Real-World Functional Programming

With examples in F# and C#

SAMPLE CHAPTER

Tomas Petricek
with Jon Skeet

Foreword by Mads Torgersen

MANNING

Real-World Functional Programming

by Tomas Petricek
with Jon Skeet

Chapter 4

brief contents

4 Exploring F# and .NET libraries by example

This chapter covers

- Working with common .NET and F# libraries
- Implementing our first real-world application in F#
- Developing code using F# Interactive
- Loading data from file and drawing charts

Even though we've looked at only the most basic F# language features so far, you should already know enough to write a simple application. In this chapter we won't introduce any new functional language constructs; instead we'll look at practical aspects of developing .NET applications in F#. You probably know how to write a similar application in C#, so all code in this chapter will be in F#.

As we write our first real-world application in F#, we'll explore several functions from the F# library and also see how to access .NET classes. The .NET platform contains many libraries and all of them can be used from F#. In this chapter we'll look at several examples, mainly in order to work with files and create the UI for our application. We'll come across several other .NET libraries in the subsequent chapters, but after reading this one you'll be able to use most of the functionality provided by .NET from your F# programs, because the technique is often the same.

4.1 Drawing pie charts in F#

We'll develop an application for drawing pie charts, as shown in figure 4.1. The application loads data from a comma-separated value (CSV) file and performs preprocessing in order to calculate the percentage of every item in the data source. Then it plots the chart and allows the user to save the chart as a bitmap file. We could use a library to display the chart (and we'll do that in chapter 13), but by implementing the functionality ourselves, we'll learn a lot about F# programming and using .NET libraries from F# code.

We'll implement the application in three parts:

- Section 4.2—We'll implement loading information from a file and perform basic calculations on the data. We'll use the tuple and list types that we introduced in the previous chapter.
- Section 4.3—We'll add simple console-based output, so we can see the results of the calculations in a human-readable form.
- Section 4.4—We'll add a graphical user interface (GUI), drawing charts of the data. We'll use the standard .NET Windows Forms library to implement the UI and the `System.Drawing` namespace for drawing.

Even though you're only a quarter of the way through this book, the code that we'll write will be very close to what you'd do if you wanted to develop an application like this after reading the entire book. This is because F# code is developed in an iterative way: you start with the simplest possible way to solve the problem and later refine it to fit your advanced needs. Many people prefer developing F# code like this because it allows you to get interesting results as soon as possible, and because F# makes it easy to refactor code later to improve its organization and readability. The ability to quickly write a working prototype for a problem is extremely useful.

One benefit of iterative development is that you can interactively test your application when writing the first version, as you'll see in this chapter. In the early phase, you

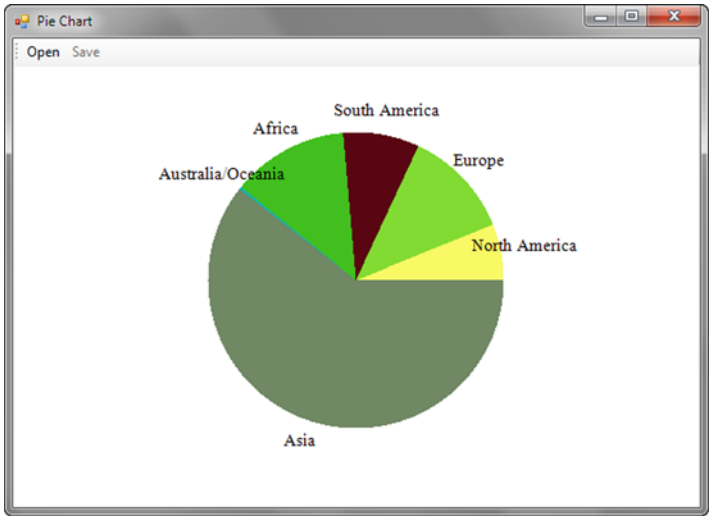

Figure 4.1 Running the F# application for drawing pie charts developed in this chapter. The chart shows distribution of the world population among continents.

can also easily author unit tests (learn more in chapter 11). Another benefit is that it's easier to correctly design the whole application if you already know how the core parts look in the prototype. Also, F# and Visual Studio are perfect tools for this kind of development. You can start writing the code in Visual Studio and execute it using F# Interactive to see whether it works as you expected and later start wrapping this experimental code in modules or types.

4.2 Writing and testing code in FSI

From the description in section 4.1, you should have a good idea of what kind of data we'll be using. The application works with a series of elements containing a title to be displayed in the chart and a number. It will load the data from a simplified CSV file, which contains a single element per line. Listing 4.1 shows a sample file with world population distribution in millions.

Listing 4.1 Our CSV file with population information

```
Asia,3634
Australia/Oceania,30
Africa,767
South America,511
Europe,729
North America,307
```

CSV files like this one are supported by many spreadsheet editors, including Microsoft Excel, so if you save the file with the .csv extension, you can easily edit it. Our application will only support basic files, so we'll assume that values are separated using commas and that there are no commas or quotation marks in the titles. Those extra elements would make the file format more complicated, leading to more complex parsing code.

Let's start by writing F# functions to read the file and perform basic calculations on the loaded data. We'll develop the code interactively, which will allow us to test every single function immediately after writing it.

4.2.1 Loading and parsing data

As a first step, we'll implement a function called `convertDataRow`, which takes a single row from the CSV file as a string and returns two components from the row in a tuple. Immediately after implementing the function, we test it by giving it a sample input that should be correctly parsed (a string "Testing reading,1234"). You can see the code for this function and the result of our test in listing 4.2.

Listing 4.2 Parsing a row from the CSV file (F# Interactive)

```
> open System;;
> let convertDataRow(csvLine:string) =                    (1)
    let cells = List.ofseq(csvLine.Split(','))            (2)
    match cells with
    | title::number::_ ->      <-- Should have two or more cells
```

```
      let parsedNumber = Int32.Parse(number)
      (title, parsedNumber)
    | _ -> failwith "Incorrect data format!"      <-- Reports an error
  ;;
val convertDataRow : string -> string * int

> convertDataRow("Testing reading,1234");;          (3)
val it : string * int = ("Testing reading", 1234)
```

After starting F# Interactive, we import functionality from the `System` namespace. We need to open the namespace because the code uses the `Int32.Parse` method. This has to be imported explicitly, whereas the functions from the core F# libraries, such as `List.ofseq`, are available implicitly.

The function `convertDataRow` (1) takes a string as an argument and splits it into a list of values using a comma as a separator. We're using the standard.NET `Split` method to do this. When invoking an instance method on a value, the F# compiler needs to know the type of the value in advance. Unfortunately, the type inference doesn't have any other way to infer the type in this case, so we need to use type annotation to explicitly state that the type of `csvLine` is a string (2).

