
    let yMax = 100.0

    // Offset to give a nice board.
    let offset = 10.0
```

We'll also need some pen and brush objects to define how our lines and filled areas will look:

```
// Pen and brush objects.
let brush = new SolidColorBrush(Colors.Black)
let pen = new Pen(brush,1.0)
```

To run the simulation we'll need two things: a reference to the current ball object, whose position will be updated over time, and a timer that will be responsible for executing the simulation and updating the ball's position. The current ball will be held in a reference cell. A reference cell is a mutable type built into F# to support values that can change over time. A reference cell is created by the **`ref`** function:

```
// A reference cell that holds the current ball instance.
let ball = ref (Ball.New(50., 80., 0.75, 1.25))
```

For the timer we'll use WPF's **`DispatcherTimer`**. This is a timer whose **`Tick`** event executes on the GUI thread so there is no need to worry about marshaling the data from the simulation back to the GUI thread. This works well for our simple simulation where the calculation will execute so quickly that executing the **`Tick`** events on a background thread would actually harm performance as the overhead of thread synchronization would be more costly than the cost of the simulation itself. This will be the case for many of the simulations you create; however, as the complexity and execution time of a simulation grows, it can become necessary to run it on a background thread to keep the GUI responsive. When a simulation becomes so costly that running it requires a background thread, it's probably worth introducing a new abstraction—usually a new class—to run it. However, for many situations, a simple time will be fine and this can be declared as a class member as shown in the following sample:

```
// Timer for updating the ball's position.
let timer = new DispatcherTimer(Interval = new TimeSpan(0,0,0,0,50))
```

To create a smooth animation you'll need at least 12 frames per second, so the ball's position must be updated at least 12 times every second. Initialize the timer to run every 50 milliseconds. This should give us 20 frames per second, so we should always have nice, smooth animation.

The final thing we need to do in the constructor is adjust the control's size to correspond to the size of the area we're drawing and initialize the timer's **`Tick`** event. To do this, create an inner function **`init()`** and call it at the end of the constructor:

```
// Helper function to do some initialization.
let init() =
    // Set the control's width and height.
    br.Width <- (xMax * size) + (offset * 2.0)
    br.Height <- (yMax * size) + (offset * 2.0)
```

```
        // Set up the timer.
        timer.Tick.Add(fun _ ->
            // Move the current ball to its next position.
            ball := ball.Value.Move(0.,0., xMax,yMax)
            // Invalidate the control to force a redraw.
            br.InvalidateVisual())
        timer.Start()

    do init()
```

Let's take a close look at the initialization of the timer's **`Tick`** event. This is initialization by adding an anonymous lambda function to the event. Within this function, we first move the ball to its new position by calling **`Move`** on the current ball, and then store the resulting ball in the **`ball`** reference cell. Second, we call the control's **`InvalidateVisual()`** method—this will cause the control to be redrawn so that the user can see the ball's new position.

To allow us to draw on the control, we need to override its **`OnRender`** method. This will give us access to a **`DrawingContext`** parameter which has a nice set of methods for drawing primitives such as lines, rectangles, and ellipses. This makes drawing the ball very straightforward. It's just a matter of calculating its current position by multiplying the output of the simulation by the appropriate scale factor, and then adding the offset for the border. We can then draw the ball by calling the **`DrawEllipse`** method. We also draw a border to make it easier for the user to see the area in which the ball is moving:

```
    /// The function that takes care of actually drawing the ball.
    override x.OnRender(dc: DrawingContext) =
        // Calculate the ball's position on the canvas.
        let x = (ball.Value.X * size) + offset
        let y = (ball.Value.Y * size) + offset

        // Draw the ball and an outline rectangle.
        dc.DrawEllipse(brush, pen, new Point(x, y), size, size)
        dc.DrawRectangle(null, pen, new Rect(offset, offset, size * xMax,
size * xMax))
```

The code that is required to see the control is now complete. All that remains is a little WPF plumbing to create a window that will host the control and to create an event loop that will show the window:

```
module Main =

    // Create an instance of the new control.
```

```
let br = new BallRender()
// Create a window to hold the control.
let win = new Window(Title = "Bouncing Ball",
                     Content = br,
                     Width = br.Width + 20.,
                     Height = br.Height + 40.)

// Start the event loop and show the control.
let app = new Application()
[<STAThread>]
do app.Run win |> ignore
```

Here we see that the first **let** binding creates a new instance of our ball rendering control, and the next creates a window that will host the control, setting a few properties of the window as it is created. Finally, we create a new instance of the WPF application class and use this to start a WPF event loop by calling its run method. The most important thing to note about this is that we attach a **STAThread** attribute to the call to the run method to ensure that the correct threading model is used for the event loop. The following image shows what the window will look like when the program is executed:

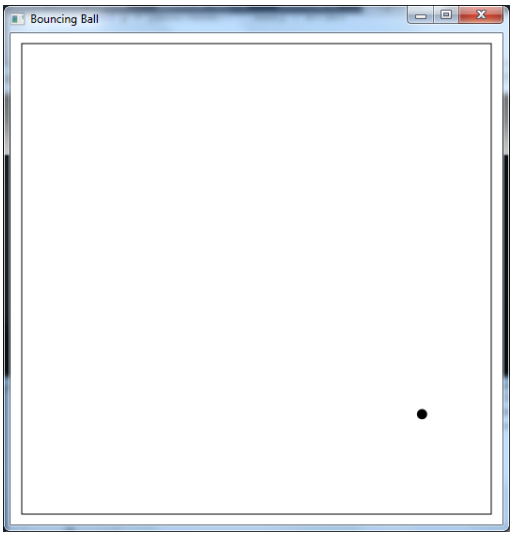

Figure 3: Bouncing Ball Simulation

Now we will take a quick look at how the control can be ported to Silverlight to take advantage of Silverlight's ability to be easily downloaded and executed in web browsers, allowing the simulations or games you create to be easily distributed to users.

The low-level interfaces to Silverlight are different than those of WPF; there are no APIs for drawing. If you want to create custom shapes, it's best to write the shapes to a writeable bitmap and then use a user control to show the bitmap.

Visual Studio 2012 comes with a template for creating an F# Silverlight application already installed. However, it's easier to create a Silverlight application using one of the templates available online. I based this sample on the F# Web Application (Silverlight) by Daniel Mohl, which can be downloaded from http://visualstudiogallery.msdn.microsoft.com/f0e9a557-3fd6-41d9-8518-c1735b382c73 or found by searching online in the **New Project** dialog as shown in the following figure:

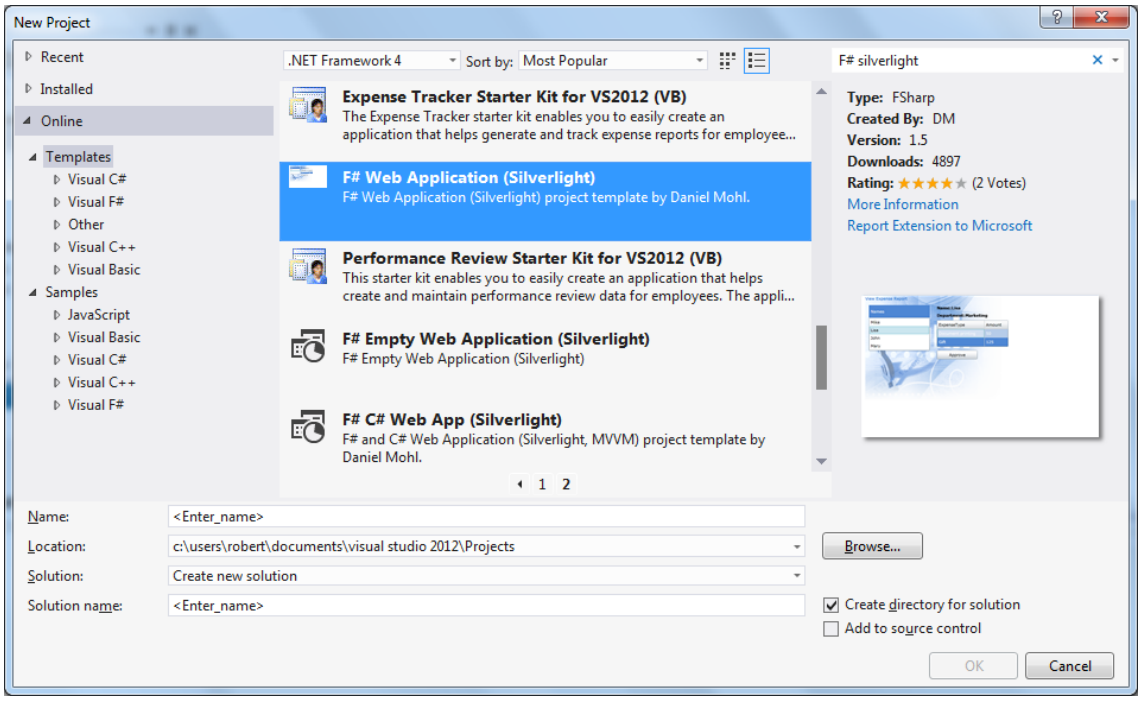

Figure 4: F# Web Application

This application comes with a number of predefined XAML pages that you will need to remove. Keep only the bare bones of the **AppLogic.fs** file that also comes with the template.

We'll structure the code similarly to what we did for the WPF version. We'll create a control that is responsible for rendering the bouncing ball, and then add the boilerplate code that is required to show this in Silverlight. The ball rendering control starts by declaring a new type that inherits from **`UserControl`**:

