Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Functional Programming in C#

Classic Programming Techniques for Modern Projects

Oliver Sturm

PROFESSIONAL FUNCTIONAL PROGRAMMING IN C#

INTRODUCTION xiii

▶ **PART I INTRODUCTION TO FUNCTIONAL PROGRAMMING**

CHAPTER 1 A Look at Functional Programming History 3

CHAPTER 2 Putting Functional Programming into a Modern Context 9

▶ **PART II C# FOUNDATIONS OF FUNCTIONAL PROGRAMMING**

CHAPTER 3 Functions, Delegates, and Lambda Expressions 17

CHAPTER 4 Flexible Typing with Generics 31

CHAPTER 5 Lazy Listing with Iterators 43

CHAPTER 6 Encapsulating Data in Closures 55

CHAPTER 7 Code Is Data 61

▶ **PART III IMPLEMENTING WELL-KNOWN FUNCTIONAL TECHNIQUES IN C#**

CHAPTER 8 Currying and Partial Application 77

CHAPTER 9 Lazy Evaluation 91

CHAPTER 10 Caching Techniques 101

CHAPTER 11 Calling Yourself 117

CHAPTER 12 Standard Higher Order Functions 131

CHAPTER 13 Sequences 141

CHAPTER 14 Constructing Functions from Functions 149

CHAPTER 15 Optional Values 159

CHAPTER 16 Keeping Data from Changing 167

CHAPTER 17 Monads 193

Continues

▶ **PART IV PUTTING FUNCTIONAL PROGRAMMING INTO ACTION**

CHAPTER 18 Integrating Functional Programming Approaches 209

CHAPTER 19 The MapReduce Pattern 233

CHAPTER 20 Applied Functional Modularization 241

CHAPTER 21 Existing Projects Using Functional Techniques 247

INDEX 261

PROFESSIONAL

Functional Programming in C#

CLASSIC PROGRAMMING TECHNIQUES FOR MODERN PROJECTS

Oliver Sturm

Wiley Publishing, Inc.

Professional Functional Programming in C#: Classic Programming Techniques for Modern Projects

This edition first published 2011

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

978-0-470-74458-1
978-0-470-97028-7 (ebk)
978-0-470-97110-9 (ebk)
978-0-470-97109-3 (ebk)

A catalogue record for this book is available from the British Library.

ABOUT THE AUTHOR

OLIVER STURM has over 20 years experience developing professional software. He is a well-known expert in various areas of application architecture, programming languages and the third party .NET tools made by DevExpress. His main focus has been on the .NET platform since 2002. Oliver has spoken at many international conferences and has written more than 20 training classes and more than 100 magazine articles in English as well as German. He has also taught classes on topics around computer programming for more than 15 years. For his contributions to the .NET community, he has been awarded the C# MVP Award by Microsoft United Kingdom for several years now.

Based in Scotland, UK, Oliver works as a freelance consultant and trainer, and he is an associate at thinktecture, an international consultancy firm. You can find his blog at `www.sturmnet.org/blog` and his commercial website at `www.oliversturm.com`. His e-mail address is `oliver@oliversturm.com`.