The `Split` method is declared using the C# `params` keyword and takes a variable number of characters as arguments. We specify only a single separator: the comma character. The result of this method is an array of strings, but we want to work with lists, so we convert the result to a list using the `ofseq` function from the F# `List` module. We'll talk about arrays and other collection types in chapters 10 and 12.

Once we have the list, we use the `match` construct to test whether it's in the correct format. If it contains two or more values, it will match the first case (`title::number::_`). The title will be assigned to a value `title`, the numeric value to `number`, and the remaining columns (if any) will be ignored. In this branch we use `Int32.Parse` to convert a string to an integer and return a tuple containing the title and the value. The second branch throws a standard .NET exception.

If you look at the signature, you can see that the function takes a string and returns a tuple containing a string as the first value and an integer as the second value. This is exactly what we expected: the title is returned as a string and the numeric value from the second column is converted to an integer. The next line demonstrates how easy it is to test the function using F# Interactive (3). The result of our sample call is a tuple containing "Testing reading" as a title and "1234" as a numeric value.

Working with .NET strings in F#

When working with strings in F#, you'll usually use the normal .NET methods. Let's see how we can use them in F#, starting with a few selected static methods available in the `String` class. We can use these as if they were ordinary F# functions (using the `String` class name). The arguments to these functions must be specified in parentheses as a comma-separated tuple. In the type signatures, tuples are written using asterisks:

(continued)

- `String.Concat` (overloaded)—Accepts a variable number of arguments of type string or object and returns a string obtained by concatenating all of them:

```
> String.Concat("1 + 3", 3);;
val it : string = "1 + 33"
```

- `String.Join (sep:string * strs:string[]) : string`—Concatenates an array of strings supplied as the `strs` parameter using a separator specified by `sep`; we can use the `[| ... |]` syntax to construct an array literal:

```
> String.Join(", ", [| "1"; "2"; "3" |]);;
val it : string = "1, 2, 3"
```

Strings in .NET are also objects and they also have instance members too. These can be used in F# using the typical dot notation. We've already seen this in the previous example when splitting a string using `str.Split`. The following examples assume that we have a string value `str` containing `"Hello World!"`:

- `str.Length`—Property that returns the length of the string; properties are accessed in F# the same way as in C#, so the call reading the property isn't followed by braces:

```
> str.Length;;
val it : int = 12
```

- `str.[index:int]`—Indexing into a string, which can be written using square braces; returns the character at the location specified by the `index` value. Note that you still need the dot before the opening brace, unlike in C#:

```
> str.[str.Length - 1];;
val it : char = '!'
```

We can also use functions available in the FSharp.PowerPack.dll library. Most of the string processing code in F# can be implemented using .NET methods.

In the previous listing we implemented the `convertDataRow` function, which takes a string containing a line from the CSV file and returns a tuple containing a label and a numeric value. As a next step we'll implement a function that takes a list of strings and converts each string to a tuple using `convertDataRow`. Listing 4.3 shows the function—and a test immediately afterward, parsing a sample list of strings.

Listing 4.3 Parsing multiple lines from the input file (F# Interactive)

```
> let rec processLines(lines) =
    match lines with
    | [] -> []                                          (1)
    | currentLine::remaining ->                         (2)
      let parsedLine = convertDataRow(currentLine)
      let parsedRest = processLines(remaining)
      parsedLine :: parsedRest
```

Processes head of list

Recursively processes tail

```
  ;;
val processLines : string list -> (string * int) list

> let testData = processLines ["Test1,123"; "Test2,456"];;      ❸
val testData : (string * int) list =
   [("Test1", 123); ("Test2", 456)]
```

This function is in many ways similar to those for processing lists that we implemented in the previous chapter. As you can see, the function is declared using the `let rec` keyword, so it's recursive. It takes a list of strings as an argument (`lines`) and uses pattern matching to test whether the list is an empty list or a cons cell. For an empty list, it directly returns an empty list of tuples ❶. If the pattern matching executes the branch for a cons cell ❷, it assigns a value of the first element from the list to the value `currentLine` and list containing the remaining elements to the value `remaining`. The code for this branch first processes a single row using the `convertDataRow` function from listing 4.2 and then recursively processes the rest of the list. Finally the code constructs a new cons cell: it contains the processed row as a head and the recursively processed remainder of the list as a tail. This means that the function executes `convertDataRow` for each string in the list and collects the results into a new list.

To better understand what the `processLines` function does, we can also look at the type signature printed by F# Interactive. It says that the function takes a list of strings (`string list` type) as an argument and returns a list containing tuples of type `string * int`. This is exactly the type returned by the function that parses a row, so it seems that the function does the right thing. We verify this by calling it with a sample list as an argument ❸. You can see the result of the call printed by F# Interactive: it's a list containing two tuples with a string and a number, so the function works well.

Now we have a function for converting a list of strings to a data structure that we'll use in our chart-drawing application. Before we can implement the key data processing part, we need to look at one simple utility.

4.2.2 Calculating with the data

In the first version of the application, we'll simply print labels together with the proportion of the chart occupied by each item (as a percentage).

To calculate the percentage, we need to know the sum of the numeric values of all the items in the list. This value is calculated by the function `calculateSum` in listing 4.4.

Listing 4.4 Calculating a sum of numeric values in the list (F# Interactive)

```
> let rec calculateSum(rows) =
    match rows with
    | [] -> 0                                  Returns zero for empty list
    | (_, value)::tail ->                                              ❶
      let remainingSum = calculateSum(tail)
      value + remainingSum                     Recursively sums elements of tail
  ;;
val calculateSum : ('a * int) list -> int
```

```
> let sum = calculateSum(testData);;
val sum : int = 579
> 123.0 / float(sum) * 100.0;;      ❷
val it : float = 21.24352332
```

This function exhibits the recurring pattern for working with lists yet again. Writing code that follows the same pattern over and over may suggest that we're doing something wrong (as well as being boring—repetition is rarely fun). Ideally, we should only write the part that makes each version of the code unique without repeating ourselves. This objection is valid for the previous example, and we can write it in a more elegant way. We'll learn how to do this in upcoming chapters. You'll still need both recursion and pattern matching in many functional programs, so it's useful to look at one more example and become familiar with these concepts.