```
type BallRender() as br =
    inherit UserControl()
```

Then we'll add some constants that will control to how the overall control will appear:

```
    // Ball size and limits.
    let size = 5.0
    let xMax = 100.0
```

```
    let yMax = 100.0

    // Offset to give a nice board.
    let offset = 10.0

    // Calculate total width and height of the area.
    let xTotalWidth = (xMax * size) + (offset * 2.0) |> int
    let yTotalWidth = (yMax * size) + (offset * 2.0) |> int
```

Next we'll need to declare a writable bitmap on which we will draw our bouncing ball, and the bitmap will need to be wrapped in an image control so it can be shown:

```
    // The writable bitmap which will allow us to draw pixels onto it.
    let bitmap = new WriteableBitmap(xTotalWidth,
                                     yTotalWidth)

    // The image control that holds the bitmap.
    let image = new Image(Source = bitmap)
```

Unfortunately, the writable bitmap doesn't support any easy way of writing to it; it has only a flat array of integers that represent the colors of the bitmap. To make drawing on the bitmap easier, we need a couple of abstractions to allow us to set the pixels of the bitmaps more easily. This takes the form of two functions: a function to convert a Silverlight **`Color`** value to a **`System.Int32`**, and a function that can set a pixel's color by its x-coordinate and y-coordinate rather than its place in the flat array of integers the writable bitmap uses to represent its pixels.

```
    // Convert an RGB color to the int format used by the bitmap.
    let colorToInt (c: Color): int =
        [c.A, 24; c.R, 16; c.G, 8; c.B, 0]
        |> List.sumBy (fun (col, pos) -> (int col) <<< pos)

    // Function to set a pixel by its x-coordinate and y-coordinate to the
given color.
    let setPixel x y c =
        bitmap.Pixels.[y * xTotalWidth + x] <- colorToInt c
```

The **`colorToInt`** function takes a **`Color`** value and returns an **`int`**. The **`Color`** value has **`byte`** members that represent the alpha, red, green, and blue components of the color. To fit these values into a single integer, each component needs to be shifted by a number of bits—0 for blue, 8 for green, 16 for red, and 24 for alpha. The function does this by creating a list of tuples pairing the alpha, red, green, and blue components of the color with the number of bits by which they have to be shifted. We can then use the **`List.sumBy`** function, which takes a function to be applied to each item in the list and then sums the result of the function applied to each element of the list. So, the lambda function passed to **`List.sumBy`** performs the bit shift, while

List.sumBy accumulates the results into a single integer. The **setPixel** function takes the x- and y-coordinates of the pixel to be set and the color it is to be set to. The function then uses a simple formula to find the appropriate pixel to set, and then converts the given color to an integer and sets this pixel in the bitmap's **Pixel** array property.

Now that we can easily write pixels to our bitmap's surface, we can create a function to draw the ball. The simplest way to do this is to use a template of a list of strings that describe the ball's layout. Each character in the string represents a pixel of the ball. An asterisk (*****) means the pixel should be colored. Any other character means it should be left as it is. This may seem an odd thing to do, but it's a very convenient way to visually describe the ball in the code while obtaining a data structure that is very easy to work with. Once we have defined this ball template, it's just a matter of creating a function that takes a ball, enumerates the rows of the template, and then enumerates the characters in each row to see which pixels of the ball should be filled in.

```
// Template that describes what the ball should look like.
let ballTemplate =
      [ "     "
        " *** "
        "*****"
        " *** "
        "     " ]

// Draws the ball on the bitmap.
let drawBall ball c =
    // Calculate top corner of the ball.
    let xBall = (ball.X * size) + offset
    let yBall = (ball.Y * size) + offset
    // Iterate over the template, setting a
    // pixel if there is an asterisk.
    ballTemplate
    |> Seq.iteri(fun x row ->
        row |> Seq.iteri (fun y item ->
            if item = '*' then
                setPixel (int xBall + x)  (int yBall + y) c))
```

The **drawBall** function takes a ball and a color to draw it. The function enumerates the **ballTemplate** using F#'s **Seq.iteri** function, which enumerates any **IEnumerable<T>** collection and applies the given function to each member of the collection. The function that is applied to each member is also passed an integer representing the index of the current value (**iter** is short for iterate, and the **i** at the end of **iteri** is short for index). So we enumerate first the rows, then the individual columns, and draw a pixel each time we find an asterisk using the two indexes as an offset from the given ball's position. You might like to try experimenting with different ball layouts by changing the asterisks in the ball to see if you can find a more pleasing ball shape.

Now that we can draw a ball, we need to be able to draw a border around the area the ball will bounce in. We'll do this by creating a **drawSquare** function:

```
// Draws a square around the border of the area.
let drawSquare c =
    // Calculate where sides should end.
    let xSize = xMax * size + offset |> int
    let ySize = yMax * size + offset |> int
    // Convert the offset to an int.
    let offset = offset |> int
    // Draw the x sides.
    for x in offset .. xSize do
        setPixel x  offset c
        setPixel x  ySize c
    // Draw the y sides.
    for y in offset .. ySize do
        setPixel offset y c
        setPixel xSize y c
```

Drawing the square is quite easy; we just need to know on what pixel the top-left corner will start and where the bottom-right corner will end. The top-left corner will start at the **offset** for both the x and y directions, and the bottom-left corner will have coordinates of **offset + xMax * size, offset + xMax * size**. Once we have calculated these values, we can draw the horizontal lines of the square by iterating from the minimum x-coordinate to the maximum x-coordinate; setting the pixels at the current x, minimum y and current x, maximum y; and then doing the same thing in between the minimum y and maximum y to draw the vertical lines.

We're now capable of drawing both the ball and the border that will enclose it. Now we must use these two functions to show the ball's movement over time. To do this, we use the same technique we used for WPF: we create the ball in a reference cell to track the changes over time, and we use a time to update the ball and draw the results. The code to initialize the ball and timer is shown in the following sample.

```
// A reference cell that holds the current ball instance.
let ball = ref (Ball.New(50., 80., 0.75, 1.25))

// Timer for updating the ball's position.
let timer = new DispatcherTimer(Interval = new TimeSpan(0,0,0,0,50))
```

The event handler must replace the ball's old position with a black ball, move the ball to its new position, draw the new ball and the border, and finally call the bitmap's **Invalidate()** method to cause a redraw. The code to do this is fairly straightforward:

```
// Handler for the timer's tick event.
```

```
    let onTick() =
        // Write over the old ball.
        drawBall !ball Colors.Black
        // Move the current ball to its next position.
        ball := ball.Value.Move(0.,0.,xMax,yMax)

        // Draw ball and square.
        drawBall !ball Colors.Red
        drawSquare Colors.Red

        // Invalidate the bitmap to force a redraw.
        bitmap.Invalidate()
```

All that remains after that is to wire everything together. We'll define and call an **`init()`**function to do that. The **`init()`** function will fill the bitmap with the color black, assign the image to the user control's **`Content`** property so it will be seen in the user control, and finally add our **`onTick`** function to the timer's **`Tick`** event and start it.