CREDITS

VP CONSUMER AND TECHNOLOGY PUBLISHING DIRECTOR
Michelle Leete

ASSOCIATE DIRECTOR—BOOK CONTENT MANAGEMENT
Martin Tribe

ASSOCIATE PUBLISHER
Chris Webb

PUBLISHING ASSISTANT
Ellie Scott

ASSOCIATE MARKETING DIRECTOR
Louise Breinholt

MARKETING EXECUTIVE
Kate Parrett

EDITORIAL MANAGER
Jodi Jensen

SENIOR PROJECT EDITOR
Sara Shlaer

PROJECT EDITOR
Brian Herrmann

TECHNICAL EDITORS
Michael Giagnocavo
Matthew Podwysocki

SENIOR PRODUCTION EDITOR
Debra Banninger

COPY EDITOR
Maryann Steinhart

PROOFREADER
Sheilah Ledwidge, Word One

INDEXER
Robert Swanson

COVER DESIGNER
Mike Trent

COVER IMAGE
© Oleg Kruglov/istockphoto.com

CONTENTS

INTRODUCTION *xiii*

PART I: INTRODUCTION TO FUNCTIONAL PROGRAMMING

CHAPTER 1: A LOOK AT FUNCTIONAL PROGRAMMING HISTORY **3**

What Is Functional Programming? 3
Functional Languages 5
The Relationship to Object Oriented Programming 7
Summary 8

CHAPTER 2: PUTTING FUNCTIONAL PROGRAMMING INTO A MODERN CONTEXT **9**

Managing Side Effects 10
Agile Programming Methodologies 11
Declarative Programming 11
Functional Programming Is a Mindset 12
Is Functional Programming in C# a Good Idea? 13
Summary 13

PART II: C# FOUNDATIONS OF FUNCTIONAL PROGRAMMING

CHAPTER 3: FUNCTIONS, DELEGATES, AND LAMBDA EXPRESSIONS **17**

Functions and Methods 17
Reusing Functions 19
Anonymous Functions and Lambda Expressions 23
Extension Methods 26
Referential Transparency 28
Summary 30

CHAPTER 4: FLEXIBLE TYPING WITH GENERICS **31**

Generic Functions 32
Generic Classes 34
Constraining Types 35
Other Generic Types 37

Covariance and Contravariance 38
Summary 41

CHAPTER 5: LAZY LISTING WITH ITERATORS 43

The Meaning of Laziness 43
Enumerating Things with .NET 44
Implementing Iterator Functions 47
Returning IEnumerator 50
Chaining Iterators 51
Summary 53

CHAPTER 6: ENCAPSULATING DATA IN CLOSURES 55

Constructing Functions Dynamically 55
The Problem with Scope 56
How Closures Work 57
Summary 60

CHAPTER 7: CODE IS DATA 61

Expression Trees in .NET 63
Analyzing Expressions 64
Generating Expressions 68
.NET 4.0 Specifics 72
Summary 74

PART III: IMPLEMENTING WELL-KNOWN FUNCTIONAL TECHNIQUES IN C#

CHAPTER 8: CURRYING AND PARTIAL APPLICATION 77

Decoupling Parameters 77
Manual Currying 78
Automatic Currying 80
Calling Curried Functions 82
The Class Context 82
What FCSlib Contains 85
Calling Parts of Functions 86
Why Parameter Order Matters 88
Summary 89

CHAPTER 9: LAZY EVALUATION 91

What's Good about Being Lazy? 92
Passing Functions 93
Explicit Lazy Evaluation 95
Comparing the Lazy Evaluation Techniques 98
Usability 98
Efficiency 98
How Lazy Can You Be? 99
Summary 100

CHAPTER 10: CACHING TECHNIQUES 101

The Need to Remember 101
Precomputation 102
Memoization 107
Deep Memoization 110
Considerations on Memoization 114
Summary 115

CHAPTER 11: CALLING YOURSELF 117

Recursion in C# 117
Tail Recursion 119
Accumulator Passing Style 121
Continuation Passing Style 122
Indirect Recursion 126
Summary 129

CHAPTER 12: STANDARD HIGHER ORDER FUNCTIONS 131

Applying Operations: Map 132
Using Criteria: Filter 132
Accumulating: Fold 133
Map, Filter, and Fold in LINQ 138
Standard Higher Order Functions 140
Summary 140

CHAPTER 13: SEQUENCES 141

Understanding List Comprehensions 141
A Functional Approach to Iterators 142
Ranges 143
Restrictions 146
Summary 147

CHAPTER 14: CONSTRUCTING FUNCTIONS FROM FUNCTIONS 149

Composing Functions 149
Advanced Partial Application 152
Combining Approaches 155
Summary 158