For an empty list, the function `calculateSum` simply returns 0. For a cons cell, it recursively sums values from the tail (the original list minus the first element) and adds the result to a value from the head (the first item from the list). The pattern matching in this code demonstrates one interesting pattern that's worth discussing. In the second branch ❶, we need to decompose the cons cell, so we match the list against the `head::tail` pattern. The code is more complicated than that, since at the same time, it also matches the head against pattern for decomposing tuples, which is written as `(first, second)`. This is because the list contains tuples storing the title as the first argument and the numeric value as the second argument. In our example, we want to read the numeric value and ignore the title, so we can use the underscore pattern to ignore the first member of the tuple. If we compose all these patterns into a single one, we get `(_, value)::tail`, which is what we used in the code.

If we look at the function signature printed by F# Interactive, we can see that the function takes a list of tuples as an input and returns an integer. The type of the input tuple is `'a * int`, which means that the function is generic and works on lists containing any tuple whose second element is an integer. The first type is not relevant, because the value is ignored in the pattern matching. The F# compiler makes the code generic automatically in situations like this using a feature called *automatic generalization*. You'll learn more about writing generic functions and automatic generalization in chapters 5 and 6.

The last command ❷ from listing 4.3 prepared the way for the test in listing 4.4: why enter test data more than once? Having calculated the sum to test the function, we finally calculate the percentage occupied by the record with a value 123. Because we want to get the precise result (21.24 percent), we convert the obtained integer to a floating point number using a function called `float`.

Converting and parsing numbers

F# is a .NET language, so it works with the standard set of numeric types available within the platform. The following list shows the most useful types that we'll work with. You can see the name of the .NET class in italics and the short name used in F# in parentheses:

(continued)

- *Int32, UInt32* (int, uint32)—Standard 32-bit integer types; literals are written in F# as `42` (signed) or `42u` (unsigned); there are also 16- and 64-bit variants written as `42s` and `42us` for 16-bit (int16, uint16) and `1L` or `1UL` for 64-bit (int64, uint64).
- *Double, Single* (float, float32)—Represent a double-precision and a single-precision floating-point number; the literals are written as `3.14` and `3.14f`, respectively. Note the difference between F# and C# here—`double` in C# is `float` in F#; `float` in C# is `float32` in F#.
- *SByte, Byte* (sbyte, byte)—Signed and unsigned 8-bit integers; the literals are written as `1y` (signed) and `1uy` (unsigned).
- *Decimal* (decimal)—Floating decimal point type, appropriate for financial calculations requiring large numbers of significant integral and fractional digits. Literals are written as `3.14M`.
- *BigInteger* (bigint)—A type for representing integers of arbitrary size. This is a new type in .NET 4.0 and is available in the `System.Numerics` namespace; earlier versions of F# contain their own implementation of the type, so you can use it when targeting earlier versions of the .NET Framework in Visual Studio 2008. In F#, the literals of this type are written as `1I`.

Unlike C#, the F# compiler doesn't insert automatic conversions between distinct numeric types when precision can't be lost. F# also doesn't use a type-cast syntax for explicit conversions, so we have to write all conversions as function calls. The F# library contains a set of conversion functions that typically have the same name as the F# name of the target type. The following list shows a few of the most useful conversion functions:

- *int*—Converts any numeric value to an integer; the function is polymorphic, which means that it works on different argument types. We can, for example, write `(int 3.14)`, for converting a float value to an integer, or `(int 42uy)`, for converting a byte value to an integer.
- *float, float32*—Converts a numeric value to a double-precision or a single-precision floating-point number; it's sometimes confusing that `float` corresponds to .NET `Double` type and `float32` to .NET `Single` type.

These functions can be also used when converting strings to numbers. If you need more control over the conversion and specify for example the culture information, you can use the `Parse` method. This method is available in a .NET class corresponding to the numeric type that can be found in a `System` namespace. For example, to convert a string to an integer you can write `Int32.Parse("42")`. This method throws an exception on failure, so there's also a second method called `TryParse`. Using this method, we can easily test whether or not the conversion succeeded. The method returns the Boolean flag and gives us the parsed number via an `out` parameter, but it can be accessed in a simpler way in F#. We'll talk about the details in chapter 5, but as you can see, the usage is straightforward:

(continued)

```
let (succ, num) = Int32.TryParse (str)
if succ then Console.Write("Succeeded: {0}", num)
else Console.Write("Failed")
```

This is by no means a comprehensive reference for working with numbers in F# and .NET. We've discussed only the most commonly used numeric types and functions. To learn more, refer to the standard .NET reference or the F# online reference [F# website].

In listing 4.4 we ended with an equation that calculates the percentage of one item in our test data set. This is another example of iterative development in F#, because we'll need exactly this equation in the next section. We tried writing the difficult part of the computation to make sure we could do it in isolation: now we can use it in the next section. We'll start by writing code to read the data from a file and then use this equation as a basis for code to print the data set to the console.

4.3 Creating a console application

Writing a simple console-based output for our application is a good start, because we can do it relatively easily and we'll see the results quickly. In this section, we'll use several techniques that will be important for the later graphical version as well. Even if you don't need console-based output for your program, you can still start with it and later adapt it into a more advanced, graphical version as we'll do in this chapter.

We've already finished most of the program in the previous section by writing common functionality shared by both the console and graphical versions. We have a function, `processLines`, that takes a list of strings loaded from the CSV file and returns a list of parsed tuples, and a function, `calculateSum`, that sums the numerical values from the data set. In listing 4.4, we also wrote the equation for calculating the percentage, so the only remaining tasks are reading data from a file and printing output to a console window. You can see how to put everything together in listing 4.5.

Listing 4.5 Putting the console-based version together (F# Interactive)

```
> open System.IO;;

> let lines = List.ofseq(File.ReadAllLines(@"C:\Ch03\data.csv"));;   (1)
val lines : string list

> let data = processLines(lines);;
val data : (string * int) list =
  [("Asia", 3634); ("Australia/Oceania", 30); ("Africa", 767);
   ("South America", 511); ("Europe", 729); ("North America", 307)]

> let sum = float(calculateSum(data));;
val sum : float = 5978.0

> for (title, value) in data do   (2)
```

Converts lines to list of tuples

Sums numeric values

```
    let percentage = int((float(value)) / sum * 100.0)
    Console.WriteLine("{0,-18} - {1,8} ({2}%)",
                      title, value, percentage)
  ;;
Asia                - 3634 (60%)
Australia/Oceania   -   30 (0%)
Africa              -  767 (12%)
South America       -  511 (8%)
Europe              -  729 (12%)
North America       -  307 (5%)
```

Calculates percentage, prints it

Listing 4.5 starts by opening the `System.IO` namespace, which contains .NET classes for working with the filesystem. Next, we use the class `File` from this namespace and its method `ReadAllLines` ❶, which provides a simple way for reading text content from a file, returning an array of strings. Again we use the `ofseq` function to convert the array to a list of strings. The next two steps are fairly easy, because they use the two functions we implemented and tested in previous sections of this chapter—we process the lines and sum the resulting values.