```
    // Helper function to do some initialization.
    let init() =
        // First color our bitmap black.
        Array.fill
            bitmap.Pixels
            0 bitmap.Pixels.Length
            (colorToInt Colors.Black)

        // Make the image the user control's content.
        br.Content <- image

        // Set up the timer.
        timer.Tick.Add(fun _ -> onTick())
        timer.Start()

    do init()
```

That completes our work on the control to show the bouncing ball. All that remains is the Silverlight plumbing that will show the control when the application starts up. This is very straightforward:

```
type App() as this =
    inherit Application()

    do this.Startup.Add(fun _ -> this.RootVisual <- new BallRender())
```

Of course there's still a need to create a test HTML page that will load the Silverlight runtime along with the application we just created, but the template we've used should take care of that. Viewing the resulting simulation should just be a matter of pressing F5. You can see the simulation running on my machine in the following image.

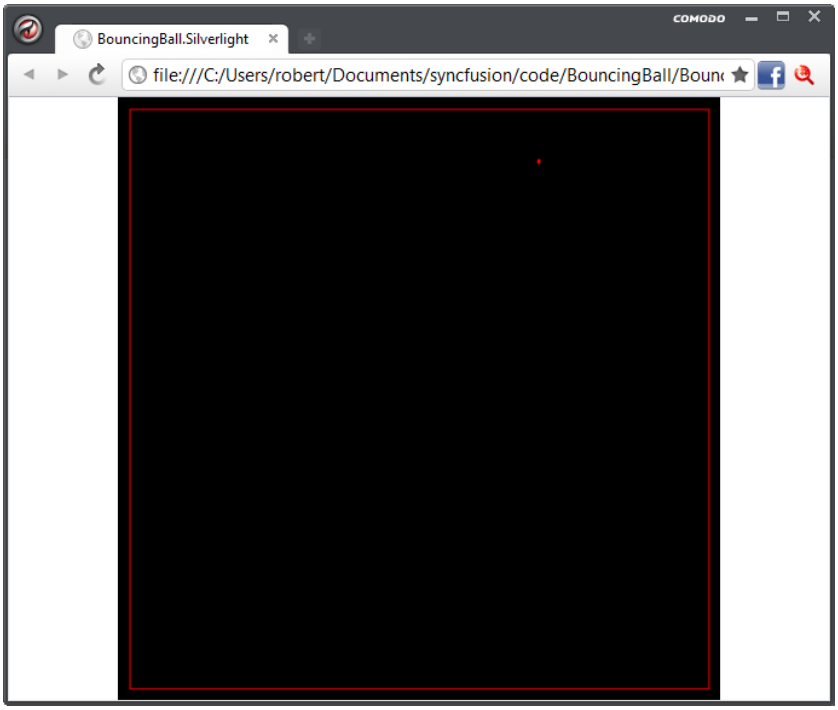

Figure 5: Bouncing Ball Simulation

Even though Silverlight's API offers significantly less low-level drawing functionality, we've seen that by creating a few simple and easy-to-use abstractions it's still fairly easy to create applications with custom graphics.

Summary

In this chapter we've seen how to approach building a simple simulation in F#, and how to use low-level drawing APIs to create custom graphics in F#.

Chapter 7 Form User Interfaces

The previous chapter looked at creating custom graphics using F#. In this chapter, we're going to look at how to create form-based applications in F#. There are a number of different technologies for creating form-based applications in .NET, including Windows Forms, WPF, Silverlight, and GTK#. Since each of these technologies is based on similar ideas, we'll stick to looking at one: WPF.

A Simple Form

WPF allows you to define forms in two ways: first by manipulating the WPF objects directly, and second by defining forms in an XML dialect called XAML. XAML is useful because there are several different user interface design packages available that allow designers create rich user experiences in XAML. Designers can then pass these over to developers to wire them into the application. We'll look at XAML later in the chapter; for now we'll take a look at how to create forms using the WPF objects directly.

Using the objects directly is a good approach when you want to create a simple form that only consists of a few controls. In this case, the creation of the form is probably simple enough that it's not worth the effort of using a designer. Also, creating forms in this way is a good learning experience because it helps you better understand how each of the control types fits together.

We want to create a form with three controls: a text box, a descriptive label in front of the text box, and a button immediately below it. To create our desired layout, we're going to use two stack panels: a horizontal one to hold the label and the text box, and a vertical one to hold the horizontal stack panel and the button directly below it.

To create the application, we're going to start a new F# project and add references to **PresentationCore.dll**, **PresentationFramework.dll**, **System.XAML.dll** (required even though we're not using any XAML at this stage), and finally **WindowsBase.dll**. We'll then need some open statements to provide easy access to the namespaces:

```
open System
open System.Windows
open System.Windows.Controls
```

We're almost ready to create our controls, but first we need to tidy up one slightly annoying aspect of WPF. In WPF, there is a collection called **`UIElementCollection`** which is used to store child controls. This collection has an **`Add`** method which has both a side effect—adding the control to the collection—and a returned value—the index of the newly added control. We almost never have any interest in the index of the control, so we can ignore the return value; however, F# will give us a warning that we are ignoring a return value because this is often the indication of an error in functional programming. To work around this we need to add a short

helper function to add the item to the control collection without returning the index. In WPF, not all controls can have collections of children; only controls that inherit from **`Panel`** have a **`Children`** collection. This is why we add a type constraint to the second parameter of the helper function shown in the following sample. The function will only accept controls that derive from **`Panel`** as its second parameter. The implementation of the **`addChild`** helper function is shown in the following code as well. It simply adds the given control to the **`Children`** collection and ignores the result.

```
// Adds a child to a panel control.
let addChild child (control: Panel) =
    control.Children.Add child |> ignore
```

Now we can start creating our controls. We're going to define a **`createForm`** function to handle creating the controls. There's really no magic to it. We simply create the horizontal stack panel and then add a label and text box to it. Next we create the vertical stack panel and add the horizontal stack panel, followed by the vertical, followed by the button control, all while not forgetting to wire up the button's event handler.

```
// Function to create the form interface.
let createForm() =
    // Horizontal stack panel to hold label and text box.
    let spHorozontal =
        new StackPanel(Orientation = Orientation.Horizontal)

    // Add the label to the stack panel.
    spHorozontal |> addChild (new Label(Content = "A field",
                                        Width = 100.))

    // Add a text box to the stack panel.
    let text = new TextBox(Text = "<enter something>")
    spHorozontal |> addChild text

    // Create a second stack panel to hold our label
    // and a text box with a button below it.
    let spVert = new StackPanel()
    spVert |> addChild spHorozontal

    // Create the button and make it show the content
    // of the text box when clicked.
    let button = new Button(Content= "Press me!")
    button.Click.Add(fun _ -> MessageBox.Show text.Text |> ignore)
    spVert |> addChild button
```

```
    // Return the outermost stack panel.
    spVert
```

And that's all there is to creating a form. To show the form, we need to host it in a window, and then create a WPF application to create an event loop and show the control. The code to do this is as follows:

```
// Create the window that will hold the controls.
let win = new Window(Content = createForm(),
                     Title = "A simple form",
                     Width = 300., Height = 150.)

// Create the application object and show the window.
let app = new Application()
[<STAThread>]
do app.Run win |> ignore
```

As we saw in the previous chapter, when starting in the WPF event loop we need to ensure the **`STAThread`** attribute is attached to the starting method call. On executing this application, we should see the following form:

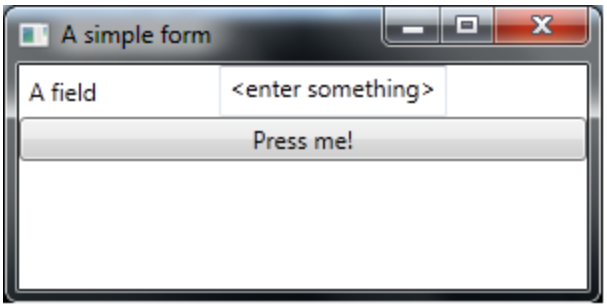

Figure 6: WPF Form Created in F#

A Form Using XAML

While the approach of creating the user interface yourself can work well for simple applications, if we want more complex styles and effects it's often better to use XAML. In this example we're going to look at creating a form with exactly the same layout and functionality as the previous form. The only thing that changes is the layout will now be defined in XAML and the behavior of the form will be defined using F#. The XAML definition of our form is the following:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="A XAML Form" Height="350" Width="525">
```

```
  <StackPanel>
    <StackPanel Orientation="Horizontal">
      <Label Width="100">A field</Label>
      <TextBox x:Name="MessageTextBox">&lt;enter something&gt;</TextBox>
    </StackPanel>
    <Button x:Name="PressMeButton">Press me!</Button>
  </StackPanel>
</Window>
```

It's fairly easy to see from the XAML definition that the layout consists of two stack panels, a label, a text box, and a button as before. We've given names to the text box and button as these are the objects we'll need to access from code. The text box is called **`MessageTextBox`** and the button is called **`PressMeButton`**. F# has slightly less integration with XAML than C#, so we need to define a couple of helper functions to help us load the XAML and access the controls defined within in it. In a small example like this, the extra code required for these helpers can seem like a lot of overhead, but in a more realistically sized application the cost of these extra functions will quickly be amortized as the helper functions are reused throughout the application.