CHAPTER 15: OPTIONAL VALUES 159

The Meaning of Nothing 159
Implementing Option(al) Values 160
Summary 165

CHAPTER 16: KEEPING DATA FROM CHANGING 167

Change Is Good — Not! 167
False Assumptions 168
Being Static Is Good 169
A Matter of Depth 170
Cloning 171
Automatic Cloning 173
Implementing Immutable Container Data Structures 177
Linked List 177
Queue 183
Unbalanced Binary Tree 185
Red/Black Tree 187
Alternatives to Persistent Data Types 190
Summary 191

CHAPTER 17: MONADS 193

What's in a Typeclass? 194
What's in a Monad? 197
Why Do a Whole Abstraction? 198

A Second Monad: Logging **201**
Syntactic Sugar **203**
Binding with SelectMany? **204**
Summary **205**

PART IV: PUTTING FUNCTIONAL PROGRAMMING INTO ACTION

CHAPTER 18: INTEGRATING FUNCTIONAL PROGRAMMING APPROACHES **209**

Refactoring **210**
List Filtering with a Windows Forms UI 211
Calculating Mandelbrot Fractals 216
Writing New Code **224**
Use Static Methods 224
Prefer Anonymous Methods Over Named Ones 226
Prefer Higher Order Functions over Manual Algorithm Implementation 227
Prefer Immutable Data 228
Watch Behavior Implementation in Classes 229
Finding Likely Candidates for Functional Programming **229**
Shades of Grey 230
Using What's There 231
Summary **232**

CHAPTER 19: THE MAPREDUCE PATTERN **233**

Implementing MapReduce **234**
Abstracting the Problem **237**
Summary **240**

CHAPTER 20: APPLIED FUNCTIONAL MODULARIZATION **241**

Executing SQL Code from an Application **241**
Rewriting the Function with Partial Application and Precomputation in Mind **243**
Summary **245**

CHAPTER 21: EXISTING PROJECTS USING FUNCTIONAL TECHNIQUES **247**

The .NET Framework **247**
LINQ **249**
LINQ to Objects 249
LINQ to a Query Backend 253
Parallelization 255
Google MapReduce and Its Implementations **257**
NUnit **258**
Summary **260**

INDEX ***261***

INTRODUCTION

FUNCTIONAL PROGRAMMING is an important paradigm of programming that looks back on a long history. The subject has always been very relevant to people who teach others how to program — the clean and logical concepts of functional programming lend themselves especially well to teaching. Certain industries that use computers and self-written programs heavily have also found functional programming to be the most productive approach for their purposes. However, for many of the "mainstream" software manufacturers, functional programming has long held an air of the academic and they widely chose to use approaches with an imperative heritage, like object orientation.

In recent years, more and more functional elements have been included in imperative languages on the .NET platform, and with Visual Studio 2010, F# has been included — the first hybrid functional language in the box with Microsoft's mainstream development platform. Even more than the functional features that have been introduced to C# and VB.NET, this shows a commitment on Microsoft's side.

WHO THIS BOOK IS FOR

The topic of functional programming in C# can be seen from two different angles. On the .NET platform there are many experienced developers and development teams, who have been using C# or VB.NET, or in some cases C++, to create software for the platform. If you have that sort of experience, there are lots of reasons you should be looking into functional programming: it's a clean and easily maintainable style, it's an important basis of programming as we know it today, and certain specific current concerns, like parallelization, can be targeted successfully with the help of functional programming ideas.

On the other hand, perhaps you're not a .NET programmer at all. Instead, you have experience in one or more "traditional" functional programming languages. You need to work with people who use C#, or you want to use the language yourself. This book will help you understand how you can use the approaches you're familiar with in C#, and it may give you valuable starting points when it comes to explaining these ideas to team members without your functional background.