Let's now look at the last piece of code ❷. It uses a `for` loop to iterate over all elements in the parsed data set. This is similar to the `foreach` statement in C#. The expression between keywords `for` and `in` isn't just a variable; it's a pattern. As you can see, pattern matching is more common in F# than you might expect! This particular pattern decomposes a tuple into a title (the value called `title`) and the numeric value (called `value`). In the body of the loop, we first calculate the percentage using the equation that we tested in listing 4.4 and then output the result using the familiar .NET `Console.WriteLine` method.

Formatting strings in F# and .NET

String formatting is an example of a problem that can be solved in two ways in F#. The first option is to use functionality included in the F# libraries. This is compatible with F# predecessors (the OCaml language), but it's also designed to work extremely well with F#. The other way is to use functionality available in the .NET Framework, which is sometimes richer than the corresponding F# functions. The `printfn` function, which we've used in earlier examples, represents the first group, and `Console.WriteLine` from the last listing is a standard .NET method.

When formatting strings in .NET, we need to specify a *composite format string* as the first argument. This contains placeholders that are filled with the values specified by the remaining arguments. The placeholders contain an index of the argument and optionally specify alignment and format. Two of the most frequently used formatting methods are `Console.WriteLine` (for printing to the console) and `String.Format` (which returns the formatted string):

```
> let name, date = "Tomas", DateTime.Now;;
> let s = String.Format("Hello {0}! Today is: {1:D}", name, date);;
val s : string = "Hello Tomas! Today is: Sunday, 15 March 2009"
```

(continued)

The *format string* is specified after the colon; for example, `{0:D}` for a date formatted using the long date format, `{0:e}` for the scientific floating point, or `{0:x}` for a hexadecimal integer. In the last listing, we also specified alignment and padding of the printed value, which is done by adding a number after the comma:

```
> Console.WriteLine("Number with spaces: {0,10}!", 42);;
Number with spaces:       42!
> Console.WriteLine("Number with spaces: {0,-10}!", 42);;
Number with spaces: 42      !
```

Aside from the specification of alignment and padding, the .NET libraries are frequently used from F# when formatting standard .NET data types (such as the `DateTime` type or the `DateTimeOffset` type, which represents the time relatively to the UTC time zone). The following example briefly recapitulates some of the useful formatting strings. Note that the output is culture-sensitive, so it can vary depending on your system settings:

```
> let date = DateTimeOffset.Now;;
val date : DateTimeOffset = 03/15/2009 16:37:53 +00:00
> String.Format("{0:D}", date);;
val it : string = "Sunday, 15 March 2009"
> String.Format("{0:T}", date);;
val it : string = "16:36:09"
> String.Format("{0:yyyy-MM-dd}", date);;
val it : string = "2009-03-15"
```

The F#-specific functions for formatting strings are treated specially by the compiler, which has the benefit that it can check that we're working correctly with types. Just like in .NET formatting, we specify the format as a first argument, but the placeholders in the format specify the type of the argument. There's no index, so placeholders have to be in the same order as the arguments. In F#, you'll often work with `printf` and `printfn` to output the string to the console (`printfn` adds a line break) and `sprintf`, which returns a formatted string:

```
printfn "Hello %s! Today is: %A" name date
let s = sprintf "Hello %s! Today is: %A" name date
```

The following list shows the most common types of placeholders:

- `%s`—The argument is of type `string`.
- `%d`—Any signed or unsigned integer type (e.g., `byte`, `int`, or `ulong`)
- `%f`—A floating-point number of type `float` or `float32`
- `%A`—Outputs the value of any type using a combination of F# reflection and .NET `ToString` method. This prints the most readable debug information about any value.

Choosing between the .NET and F# approach is sometimes difficult. In general, it's usually better to use the F# function, because it has been designed to work well with F# and checks the types of arguments based on the format string. If you need functionality that isn't available or is hard to achieve using F# functions, you can switch to .NET formatting methods, because both can be easily used in F#.

Instead of running everything in F# Interactive, we could turn the code from listing 4.5 into a standard console application. If you're writing the code in Visual Studio and executing it in F# Interactive by pressing Alt+Enter, you already have the complete source code for the application. The only change that will make it more useful is the ability to read the filename from the command line. In F#, we can read command-line arguments using the standard .NET `Environment.GetCommandLineArgs` method. Alternatively, you can write an entry-point function that takes a string array as an argument and mark it using the `EntryPoint` attribute. The first element of the array is the name of the running executable, so to read the first argument, we can write `args.[1]`.

In this section, we added a simple console-based output for our data processing application. Now it's time to implement the GUI using the Windows Forms library and then draw the pie chart using classes from the `System.Drawing` namespace. Thanks to our earlier experiments and the use of F# Interactive during the development, we already know that a significant part of our code works correctly. If we were to write the whole application from scratch, we'd quite possibly already have several minor, but hard-to-find, bugs in the code. Of course, in a later phase of the development process, we should turn these interactive experiments into unit tests. We'll talk about this topic in chapter 11.

4.4 Creating a Windows Forms application

Windows Forms is a standard library for developing GUI applications for Windows and is nicely integrated with functionality from the `System.Drawing` namespace. These two libraries allow us, among other things, to draw graphics and display them on the screen. The .NET ecosystem is quite rich, so we could use other technologies as well. We can use Windows Presentation Foundation (WPF), which is part of .NET 3.0, for creating visually attractive UIs that use animations, rich graphics, or even 3D visualizations.

4.4.1 Creating the user interface

For this chapter we're using Windows Forms, which is in many ways simpler, but using other technologies from F# shouldn't be a problem for you. The UI in Windows Forms is constructed using components (like `Form`, `Button`, or `PictureBox`), so we're going to start by writing code that builds the UI controls. This task can be simplified by using a graphical designer, but our application is quite simple, so we'll write the code by hand. In some UI frameworks (including WPF), the structure of controls can be described in an XML-based file, but in Windows Forms, we're going to construct the appropriate classes and configure them by specifying their properties.

Before we can start, we need to configure the project in Visual Studio. By default, the F# project doesn't contain references to the required .NET assemblies, so we need to add references to `System.Windows.Forms` and `System.Drawing`. We can do this using the Add Reference option in Solution Explorer. Also, we don't want to display the console window when the application starts. You can open the project properties and select the Windows Application option from the Output Type drop-down list.