The XAML definition we've just seen will need to be added to the F# project as **`MainWindow.xaml`**. In the properties window for this file you'll need to set the build action to **`EmbeddedResource`** so that it will be embedded into the assembly manifest as a resource stream. That's pretty much all we need to do for the XAML part.

Before we start on the F# part, we'll need to reference the same assemblies as the previous application (**PresentationCore.dll**, **PresentationFramework.dll**, **System.XAML.dll**, and **SystemXML.dll**) and we'll need the following **`open`** statements.

```
open System
open System.Reflection
open System.Windows
open System.Windows.Markup
open System.Windows.Controls
```

We need two helper functions: a function to load the XAML window, and an operator to help us access the controls that are defined within the window. The first function is straightforward. To get access to a manifest resource stream we need access to the assembly object that contains the resource. In this case, as our helper function is defined in the same assembly that the resource will be embedded in, we can use **`Assembly.GetExecutingAssembly()`**to get a reference to the assembly object. If the helper function was not in the same assembly as the resource file, for example, it might have been moved into another assembly so it could be shared between projects more easily. We would need to pass the assembly object to the function as a parameter. Once we have the resource stream we can load it using the **`XAMLReader`** class that is part of the WPF framework. This is what our function looks like:

```
// Load a XAML file from the current assembly.
let loadWindowFromResources name =
    let currentAssem = Assembly.GetExecutingAssembly()
    use xamlStream = currentAssem.GetManifestResourceStream(name)
    // Resource stream will be null if not found.
    if xamlStream = null then
        failwithf "The resouce stream '%s' was not found" name
    XamlReader.Load(xamlStream) :?> Window
```

There are a few points worth noting about the function. First, we use the **use** keyword instead of **let** to create the binding to our **xamlStream** identifier that represents the resource stream. The **use** keyword is equivalent to **using** in C#, and means that the stream will have its **Dispose** method called as soon as it drops out of scope. Secondly, we make a null check on the **xamlStream** identifier and raise an exception if the value is null. This is because the method **GetManifestResourceStream** returns null if no resource stream matching that name is found, and by raising an exception in this case we'll get a more meaningful exception. Finally, we load the XAML file using the **XAMLReader.Load** method. This method returns an object so we need to cast to the actual type of the object that we are expecting, which in this case is the **Window** class. We could easily have chosen to load the XAML from a file on the disk rather than a file embedded in the manifest; the advantage of using a file in the manifest is that we can be sure that it is distributed with the assembly.

The second function we need to define is a way to access the controls that are positioned within the window. The base class **FrameworkElement**, which **Window** inherits from, provides a **FindName** method to find our named controls. One possibility would be to use this method directly to find the controls we're interested in:

```
let pressMeButton = win.FindName("PressMeButton") :?> Button
```

While this is a perfectly acceptable way to do this, F# provides a fun alternative which can save you a little typing. F# lets you define a custom dynamic operator, which behaves in a similar way to the dynamic keyword in C#. The dynamic operator is a question mark (**?**) and it allows you to make what looks like a method or property call, but recover the method or property name as a string so that you can do a dynamic lookup. Here is how we would define our dynamic operator:

```
// Dynamic lookup operator for controls.
let (?) (c:obj) (s:string) =
    match c with
    | :? FrameworkElement as c -> c.FindName(s) :?> 'T
    | _ -> failwith "dynamic lookup failed"
```

Here we see a custom operator followed by two parameters: the first of type object and the second of type string. The first object is the object we're dynamically invoking, and the second is

the name of the property or method being called. The implementation of the operator pattern matches against the object to check if it is of type **FrameworkElement**. If it isn't, we throw an exception. If it is, we call the framework element's **FindName** method. Don't worry if that isn't 100% clear at the moment; the dynamic custom operator is one of the more advanced features of F#. The thing to retain is that defining this operator now lets you do a dynamic lookup on the window, which could be used to find the button we are interested in using:

```
let pressMeButton: Button = win?PressMeButton
```

Now that we have those two helper functions in place, we're ready to create our XAML-based window. To do this we'll implement a function, **createMainWindow**, which will be responsible for creating the window and wiring up events to the relevant controls.

```
// Creates our main window and hooks up the events.
let createMainWindow() =
    // Load the window from the resource file.
    let win = loadWindowFromResources "MainWindow.xaml"
    // Find the relevant controls in the window.
    let pressMeButton: Button = win?PressMeButton
    let messageTextBox: TextBox = win?MessageTextBox
    // Wire up an event handler.
    let onClick() =
        MessageBox.Show(messageTextBox.Text) |> ignore
    pressMeButton.Click.Add(fun _ -> onClick())
    // Return the newly created window.
    win

// Create the window.
let win = createMainWindow()

// Create the application object and show the window.
let app = new Application()
[<STAThread>]
do app.Run win |> ignore
```

The implementation of **createMainWindow** is quite straightforward. We load the window itself using the helper function **loadWindowFromResource**. Once we have the window, we grab references to two of the controls that it contains: the button, **pressMeButton**, and the text box, **messageTextBox**. Once we have these references it's easy to add an event handler to the button's click event and get access to the text box's **Text** property from this event handler. To complete our function, it's just a matter of returning our newly created window.

To show the window, we just have to call the **`createMainWindow`** function and then start the WPF event loop to show the window, just as we have done in the previous examples. The window will look exactly as it did in the previous example, but now that the layout of the window is defined in XAML it is easy to add some more styles so the controls won't look as plain.

A Form Using MVVM

A common way of implementing forms in WPF is to use the MVVM design pattern. For those of you who may be unfamiliar with MVVM, let's do a quick recap of what this design pattern is. MVVM stands for Model-View-ViewModel. In this design pattern, the Model is the objects that represent the data or domain. The View is the user interface—in this case we'll be using XAML, but we could use WPF objects directly if we so choose. The ViewModel is the layer of objects that sits between the View and the Model to provide the mapping between them and to react to events and changes within the GUI. The View communicates with the ViewModel through WPF's powerful data binding mechanism.

We'll take a look at implementing a simple form in the MVVM style. MVVM is a large topic and this example isn't designed to show all the techniques involved in implementing the MVVM design pattern. Instead, it aims to give you a taste of doing MVVM in F# and to allow you to apply your existing MVVM knowledge, or information from other MVVM articles, to an F# MVVM implementation. The example we're going to look at is how to implement a simple master detail page in the MVVM style. The application we're constructing will be for viewing our stock of robots. We'll be presented with a list of robots and allowed to click on a robot to see more details about it.

As we've already said, MVVM separates code into three different components: the model, the view, and the ViewModel. In addition to this, we'll add a "repository" which will abstract the data access logic. We'll go through the parts of the application in the following order: we'll look at the model and repository, then the ViewModel, and finally the view.