The book assumes a basic level of understanding of C# language constructs, at least up to version 3.0 of the language. However, Part II is written to explain a few particular features of the language that are especially important, rather complex or often misunderstood. From experience, I recommend you give Part II a good look even if you're quite fluent in C# — there are usually some little-known intricacies about the features that have been selected for this part, which may lead to misunderstandings later.

WHAT THIS BOOK COVERS

The language of the vast majority of examples in this book is C# 4.0, running on Microsoft .NET. There are a few examples in other languages, but they are for illustrative purposes only. If you want to try out the examples for yourself, but you're not on C# 4.0 or Visual Studio 2010 yet, you may still have success using C# 3.0 and Visual Studio 2008 — there aren't many new features in C# 4.0 specifically, and none of them have been exploited in the examples. However, a few examples utilize .NET Framework features like Parallel Extensions, which are available only in .NET 4.0.

The book introduces you to concepts of functional programming and describes how these can be used with the C# language. An effort has been made to provide samples with a practical background, but most of them still focus mostly on language level considerations. Functional programming is a technique for code, algorithm and program structure — as opposed to, for instance, application architecture. Of course it needs to fit in with application architecture . . . you get the point: it's sometimes hard to find the perfect compromise between being too theoretical and going off-focus, but I've tried my best.

While I wrote this book, I developed a library of functionally oriented helpers, called FCSlib (that's "Functional CSharp Library"). You can use this library in your own projects as you like, but please note that it doesn't come with any warranty. The downloadable file containing the library code (more information about downloads in the upcoming section "Source Code") includes a copy of the LGPL license text, which applies to the FCSlib code.

HOW THIS BOOK IS STRUCTURED

This book has four parts. The first part provides an overview of functional programming, both from a historical and a current point of view. Part II proceeds to give you the C# background you'll need to understand the more complex examples that follow later. Again, reading this is recommended even if you know C# — it does have a few pretty basic items, but generally it's not meant to be a language introduction for newbies.

Part III is the most important one. Its 10 chapters describe a variety of functional programming topics from a C# point of view, showing lots of examples and code snippets. The code library that accompanies this book, FCSlib, is built on the ideas described in this part.

Finally, Part IV gives you an overview of practical concerns of using functional programming in C#. I picked a few specific scenarios, and there are descriptions of functional programming ideas in existing products and technologies that you may be familiar with.

WHAT YOU NEED TO USE THIS BOOK

All code in this book has been tested with Visual Studio 2010, C# 4.0 and .NET 4.0. Much of it has been originally developed on C# 3.0, so you should have good success running the code on .NET 3.5. Going back further than that would mean major rewrites in many areas — the concepts

may translate even to C# 2.0 in many cases, but the language features that make them reasonably easy to use are just not available in that version.

I have made several attempts to build the code on the Mono platform, but unfortunately I stumbled upon compiler bugs every time. Your mileage may vary if you try to use Mono — after all, it changes all the time.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

The pencil icon indicates notes, tips, hints, tricks, and asides to the current discussion.

As for styles in the text:

- We *italicize* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show file names, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

```
We use a monofont type with no highlighting for most code examples.
We use bold to emphasize code that is particularly important in the present
context or to show changes from a previous code snippet.
```

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually, or to use the source code files that accompany the book. All the source code used in this book is available for download at `www.wrox.com`. When at the site, simply locate the book's title (use the Search box or one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book. Code that is included on the website is highlighted by the following icon:

Listings include the filename in the title. If it is just a code snippet, you'll find the filename in a code note such as this:

Code snippet filename

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-74458-1.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at `www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time, you will be helping us provide even higher quality information.

To find the errata page for this book, go to `www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page, you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list, including links to each book's errata, is also available at `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At p2p.wrox.com, you will find a number of different forums that will help you, not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

PART I

Introduction to Functional Programming

- **CHAPTER 1:** A Look at Functional Programming History
- **CHAPTER 2:** Putting Functional Programming into a Modern Context

1

A Look at Functional Programming History

WHAT'S IN THIS CHAPTER?