After configuring the project, we can write the first part of the application, as shown in listing 4.6.

Listing 4.6 Building the user interface (F#)

```
open System
open System.Drawing
open System.Windows.Forms

let mainForm = new Form(Width = 620, Height = 450, Text = "Pie Chart")   ❶

let menu = new ToolStrip()
let btnOpen = new ToolStripButton("Open")
let btnSave = new ToolStripButton("Save", Enabled = false)
ignore(menu.Items.Add(btnOpen))
ignore(menu.Items.Add(btnSave))

let boxChart =
  new PictureBox
    (BackColor = Color.White, Dock = DockStyle.Fill,
     SizeMode = PictureBoxSizeMode.CenterImage)

mainForm.Controls.Add(menu)
mainForm.Controls.Add(boxChart)

// TODO: Drawing of the chart & user interface interactions

[<STAThread>]   ❷
do
  Application.Run(mainForm)
```

Constructs application menu

Constructs control to display pie chart

Starts application with main form

The listing starts by opening .NET namespaces that contain classes used in our program. Next, we start creating the controls that represent the UI. We start with constructing the main window (also called the *form*). We're using an F# syntax that allows us to specify properties of the object directly during the initialization ❶. This makes the code shorter, but also hides side effects in the code. Internally, the code first creates the object using a constructor and then sets the properties of the object specified using this syntax, but we can view it as single operation that creates the object. When creating the form, we're using a parameterless constructor, but it's possible to specify arguments to the constructor too. You can see this later in the code when we create `btnSave`, whose constructor takes a string as an argument. A similar syntax for creating objects is now available in C# 3.0 as well and has an interesting history on the .NET platform (for more information, see the sidebar "Constructing classes in F#, C# 3.0, and Cω").

When adding the toolbar buttons to the collection of menu items, we call the `Add` method, which returns an index of the added item. In C#, you can call this method and ignore the return value, but F# is stricter. In functional programming, return values are much more important, so it is usually a mistake to ignore them. For this reason, F# compiler reports a warning when we ignore a return value. Fixing the code is quite easy, we can wrap the call inside a call to the `ignore` function. The function takes any value as an argument and returns `unit` (representing no return value) so that the compiler stops complaining.

The listing continues by constructing the menu and `PictureBox` control, which we'll use for showing the pie chart. We're not using F# Interactive this time, so there's a placeholder in the listing marking the spot where we'll add code for drawing the charts and for connecting the drawing functionality to the UI.

The final part of listing 4.6 is a standard block of code for running Windows Forms applications ❷. It starts with a specification of threading model for COM technology, which is internally used by Windows Forms. This is specified using a standard .NET attribute (`STAThreadAttribute`) so you can find more information about it in the .NET reference. In C# we'd place this attribute before the `Main` method, but in F# the source can contain code to be executed in any place. Since we need to apply this attribute, we're using a `do` block, which groups together the code to be executed when the application starts.

Constructing classes in F#, C# 3.0, and Cω

We already mentioned that some GUI frameworks use XML to specify how the controls should be constructed. This is a common approach, because constructing objects and setting their properties is similar to constructing an XML node and setting its attributes. This similarity was a motivation for researchers working on a language Cω [Meijer, Schulte, and Bierman, 2003] in Microsoft Research in 2003, which later motivated many features that are now present in C# 3.0. In Cω, we could write a code to construct `ToolStripButton` control like this:

```
ToolStripButton btn = <ToolStripButton>
                        <Text>Save</Text>
                        <Enabled>True</Enabled>
                        <Image>{saveIco}</Image>
                      </ToolStripButton>
```

In Cω, the XML syntax was integrated directly in the language. The elements nested in the `ToolStripButton` node specify properties of the object, and the syntax using curly braces allows us to embed usual non-XML expressions in the XML-like code. The ease of constructing objects in this way probably inspired the designers of XAML, which is an XML-based language used in WPF for describing UIs. On the language side, it motivated C# 3.0 feature called *object initializers*:

```
var btn = new ToolStripButton("Save"){ Enabled = false, Image = saveIco };
```

It no longer uses XML-based syntax, but the general idea to construct the object and specify its properties is essentially the same. We can also specify arguments of the constructor using this syntax, because the properties are specified separately in curly braces. Listing 4.6 shows that the same feature is available in F# as well:

```
let btn = new ToolStripButton("Save", Enabled = false, Image = saveIco)
```

The only difference from C# 3.0 is that in F# we specify properties directly in the constructor call. The arguments of the constructor are followed by a set of key-value pairs specifying the properties of the object.

(continued)

Another way to parameterize construction of a class, but also any ordinary method call, is to use named arguments. The key difference is that names of the parameters are part of the constructor or method declaration. Named parameters can also be used to initialize immutable classes, because they don't rely on setting a property after the class is created. This feature is available in F#, and you can find more information in the F# documentation. In C#, named arguments are being introduced in version 4.0 and the syntax is similar to specification of properties in F#. However, it's important to keep in mind that the meaning is quite different.

So far, we've implemented a skeleton of the application, but it doesn't *do* anything yet—at least, it doesn't do anything with our data. In the next section, we're going to fill in the missing part of the code to draw the chart and display it in the existing `PictureBox` called `boxChart`.

4.4.2 Drawing graphics

The application will draw the pie chart in two steps: it will draw the filled pie and it will add the text labels. This way, we can be sure that the labels are never covered by the pie.

A large part of the code that performs the drawing can be shared by both steps. For each step, we need to iterate over all items in the list to calculate the angle occupied by the segment of the pie chart. The solution to this problem is to write a function that performs the shared operations and takes a drawing function as an argument. The code calls this function twice. The drawing function in the first step fills segments of the pie chart, and the one in the second step draws the text label.

CREATING RANDOM COLOR BRUSHES

Let's start by drawing the pie. We want to fill specified segments of the pie chart using random colors, so first we'll write a simple utility function that creates a randomly colored brush that we can use for filling the region, as shown in listing 4.7.

Listing 4.7 Creating brush with random color (F#)

```
let rnd = new Random()
let randomBrush() =
  let r, g, b = rnd.Next(256), rnd.Next(256), rnd.Next(256)
  new SolidBrush(Color.FromArgb(r,g,b))
```

The code declares two top-level values. The first is an instance of a .NET class `Random`, which is used for generating random numbers. The second is a function `randomBrush`. It has a `unit` type as a parameter, which is an F# way of saying that it doesn't take any meaningful arguments. Thanks to this parameter, we're declaring a function that can be run several times giving a different result. If we omitted it, we'd create a value that would be evaluated only once when the application starts. The only possible `unit`

value is `()`, so when calling the function later in the code, we're actually giving it `unit` as an argument, even though it looks like a function call with no arguments at all. The `randomBrush` function uses the `rnd` value and generates `SolidBrush` object, which can be used for filling of specified regions. It has side effects and as you already know, we should be careful when using side effects in functional programs.