Because our application is a simple read-only view of some data, there is no domain logic. This means the model part of the application is very simple—only a data container is required. For the data container we'll use an F# record type:

```
type Robot =
    { Name: string
      Movement: string
      Weapon: string }
```

As you can see, we're only going to store three pieces of information about our robot: its name, how it moves, and its weapon. Our application is so simple that this is all we need for the model. For the sake of simplicity we're going to hard code the data in our repository, but in a more realistic example this would come from a database.

```
type RobotsRepository() =
    member x.GetAll() =
        seq{ yield {Name = "Kryten"
                    Movement = "2 Legs"
                    Weapon = "None" }
             yield {Name = "R2-D2"
                    Movement = "3 Legs with wheels"
                    Weapon = "Electric sparks" }
             yield {Name = "Johnny 5"
                    Movement = "Caterpillars"
                    Weapon = "Laser beam" }
             yield {Name = "Turminder Xuss"
                    Movement = "Fields"
                    Weapon = "Knife missiles" }
           }
```

Now that we have our model and repository, we're ready to look at the ViewModel. One of the main purposes of the ViewModel is to provide the view with some properties that it can data bind to. This means that the ViewModel class is made up of a number of small methods and properties that interact with each other, so I think it would be helpful for me to show you the whole ViewModel and then discuss the individual parts that comprise it.

```
type RobotsViewModel(robotsRepository: RobotsRepository) =
    // Backing field of the on property change event.
    let propertyChangedEvent =
        new DelegateEvent<PropertyChangedEventHandler>()


    // Our collection of robots.
    let robots =
        let allRobots = robotsRepository.GetAll()
        new ObservableCollection<Robot>(allRobots)

    // The currently selected robot
    // initialized to an empty robot.
    let mutable selectedRobot =
        {Name=""; Movement=""; Weapon= ""}

    // Default constructor, which creates a repository.
    new () = new RobotsViewModel(new RobotsRepository())

    // Implementing the INotifyPropertyChanged interface
    // so the GUI can react to events.
    interface INotifyPropertyChanged with
```

```
        [<CLIEvent>]
        member x.PropertyChanged = propertyChangedEvent.Publish

    // Helper method to raise the property changed event.
    member x.OnPropertyChanged propertyName =
        let parameters: obj[] =
            [| x; new PropertyChangedEventArgs(propertyName) |]
        propertyChangedEvent.Trigger(parameters)

    // Collection of robots that the GUI will data bind to.
    member x.Robots =
        robots

    // Currently selected robot that the GUI will data bind to.
    member x.SelectedRobot
        with get () = selectedRobot
        and set value =
            selectedRobot <- value
            x.OnPropertyChanged "SelectedRobot"
```

The first thing to notice about the **RobotsViewModel** is the two constructors. The first constructor accepts a **RobotsRepository** parameter so that the class can access the robot data. The rest of this constructor initializes fields that will be accessible to the rest of the class' methods. After these fields have been initialized, we then define a second constructor, one without parameters, which must call the first constructor. We call the first constructor and pass it a new instance of our repository:

```
    // Default constructor, which creates a repository.
    new () = new RobotsViewModel(new RobotsRepository())
```

Now that we've seen both constructors, let's take a look at each field we defined and how they are used in the class. First we define a field that will provide the backing store for our event handler. An event handler is needed to implement the **INotifyPropertyChanged** interface, which is how the ViewModel informs the view of changes. Creating the backing store is simple enough, we just need to create a **DelegateEvent** object and bind it to a field.

```
    // Backing field of the on property change event.
    let propertyChangedEvent =
        new DelegateEvent<PropertyChangedEventHandler>()
```

What is more interesting than the backing store for the event is how we use the **properyChangedEvent** field. We use this field in two ways: to implement the event itself, and to

create a method to raise the event. This is how we create the event by exposing the **DelegateEvent.Publish** property:

```
    // Implementing the INotifyPropertyChanged interface
    // so the GUI can react to events.
    interface INotifyPropertyChanged with
        [<CLIEvent>]
        member x.PropertyChanged = propertyChangedEvent.Publish
```

We see here that the event is exposed as part of the implementation of the **INotifyPropertyChanged** interface which has just one member: the event **PropertyChanged**. We need to mark the **PropertyChanged** with a **[<CLIEvent>]** attribute so that the F# compiler knows it should generate an event compatible with other CLR languages, such as C#, as F# has its own optimized systems of events. Now that we have exposed the event, we need to be able to invoke the event. We do this by creating an **OnPropertyChanged** method that will trigger the event.

```
    // Helper method to raise the property changed event.
    member x.OnPropertyChanged propertyName =
        let parameters: obj[] =
            [| x; new PropertyChangedEventArgs(propertyName) |]
        propertyChangedEvent.Trigger(parameters)
```

These three members of the class, the **propertyChangedEvent** field, the interface implementation of **INotifyPropertyChanged**, and the **OnPropertyChanged** method, would normally be placed in a base class so that they could be shared between all the ViewModels of the application. For the purposes of simplifying this example I've kept them in the same class as the ViewModel, as we only have one ViewModel in this application. The next two fields relate to properties that will be exposed to allow the view to bind to the ViewModel. The first field, **robots**, holds the collection of all robot data:

```
    // Our collection of robots.
    let robots =
        let allRobots = robotsRepository.GetAll()
        new ObservableCollection<Robot>(allRobots)
```

We've used an observable collection to represent the list of robots. This collection type will automatically notify the view if we add or remove items from the collection. We then expose this collection by a property:

```
    // Collection of robots that the GUI will data bind to.
    member x.Robots =
        robots
```

This **Robots** property will be bound to a list view control in the GUI that will display the list of robots. The next field **selectedRobot** represents the currently selected robot.

```
    // The currently selected robot
    // initialized to an empty robot.
    let mutable selectedRobot =
        {Name=""; Movement=""; Weapon= ""}
```

The field needs to be mutable as it will change over time. As the user selects a robot the field will be updated. This update will occur because of the data binding that we'll look at when we examine how the view is implemented. Again, we use a field to expose this property, but this time we need to provide both a getter and a setter:

```
    // Currently selected robot that the GUI will data bind to.
    member x.SelectedRobot
        with get () = selectedRobot
        and set value =
            selectedRobot <- value
            x.OnPropertyChanged "SelectedRobot"
```

The getter simply returns our field, but the setter must do two things. It must first update the field **selectedRobot**, and then call the **OnPropertyChanged** method to raise the property changed event that will notify the GUI of the change.

We've looked at everything that makes up the ViewModel, and now we'll take a look at the view itself. As this view is more complex than the previous XAML views that we've seen, I think it would be helpful to first look at an image of the layout, then take a look at the complete XAML listing, and finally walk through the important points of the XAML. The XAML listing is quite long, but don't worry about this too much. Most of the XAML just describes the positioning of the controls; there are only a few important bits that we need to examine in more detail.

First, let's look at a screenshot of the application:

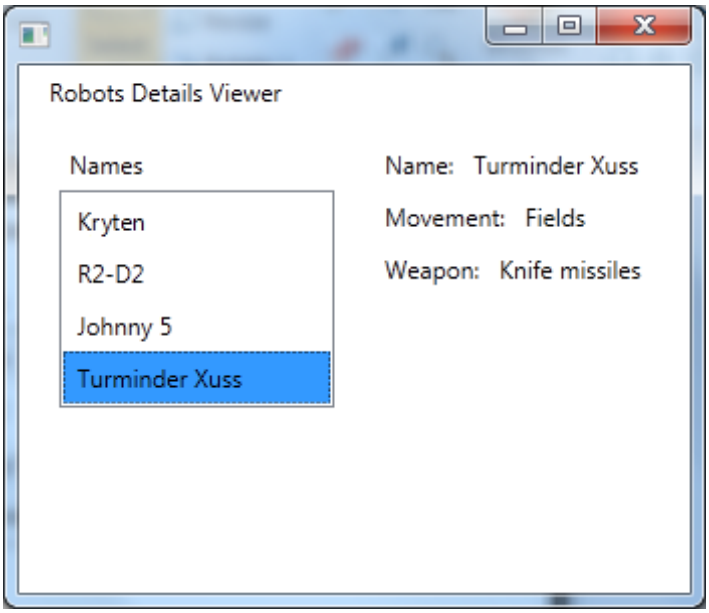

Figure 7: Robot Inventory Application

Here we see that the application is composed of a list box on the left side and some labels that show the details of the robot on the right side. As the user changes the selection, the details of the robot change.

This is what the full view listing looks like:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:ViewModel="clr-
namespace:FsSuccinctly.RobotsMvvm.ViewModel;assembly=Form_MVVM"
        mc:Ignorable="d"
        Width="350"
        Height="300">

  <!-- Create and data bind the ViewModel. -->
  <Window.DataContext>
    <ViewModel:RobotsViewModel></ViewModel:RobotsViewModel>
  </Window.DataContext>

  <Grid Margin="10,0,10,10" VerticalAlignment="Stretch">
```

```
<Grid.Resources>
  <!-- Name item template. -->
  <DataTemplate x:Key="nameItemTemplate">
    <Label Content="{Binding Path=Name}"/>
  </DataTemplate>
</Grid.Resources>

<Grid.ColumnDefinitions>
  <ColumnDefinition />
  <ColumnDefinition />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
  <RowDefinition Height="auto"/>
  <RowDefinition Height="auto"/>
  <RowDefinition Height="auto"/>
</Grid.RowDefinitions>
<!-- Robots list. -->
<Label Grid.Row="0" Grid.ColumnSpan="2">
  Robots Details Viewer
</Label>
<Grid Margin="10" Grid.Column="0"
      Grid.Row="1" VerticalAlignment="Top">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>

  <Border Grid.Row="1">
    <Label>Names</Label>
  </Border>

  <ListBox Name="robotsBox" Grid.Row="2"
       ItemsSource="{Binding Path=Robots}"
       ItemTemplate="{StaticResource nameItemTemplate}"
       SelectedItem="{Binding Path=SelectedRobot,Mode=TwoWay}"
       IsSynchronizedWithCurrentItem="True">
  </ListBox>

</Grid>
<Grid Margin="10" Grid.Column="1" Grid.Row="1"
      DataContext="{Binding SelectedRobot}" VerticalAlignment="Top">
  <Grid.ColumnDefinitions>
```

```
        <ColumnDefinition Width="57*" />
        <ColumnDefinition Width="125*" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>
      <!-- Name -->
      <StackPanel Grid.Column="0" Grid.ColumnSpan="2"
                  Grid.Row="0" Orientation="Horizontal">
        <Label>Name:</Label>
        <Label Content="{Binding Path=Name}"></Label>
      </StackPanel>
      <!-- Movement -->
      <StackPanel Grid.Column="0" Grid.ColumnSpan="2"
                  Grid.Row="1" Orientation="Horizontal">
        <Label>Movement:</Label>
        <Label Content="{Binding Path=Movement}"></Label>
      </StackPanel>
      <!-- Weapon -->
      <StackPanel Grid.Column="0" Grid.ColumnSpan="2"
                  Grid.Row="2" Orientation="Horizontal">
        <Label>Weapon:</Label>
        <Label Content="{Binding Path=Weapon}"></Label>
      </StackPanel>
    </Grid>
  </Grid>
</Window>
```

As you can see, the list for the XAML view is quite long, but don't worry, there are only a few key points to note. To begin with, let's look at how the ViewModel is bound to the view. First we need an attribute in the **`Window`** tag to create an alias so we can access the ViewModel's namespace:

```
xmlns:ViewModel="clr-
namespace:FsSuccinctly.RobotsMvvm.ViewModel;assembly=Form_MVVM"
```

We can now use the prefix **`ViewModel`** to let us access classes in the **`FsSuccinctly.RobotsMvvm.ViewModel`** as tags in our XAML document. This means we can create an instance of the ViewModel and bind it to the view's data context in XAML like this:

```
  <!-- Create and data bind the ViewModel. -->
```

```
<Window.DataContext>
  <ViewModel:RobotsViewModel></ViewModel:RobotsViewModel>
</Window.DataContext>
```

The next important part of the view is the list box that will display the robots, which is coded as follows:

```
    <ListBox Name="robotsBox" Grid.Row="2"
         ItemsSource="{Binding Path=Robots}"
         SelectedItem="{Binding Path=SelectedRobot,Mode=TwoWay}"

         ItemTemplate="{StaticResource nameItemTemplate}"

         IsSynchronizedWithCurrentItem="True">
    </ListBox>
```

This list box makes good use of WPF's powerful data binding. We bind the **`ItemsSource`** property to the ViewModel's **`Robot`** property so that the list box will display the list of robots. The **`SelectedItem`** property is bound to the **`SelectedRobot`** property. This takes advantage of WPF's two-way bindings, meaning that as the user updates the selected item in the user interface, the **`SelectedItem`** of the list view is also updated, and thanks to the two-way binding the **`SelectedRobot`** property is also updated. The **`ItemTemplate`** property allows you to control how each item in the list box is rendered. In this case we reference a template we've defined earlier:

```
  <Grid.Resources>
    <!-- Name item template. -->
    <DataTemplate x:Key="nameItemTemplate">
      <Label Content="{Binding Path=Name}"/>
    </DataTemplate>
  </Grid.Resources>
```

Now that we've seen how the list box works, we just need to look at how the robot details are displayed. Here we show three fields of information about the robot, but each field that is displayed has the same implementation, so we only really need to look at how one field is implemented. The three fields are displayed in a grid; we bind this **`Grid`** tag to our **`SelectedRobot`** property from the ViewModel. This will give us easy access to the fields we want to display in the labels that show the details of our selected robot. The following shows how the grid is implemented. The important property to note here is the **`DataContext`** property:

```
  <Grid Margin="10" Grid.Column="1" Grid.Row="1"
        DataContext="{Binding SelectedRobot}" VerticalAlignment="Top">
```

Now let's take a look at one of the fields within the grid. Here we see the implementation of the **Name** field:

```
<!-- Name -->
<StackPanel Grid.Column="0" Grid.ColumnSpan="2"
            Grid.Row="0" Orientation="Horizontal">
  <Label>Name:</Label>
  <Label Content="{Binding Path=Name}"></Label>
</StackPanel>
```

We can see how the **Content** property of the second label is bound to the **Name** property of the robot. That's all the important details of the view, and we're almost finished with the implementation of our XAML form. The only thing remaining is to load the XAML view and display it. We can do this using the helper functions we defined earlier in the chapter. As you can see, it's quite simple to load the XAML window and display it:

```
// Create the window.
let win = loadWindowFromResources "RobotsWindow.xaml"

// Create the application object and show the window.
let app = new Application()
[<STAThread>]
do app.Run win |> ignore
```

There is no need to wire up any events or do any other kind of configuration. The XAML view takes care of binding itself to the ViewModel that drives the rest of the interactions.

Summary

We've now seen how F# can be used in several different ways with WPF, including using WPF's powerful MVVM design pattern. Hopefully this has given you a good idea how F# can be used in the creation of business applications that require structured data to be input by the user. We focused on WPF, but there many other GUI libraries available on the .NET framework. Although we have not seen specific examples for these libraries, most of them share similar concepts to WPF. Hopefully you will find some of the ideas in this chapter usable with other libraries.

Chapter 8 Creating an Application

In this chapter I want to show what's involved in writing an application in F#. "Application" is probably too grand a word for what we're going to implement, but the point is to show F# interacting with other technologies and frameworks in the same way you would have to if you were to create a real-world application.

We're going to create an autocomplete drop-down in an HTML webpage. The idea is to type in the village, town, or city you're interested in, and the autocomplete will help you find the correct one in the system. We're going to use RavenDB as our data storage, and PicoMvc as the MVC web framework to handle the application layer. PicoMvc is an MVC framework that I have created specifically for F#. It was largely inspired by OpenRasta, FubuMvc, and a little bit of ASP.NET MVC, so the example given here will probably be quite easily portable to any of these frameworks if you prefer not to use PicoMvc. RavenDB is a NoSQL database implemented in C#. I like this database not because it is fast and provides good support for scalability through sharding, which it undoubtedly does, but because its implementers have a real focus on making it very easy for developers to use.

Creating an autocomplete drop-down in a HTML form is fairly common these days, and there's a nice jQuery plugin that takes care of the UI side of things. Loading the data into RavenDB and then exposing a service that will return JSON records corresponding to the user's search will be the focus of this chapter.

The full code of this example is available from the examples directory of PicoMvc itself at https://github.com/robertpi/PicoMvc/tree/master/examples/AutoComplete.

Project Setup

Our Visual Studio solution will have four projects: **Common.fsproj**, **LoadCommunes.fsproj** (commune is our generic term for village/town/city), **Web.fsproj**, and **WebHost.csproj**.

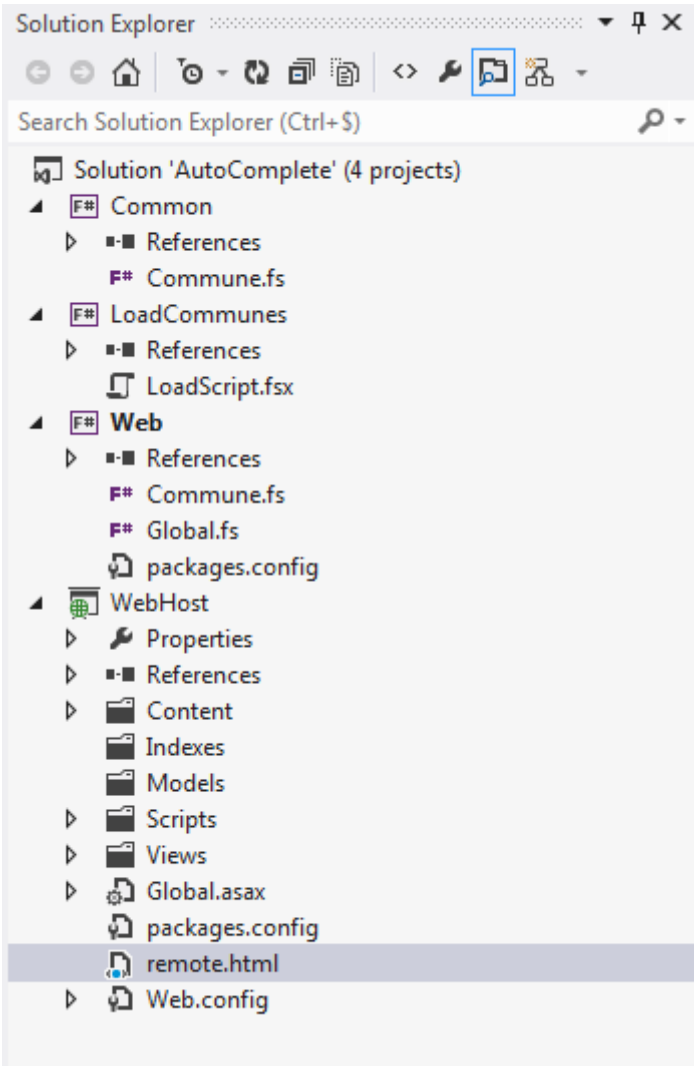

Figure 8: Autocomplete Project Setup

The **Common** project contains the definition of our types that will be stored in RavenDB. The common project will be referenced from the **LoadCommunes** and **Web** projects. The **LoadCommunes** project will contain the ETL logic to load the data into RavenDB. The **Web** project will contain the logic that drives the webpages. The **WebHost** project is a C# "web project"—it's just there to hold the HTML parts of the project and make launching the web server for debugging easier.

Now that the projects are set up to our satisfaction, it's time to look at some actual code in the ETL (extract, transform, and load) part of the application.

The ETL (Extract/Transform/Load)

Since I live in France, we're going to use data based on French villages, towns, and cities, but the techniques described here will be easily adaptable to wherever you live. First, we need to download the data. I obtained the data from http://www.galichon.com/codesgeo/. You can click

the **Télécharger la base** link under **Coordonnées géographiques des villes Françaises**, or download the file directly from http://www.galichon.com/codesgeo/data/ville.zip. It's not the best data source in the world, but it's the best freely available data source that I've found. Once you've unzipped the Excel file and converted the file to a CSV, loading it into RavenDB is pretty straightforward.

First we need to design a type to hold the data in the **Common** project:

```
type Commune =
    { mutable Id: string
      Name: string
      Postcode: string }
```

We're only going to store the name of the commune and its postcode because that's all we're going to search for or show—thus the fields **`Name`** and **`Postcode`**. RavenDB is pretty robust when it comes to adding or deleting fields, so it's fine to start with a minimal set of data and add stuff later. The **`Id`** field is the unique identifier of the record. It is mutable because this just seems to work better with RavenDB. We could let RavenDB generate this for us, but since INSEE, the French government's bureau for statics and economic studies, assigns each village its own unique identifier which is included in the file, we'll use this. In France, several communes can share the same postcode, so this would not be a good candidate for the identifier.

Once we've designed the type to store the commune data, the code to load it from the file and store it in RavenDB is fairly simple:

```
let loadCommuneData() =
    use store = DocumentStore.OpenInitializedStore()
    let lines = File.ReadLines(Path.Combine(__SOURCE_DIRECTORY__,
@"ville.csv"), System.Text.Encoding.Default)

    use session = store.OpenSession()
    session.Advanced.MaxNumberOfRequestsPerSession <- 30000
    lines
    |> Seq.skip 1
    |> Seq.iteri(fun i line ->
        let line = line.Split(';')
        match line with
        | [|  name; nameCaps; postcode; inseeCode; region; latitude;
longitude; eloignementf|] ->
            let id = sprintf "communes/%s" (inseeCode.Trim())
            printfn "Doing %i %s (%s)" i name id
            let place: Commune =
                { Id = id
                  Name = name.Trim()
```

```
                Postcode = postcode.Trim() }
            session.Store(place)
            if i % 1000 = 0 then session.SaveChanges()
        | line -> printfn "Error in line: %A" line)
    session.SaveChanges()
```

There are a few points worth highlighting:

- We use **`File.ReadLines`** to give us an IEnumerable of all the lines in the file. This gives us a nice convenient way to read the file line by line without loading it all into memory.

- Notice we're passing **`System.Text.Encoding.Default`** to **`File.ReadLines`**. French communes often have accented characters in their names, so we need to ensure we're using the right encoding.

- It's necessary to set the **`session.Advanced.MaxNumberOfRequestsPerSession`** because it is limited to 10 by default, meaning that after 10 requests or stores the session would throw an exception. Typically sessions are meant to be short lived, so this exception is meant as an early warning for developers. Since this is an atypical use of a session it's okay to set this number. However, I think sessions cache the data that they store, so you may want to clear the session after each write to RavenDB. It doesn't seem to make much difference in this case.

- We enumerate each row in the file using **`Seq.iteri`**. This gives us the row plus the row number. We can use the row number to perform a save every 1000 items by calling **`.SaveChanges()`**. This seems to be more efficient than either saving after each row or trying to save the whole lot all at once. I haven't done much experimentation with this number; there may be a more optimal number than 1000.

- The parsing of the file is very simple, we simply call **`.Split(';')`** on each row and then pattern match over the resulting array to unpack the relevant items. These are then loaded into the **`Commune`** type and stored in RavenDB using the session's **`Store()`** method. As mentioned in the previous bullet, these aren't flushed to the DB until you call **`.SaveChanges()`**.

And that about wraps it up. The data is in the database, and you can verify this using RavenDB's administrative console as shown in the following figure.

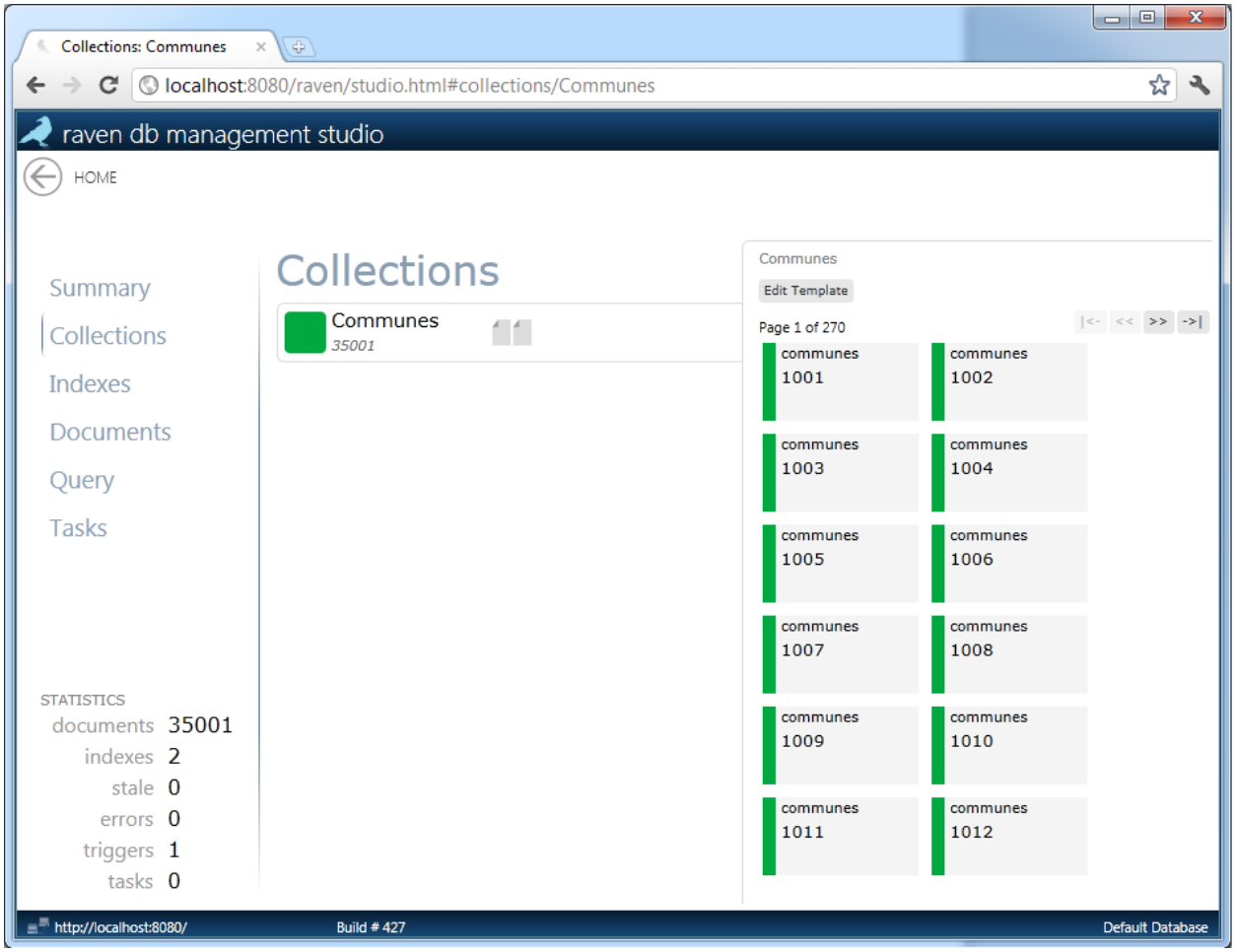

Figure 9: Administrative Console for Database

Code Supporting the Website

Now that we have the data in RavenDB, we'd like to be able to show the data to the user. To do this we need to create a service that will return a JSON document to the client. Implementing the service with PicoMvc and RavenDB is simple enough, but to do it we need to configure PicoMvc and create an index in RavenDB so that we can query it.

PicoMvc is designed to map an F# function to a URL and HTTP verb. The idea is that the basic framework is independent of the web server and host platform, but we provide hooks to allow you to plug PicoMvc into existing platforms. For now, the only hook that exists is the one to plug PicoMvc into the ASP.NET platform, and this is done via a class called **`PicoMvcRouteHandler`**, which is an ASP.NET route handler. The idea is that you register this route handler with the ASP.NET runtime, and it provides all the plumbing for mapping the calls the ASP.NET HTTP handler will receive and requests to the handler functions you have defined via PicoMvc.

As **PicoMvcRouteHandler** is just a normal HTTP handler. It requires that you register it with the ASP.NET runtime in the **global.asax**:

```
        routes.Add(new Route("{*url}", new PicoMvcRouteHandler("url",
routingTables, actions)))
```

PicoMvcRouteHandler also requires a little configuration. The route handler's first parameter is a string which tells it the name of the URL you matched against when you added the route handler. It will then use this as the URL to resolve which function will be called. The next parameter is a routing table, which holds the information about which functions should be called for which URLs. You can make PicoMvc automatically search all loaded assemblies for F# modules marked with the **[<Controller>]** attribute by calling the static method **LoadFromCurrentAssemblies**.

```
        let routingTables = RoutingTable.LoadFromCurrentAssemblies()
```

The route handler's third and final parameter tells the route handler how the parameters and return results of the functions that are dynamically invoked should be handled. There are a number of predefined actions in PicoMvc. For example, there is an action which will look up a value in the query string or post variables based on the parameter's name. This is called **ParameterActions.defaultParameterAction**. Defining new actions is fairly easy as well. For example, we'll need an action that returns a reference to the RavenDB document store if ever we see a type of **IDocumentStore**. This is how we would make it:

```
        let ravenParameterAction =
            { CanTreatParameter = fun _ _ t -> t = typeof<IDocumentStore>
              ParameterAction = fun _ _ _ -> !store :> obj }
```

It's also necessary to define an index on our data that RavenDB will use in its queries. Since this uses the C# "monadic" syntax, it is closely tied to the C# language. I find it easier to implement this in C#:

```
    public class Communes_Search : AbstractIndexCreationTask
    {
        public override IndexDefinition CreateIndexDefinition()
        {
            return new IndexDefinitionBuilder<Commune>
            {
                Map = communes => from commune in communes select new {
commune.Name, commune.Postcode }
            }.ToIndexDefinition(this.Conventions);
        }
    }
```

It is then simple to load this from our **global.asax**:

```
        let assem = Assembly.Load("WebHost")
        IndexCreation.CreateIndexes(assem, !store)
```

Now that we've tackled configuring both PicoMvc and RavenDB, we're ready to attack implementing the service itself.

The JSON Service

To create the autocomplete drop-down we need to query RavenDB and then send the results to the user's webpage as a JSON document. Implementing the JSON service is pretty straightforward:

```
type AutoCompleteResult =
    { id: string;
      label: string;
      value: string }

[<Controller>]
module Commune =
    let get (term: string) (store: IDocumentStore) =
        use session = store.OpenSession()
        let postcodeRegex = new Regex(@"^\d+$")

        let comQuery =
session.Advanced.LuceneQuery<Commune>("Communes/Search")
        let comQuery =
            if postcodeRegex.IsMatch term then
                comQuery.WhereStartsWith("Postcode", term)
            else
                comQuery.WhereStartsWith("Name", term)
        let query = comQuery.Take(20)
        let res = query |> Seq.map (fun x -> { id = x.Id; label = sprintf "%s
(%s)" x.Name x.Postcode; value = sprintf "%s (%s)" x.Name x.Postcode})
        Result res
```

First we define a type, **AutoCompleteResult**, to hold the results we want to send back to the client. This will be directly translated into JSON.

Next we define a PicoMvc controller. This is just an F# module marked with the **[<Controller>]** attribute. Because the module's name is **Commune**, it will be exposed at the URL **~/commune.xxx**, where xxx is used to determine which view will be used to render the

result (how we choose a view to handle results is discussed later). In this case, our URL will be **~/commune.json** and the view will render the result as JSON.

Our controller defines functions that handle the different HTTP verbs it might receive. In this case, we only want to handle GET verb requests, so we define a **`get`** function. The function's parameter term will be populated by the item **`term`** from the query string, and **`store`** will be populated with a reference to the RavenDB document store.

Once we have the **`term`** parameters and the **`store`** parameter to give us a reference to our RavenDB store, implementing the service is simple. We use the advanced Lucene query to query the **`Communes/Search`** index. Then we test if our input term is a postcode or a commune. In France, postcodes are completely numeric so we can perform this test using a simple regex. Once we know whether the term is a post code or town name, we can use the Lucene query's **`WhereStartsWith`** method to query against the **`Name`** or **`Postcode`** fields. We then limit our query to 20 results and transform it into the format the client is expecting.

Creating a UI for this is easy enough. I based my UI on the **remote.html** example formerly available in the jQuery developer bundle (the example is included in the downloadable code samples for this book). It's just a matter of changing the URL and a few of the labels to get it working.

There are a couple of improvements that could be made:

- A large number of communes start with the word "Saint." A user might reasonably expect the abbreviation "St." to map to "Saint."
- Words in the commune names are separated with dashes. It may be a good idea to allow users to use spaces instead.
- I'm not sure how well RavenDB handles accents. It would be nice to map "e" to both "é" and "è". It is fairly easy to code this functionality yourself if RavenDB supports it; you simply need to store an accent-free version of the names and search based on that.

Summary

That's it. It took a bit of explaining, but we learned plenty about PicoMvc along the way, and the final solution wasn't very much code at all. In fact, the entire solution came to about 150 lines of F# and C#. You can see the final solution in the **examples** directory of PicoMvc on github at https://github.com/robertpi/PicoMvc/tree/master/examples/AutoComplete.

Further Reading

I hope you've enjoyed this introduction to F#. If it has inspired you to learn more about F# there is plenty of other material available for further reading about the language. The official MSDN development center for F# has plenty of material about F# and lots of jumping off points to community resources at http://fsharp.net.

Another good source of information about F# is the developers' blogs with F# content. A wide range of bloggers write about F# with different viewpoints. Here are just a few to start with:

- Don Syme's Blog at http://blogs.msdn.com/b/dsyme/. F# creator and lead architect on the F# project.
- The F# Team Blog: http://blogs.msdn.com/b/fsharpteam/.
- My own blog, although updates can be a bit sporadic: http://strangelights.com/blog.

There are many in the F# community who are active on Twitter. Regularly following Twitter's *fsharp* hashtag is another good way to find out the latest F# news.

If you are interested in more books about F#, you may consider:

Beginning F# by Robert Pickering, Apress, December 2009. My own introduction to F#. It includes a more complete and detailed explanation of the language's syntax, as well as chapters on topics such as parallel programming and language-oriented programming.

Expert F# by Don Syme et al., Apress, June 2010. An excellent guide to F# written by the language's creator unashamedly aimed at expert programmers.