Hiding the side effects

The function `randomBrush` is an example of a function with side effects. This means that the function may return a different result every time it's called, because it relies on some changing value, other than the function arguments. In this example, the changing value is the value `rnd`, which represents a random number generator and changes its internal state after each call to the `Next` method. Listing 4.7 declares `rnd` as a global value despite the fact that it's used only in the function `randomBrush`. Of course, this is a hint that we should declare it locally to minimize the number of global values. We could try rewriting the code as follows:

```
let randomBrush() =
  let rnd = new Random()
  let r, g, b = rnd.Next(256), rnd.Next(256), rnd.Next(256)
  new SolidBrush(Color.FromArgb(r,g,b))
```

But this code doesn't work! The problem is that we're creating a new `Random` object every time the function is called and the change of the internal state isn't preserved. When created, `Random` initializes the internal state using the current time, but since the drawing is performed quickly the "current time" doesn't change enough and we end up with the whole chart being drawn in the same color.

Not surprisingly, there's a way to write the code without declaring `rnd` as a global value, but that allows us to keep the mutable state represented by it between the function calls. To write this, we need two concepts that will be discussed in chapter 5: a closure and a lambda function. We'll see a similar example showing a frequent pattern for hiding side effects like this one in chapter 8.

Now that you know how to create brushes for filling the chart, we can take a look at the first of the drawing functions.

DRAWING THE PIE CHART SEGMENTS

Listing 4.8 implements a function called `drawPieSegment`. It fills the specified segment of the chart using a random color. This function will be used from a function that performs the drawing in two phases later in the application. The processing function will call it for every segment, and it will get all the information it needs as arguments.

Listing 4.8 Drawing a segment of the pie chart (F#)

```
let drawPieSegment(gr:Graphics, title, startAngle, occupiedAngle) =
  let br = randomBrush()
  gr.FillPie
```

```
    (br, 170, 70, 260, 260,
     startAngle, occupiedAngle)
  br.Dispose()
```

Specifies center and size of pie

The function parameters are written as one big tuple containing four elements, because this helps to make the code more readable. The first argument of the function is written with a type annotation specifying that its type is `Graphics`. This is a `System.Drawing` class, which contains functionality for drawing. We use its `FillPie` method within the function, but that's all that the compiler can tell about the `gr` value. It can't infer the type from that information, which is why we need the type annotation. The next three tuple elements specify the title text (which isn't used anywhere in the code but will be important for drawing labels), the starting angle of the segment, and the total angle occupied by the segment (in degrees). Note that we also dispose of the brush once the drawing is finished. F# has a nicer way to do this, and we'll talk about it in chapter 9.

Choosing a syntax when writing functions

We've seen two ways for writing functions with multiple arguments so far: we can write the function arguments either as a comma-separated list in parentheses or as a list of values separated by spaces. Note that the first style isn't really special in any way:

```
let add(a, b) = a + b
```

This is a function that takes a tuple as an argument. The expression `(a, b)` is the usual pattern, which we used for deconstructing tuples in chapter 3. The question is which option is better. Unfortunately there isn't an authoritative answer and this is a personal choice. The only important thing is to use the choice consistently.

In this book, we'll usually write function arguments using tuples, especially when writing some more complicated utility functions that work with .NET libraries. This will keep the code consistent with the syntax you use when calling .NET methods. We'll use spaces when writing simple utility functions that deal primarily with F# values.

We'll write parentheses when calling or declaring a function that takes a single argument, so for example we'll write `sin(x)` even though parentheses are optional and we could write `sin x`. This decision follows the way functions are usually written in mathematics and also when calling .NET methods with multiple arguments. We'll get back to this topic in chapters 5 and 6, when we discuss functions in more detail and also look at implementing and using higher-order functions.

The `drawPieSegment` function from the previous listing is one of the two drawing functions that we'll use as an argument to the function `drawStep`, which iterates over all the segments of the pie chart and draws them. Before looking at the code for `drawStep`, let's look at its type. Even though we don't need to write the types in the code, it's useful to see the types of values used in the code.

DRAWING USING FUNCTIONS

The first argument to the `drawStep` function is one of the two drawing functions, so we'll use a name `DrawingFunc` for the type of drawing functions for now and define what it is later. Before discussing the remaining arguments, let's look at the signature of the function:

```
drawStep : (DrawingFunc * Graphics * float * (string * int) list) -> unit
```

We're again using the tuple syntax to specify the arguments, so the function takes a single big tuple. The second argument is the `Graphics` object for drawing, which will be passed to the drawing function. The next two arguments specify the data set used for the drawing—a `float` value is the sum of all the numeric values, so we can calculate the angle for each segment, and a value of type `(string * int) list` is our familiar data set from the console version of the application. It stores the labels and values for each item to be plotted.

Let's look at the `DrawingFunc` type. It should be same as the signature of the `drawPieSegment` function from listing 4.8. The second drawing function is `drawLabel`, which we'll see shortly has exactly the same signature. We can look at the signatures and declare the `DrawingFunc` type to be exactly the same type as the types of these two functions:

```
drawPieSegment : (Graphics * string * int * int) -> unit
drawLabel      : (Graphics * string * int * int) -> unit

type DrawingFunc = (Graphics * string * int * int) -> unit
```

The last line is a type declaration that declares a *type alias*. This means that we're assigning a name to a complicated type that could be written in some other way. We're using the `DrawingFunc` name only in this explanation, but we could use it, for example, in a type annotation if we wanted to guide the type inference or make the code more readable.

As I mentioned earlier, we don't need to write these types in the code, but it will help us understand what the code does. The most important thing that we already know is that the `drawStep` function takes a drawing function as a first argument. Listing 4.9 shows the code of the `drawStep` function.

Listing 4.9 Drawing items using specified drawing function (F#)

```
let drawStep(drawingFunc, gr:Graphics, sum, data) =
  let rec drawStepUtil(data, angleSoFar) =                    ❶
    match data with
    | [] -> ()                                                ❷
    | [title, value] ->                                       ❸
      let angle = 360 - angleSoFar
      drawingFunc(gr, title, angleSoFar, angle)
    | (title, value)::tail ->                                 ❹
      let angle = int(float(value) / sum * 360.0)
      drawingFunc(gr, title, angleSoFar, angle)
      drawStepUtil(tail, angleSoFar + angle)
  drawStepUtil(data, 0)
```

Calculates angle to add up to 360

Recursively draws the rest

Runs utility function

To make the code more readable, we implement the function that does the actual work as a nested function ❶. It iterates over all items that should be drawn on the chart. The items are stored in a standard F# list, so the code is quite like the familiar list processing pattern. There is one notable difference, because the list is matched against three patterns instead of the usual two cases matching an empty list and a cons cell.

The first branch in the pattern matching ❷ matches an empty list and doesn't do anything. As we've already seen, "doing nothing" is in F# expressed as a `unit` value, so the code returns a `unit` value, written as `()`. This is because F# treats every construct as an expression and expressions always have to return a value. If the branch for the empty list were empty, it wouldn't be a valid expression.

The second branch ❸ is what makes the list processing code unusual. As you can see, the pattern used in this branch is `[title, value]`. This is a nested pattern composed from a pattern that matches a list containing a single item `[it]` and a pattern that matches the item with a tuple containing two elements: `(title, value)`. The syntax we're using is shorthand for `[(title, value)]`, but it means the same thing. The first pattern is written using the usual syntax for creating lists, so if you wanted to write a pattern to match lists with three items, you could write `[a; b; c]`. We included this special case, because we want to correct the rounding error: if we're processing the last item in the list, we want to make sure that the total angle will be exactly 360 degrees. In this branch we simply calculate the angle and call the `drawingFunc` function, which was passed to us as an argument.

The last branch processes a list that didn't match any of the previous two patterns. The order of the patterns is important in this case, because any list matching the second pattern ❸ would also match the last one ❹ but with an empty list as the tail. The order of the patterns in the code guarantees that the last branch won't be called for the last item.

The code for the last branch calculates the angle and draws the segment using the specified drawing function. This is the only branch that doesn't stop the recursive processing of the list, because it's used until there's a last element in the list, so the last line of the code is a recursive call. The only arguments that change during the recursion are the list of remaining elements to draw and the `angleSoFar`, which is an angle occupied by all the already processed segments. Thanks to the use of local function, we don't need to pass along the other arguments that don't change. Only one thing is done in the `drawStep` function itself: it invokes the utility function with all the data and the argument `angleSoFar` set to 0.

DRAWING THE WHOLE CHART

Before looking at the second drawing function, let's see how to put things together. Figure 4.2 shows each of the layers separately: the code that we've already written draws the left part of the figure; we still need to implement the function to draw the labels shown on the right part.

The code that draws the chart first loads data from a file, then processes it is the same as in the console application. Instead of printing data to the console, we now use

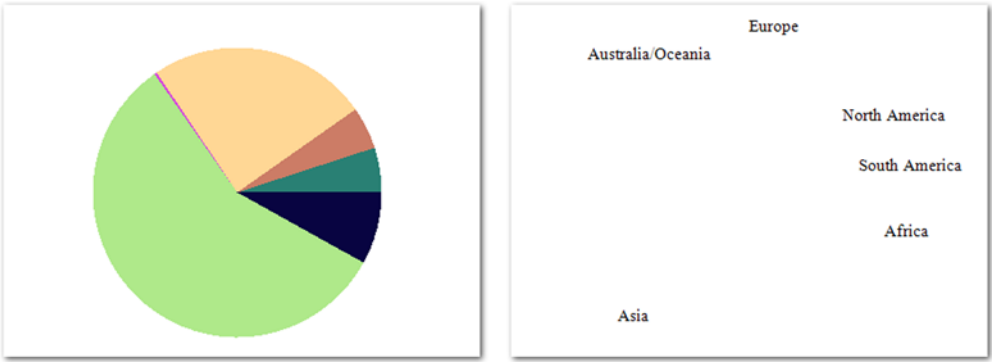

Figure 4.2 Two phases of drawing the chart: the first phase using `drawPieSegment` (left) and the second using the `drawLabel` function (right). The chart shows distribution of the world population in 1900.

the functions described earlier to draw the chart. You can see the function `drawChart` that does the drawing in listing 4.10.

Listing 4.10 Drawing the chart (F#)

```
let drawChart(file) =
  let lines = List.ofSeq(File.ReadAllLines(file))
  let data = processLines(lines)
  let sum = float(calculateSum(data))

  let pieChart = new Bitmap(600, 400)
  let gr = Graphics.FromImage(pieChart)
  gr.Clear(Color.White)
  drawStep(drawPieSegment, gr, sum, data)
  drawStep(drawLabel, gr, sum, data)

  gr.Dispose()
  pieChart
```

Loads, processes data

Creates bitmap and object for drawing

❶

❷

Finalizes drawing

The function takes a name of the CSV file as an argument and returns an in-memory bitmap with the pie chart. In the code, we first load the file and process it using our existing `processLines` and `calculateSum` functions. We then draw the chart, and on the last line we return the created bitmap as a result of the function.

To draw anything at all, we first need to create a `Bitmap` object and then an associated `Graphics` object. We've used `Graphics` for drawing in all the previous functions, so once it's created we can fill the bitmap with a white background and draw the chart using the `drawStep` function. The first call ❶ draws the pie using `drawPieSegment`, and the second call ❷ draws the text labels using `drawLabel`. You can try commenting out one of these two lines to draw only one of the steps and get the same results shown in figure 4.2. We haven't implemented the `drawLabel` function yet, because we wanted to show how the whole drawing works first, but now we're ready to finish this part of the application.

ADDING TEXT LABELS

We've already implemented the first drawing function and the second one should have the same signature, so that we can use each of them as an argument to the universal `drawStep` function. The only thing that we have to fill in is the code for drawing the label and calculating its position, as you can see in listing 4.11.

Listing 4.11 Drawing text labels (F#)

```
let fnt = new Font("Times New Roman", 11.0f)

let centerX, centerY = 300.0, 200.0
let labelDistance = 150.0

let drawLabel(gr:Graphics, title, startAngle, angle) =
  let lblAngle = float(startAngle + angle/2)                    ❶
  let ra = Math.PI * 2.0 * lblAngle / 360.0                     ❷
  let x = centerX + labelDistance * cos(ra)
  let y = centerY + labelDistance * sin(ra)
  let size = gr.MeasureString(lbl, fnt)
  let rc = new PointF(float32(x) - size.Width / 2.0f,
                      float32(y) - size.Height / 2.0f)
  gr.DrawString(title, fnt, Brushes.Black, new RectangleF(rc, size))
```

Defines properties of pie chart

Gets bounding box, draws label

We first declare a top-level font value used for drawing the text. We do this because we don't want to initialize a new instance of the font every time the function is called. Since the font will be needed during the whole lifetime of the application, we don't dispose of it explicitly; we rely on .NET to dispose of it when the application quits. The function itself starts with several lines of code that calculate location of the label.

The first line ❶ calculates the angle in degrees that specifies the center of the pie chart sector occupied by the segment. We take the starting angle of the segment and add half of the segment size to move the label to the center. The second line ❷ converts the angle to radians. Once we have the angle in radians, we can compute the X and Y coordinates of the label using trigonometric functions `cos` and `sin`. We use `MeasureString` method to estimate the size of the text label and calculate the location of the bounding box in which the text is drawn. The X and Y coordinates calculated earlier are used as a center of the bounding box.

Now that we've finished the code for drawing text labels, we're done with the whole code for drawing the pie chart. We implemented the key function (`drawChart`), which performs the drawing of the chart earlier in listing 4.10. The function takes a filename of the CSV file as an argument and returns a bitmap with the chart. All we have to do now is add code that will call this function from our UI.

4.4.3 Creating the Windows application

We started creating the GUI of the application earlier, so we already have code to create UI controls. However we still have to specify user interaction logic for our controls.

The user can control the application using two buttons. The first one (`btnOpen`) loads a CSV file, and the second one (`btnSave`) saves the chart into an image file. We

also have a `PictureBox` control called `boxChart`, which is where we'll show the chart. Listing 4.12 shows how to connect the drawing code with our UI.

Listing 4.12 Adding user interaction (F#)

```
let openAndDrawChart(e) =                                          ❶
  let dlg = new OpenFileDialog(Filter="CSV Files|*.csv")
  if (dlg.ShowDialog() = DialogResult.OK) then
    let pieChart = drawChart(dlg.FileName)                         ❷
    boxChart.Image <- pieChart                                     Displays bitmap
    btnSave.Enabled <- true                                        Enables button for saving image

let saveDrawing(e) =                                               ❸
   let dlg = new SaveFileDialog(Filter="PNG Files|*.png")
   if (dlg.ShowDialog() = DialogResult.OK) then
      boxChart.Image.Save(dlg.FileName)                            Saves current chart

[<STAThread>]
do                                                                 ❹
  btnOpen.Click.Add(openAndDrawChart)                              Registers event handlers
  btnSave.Click.Add(saveDrawing)
  Application.Run(mainForm)
```

The code first declares two functions that will be invoked when the user clicks the Open and Save buttons, respectively. For opening a file, we have a function `openAndDrawChart` ❶. The function first creates an `OpenFileDialog`, which is a Windows Forms class that shows standard dialog for selecting a file. If the user selects a file, the function calls `drawChart` ❷, which we implemented earlier. A result of this call is an in-memory bitmap, which can be assigned to the `Image` property of the `PictureBox` control. The second function is simpler, because it doesn't need to draw the chart. It saves the image currently displayed in the `PictureBox` to a file, which is specified by the user using `SaveFileDialog` ❸.

We've already talked about the code to execute a standard Windows application, but listing 4.12 shows it again ❹ because we've added two lines of code. Before running the application, we specify that the `openAndDrawChart` function should be called when the user clicks the `btnOpen` button and likewise for the second button. This is done by registering a function as a handler of the `Click` event using the `Add` method. Unlike in C#, where events are special language constructs, F# treats events as normal objects that have an `Add` method. Events in F# also have `AddHandler` and `RemoveHandler` methods that serve exactly the same purpose as `+=` and `-=` operators for events in C#. We'll talk about this topic in more detail in chapter 16, but in most of the cases you can use the `Add` method.

4.5 Summary

In this chapter we developed a simple but real-world application for drawing pie charts. We discussed basic F# and .NET numeric data types and explored both F# and .NET functionality for working with strings. We also demonstrated how to use usual

.NET libraries from F#, and you saw examples using Windows Forms, `System.Drawing` as well as basic I/O.

What we wanted to demonstrate in this chapter was a typical F# development process. In the beginning we started writing functions for working with the data, and we immediately tested them in F# Interactive. As we progressed, we implemented a function to load the real data from a file and a simple console application to verify that the core functions work correctly. Finally, we added a GUI and drew the chart using the functionality that we'd already implemented and tested.

We were able to implement the application in this way so early in the book mainly because it doesn't work with data extensively. The only data structures that we've used are tuples and lists, which were both introduced in chapter 2. Most real-world applications need to work with more complex data sets. This is a topic for part 2, where we'll see how to represent more complicated and structured data in a functional way and how to process it elegantly.

Of course, the application is still quite simple and extending it (for example, by adding different types of charts) would be difficult at this point. To make the application more extensible, we need to perform one more iteration in our development approach. This requires many of the advanced functional techniques discussed in the rest of the book.

PROGRAMMING

Real-World Functional Programming

Tomas Petricek with Jon Skeet Foreword by Mads Torgersen

Functional programming languages are good at expressing complex ideas in a succinct, declarative way. Functional concepts such as "immutability" and "function values" make it easier to reason about code—as well as helping with concurrency. The new F# language, LINQ, certain new features of C#, and numerous .NET libraries now bring the power of functional programming to .NET coders.

This book teaches the ideas and techniques of functional programming applied to real-world problems. You'll see how the functional way of thinking changes the game for .NET developers. Then, you'll tackle common issues using a functional approach. The book will also teach you the basics of the F# language and extend your C# skills into the functional domain. No prior experience with functional programming or F# is required.

What's Inside

- Thinking the functional way
- Blending OO and functional programming
- Effective F# code

Microsoft C# MVP **Tomas Petricek** is one of the leaders of the F# community. He was part of the Microsoft Research team for F# and is interested in distributed and reactive programming using F#. Microsoft C# MVP **Jon Skeet** is a veteran C# and Java developer, prolific "Stack Overflow" contributor, and author of *C# in Depth*.

For online access to the authors, and a free ebook for owners of this book, go to manning.com/Real-WorldFunctionalProgramming

"You will never look at your code in the same way again!"
—From the Foreword by Mads Torgersen, C# PM, Microsoft

"A truly functional book!"
—Andrew Siemer, .NET Architect

".NET needs more functional programmers...this book shows you how to become one."
—Stuart Caborn, Lead Consultant Thoughtworks

"Warning: this book has a very high Wow! factor. It made my head hurt...in a good way!"
—Mark Seemann Developer/Architect, Safewhere

"I recommend it to all software craftspeople, not just .NET developers."
—Paul King, Director, ASERT

ISBN 13: 978-1-933988-92-4
ISBN 10: 1-933988-92-4
54999
9 781933 988924

MANNING

$49.99 / Can $62.99 [INCLUDING eBOOK]