- An explanation functional programming
- A look at some functional languages
- The relationship to object oriented programming

Functional programming has been around for a very long time. Many regard the advent of the language LISP, in 1958, as the starting point of functional programming. On the other hand, LISP was based on existing concepts, perhaps most importantly those defined by Alonzo Church in his lambda calculus during the 1930s and 1940s. That sounds highly mathematical, and it was — the ideas of mathematics were easy to model in LISP, which made it the obvious language of choice in the academic sector. LISP introduced many other concepts that are still important to programming languages today.

WHAT IS FUNCTIONAL PROGRAMMING?

In spite of the close coupling to LISP in its early days, functional programming is generally regarded a paradigm of programming that can be applied in many languages — even those that were not originally intended to be used with that paradigm. Like the name implies, it focuses on the application of functions. Functional programmers use functions as building blocks to create new functions — that's not to say that there are no other language elements available to them, but the function is the main construct that architecture is built from.

Referential transparency is an important idea in the realm of functional programming. A function that is referentially transparent returns values that depend only on the input parameters that are passed. This is in contrast to the basic ideas of imperative programming,

where program state often influences return values of functions. Both functional and imperative programming use the term *function*, but the mathematical meaning of the referentially transparent function is the one used in functional programming. Such functions are also referred to as pure functions, and are described as having no side effects.

It's often impossible to define whether a given programming language is a functional language or not. On the other hand, it is possible to find out the extent to which a language supports approaches commonly used in the functional programming paradigm — recursion, for example. Most programming languages generally support recursion in the sense that programmers can call into a particular function, procedure, or method from its own code. But if the compilers and/or runtime environments associated with the language use stack-based tracking of return addresses on jumps like many imperative languages do, and there are no optimizations generally available to help prevent stack overflow issues, then recursion may be severely restricted in its applications. In imperative languages, there are often specialized syntax structures to implement loops, and more advanced support for recursion is ignored by the language or compiler designers.

Higher order functions are also important in functional programming. Higher order functions are those that take other functions as parameters or return other functions as their results. Many programming languages have some support for this capability. Even C has a syntax to define a type of a function or, in C terms, to refer to the function through a function pointer. Obviously this enables C programmers to pass around such function pointers or to return them from other functions. Many C libraries contain functions, such as those for searching and sorting, that are implemented as higher order functions, taking the essential data-specific comparison functions as parameters. Then again, C doesn't have any support for anonymous functions — that is, functions created on-the-fly, in-line, like lambda expressions, or for related concepts such as closures.

Other examples of language capabilities that help define functional programming are explored in the following chapters in this book.

For some programmers, functional programming is a natural way of telling the computer what it should do, by describing the properties of a given problem in a concise language. You might have heard the saying that functional programming is more about telling computers what the problem is they should be solving, and not so much about specifying the precise steps of the solution. This saying is a result of the high level of abstraction that functional programming provides. Referential transparency means that the only responsibility of the programmer is the specification of functions to describe and solve a given set of problems. On the basis of that specification, the computer can then decide on the best evaluation order, potential parallelization opportunities, or even whether a certain function needs to be evaluated at all.

For some other programmers, functional programming is not the starting point. They come from a procedural, imperative, or perhaps object oriented background. There's much anecdotal evidence of such programmers analyzing their day-to-day problems, both the ones they are meant to solve by writing programs, and the ones they encounter while writing those programs, and gravitating toward solutions from the functional realm by themselves. The ideas of functional programming often provide very natural solutions, and the fact that you can arrive there from different directions reinforces that point.

FUNCTIONAL LANGUAGES

Functional programming is not language specific. However, certain languages have been around in that space for a long time, influencing the evolution of functional programming approaches just as much as they were themselves influenced by those approaches to begin with. The largest parts of this book contain examples only in C#, but it can be useful to have at least an impression of the languages that have been used traditionally for functional programming, or which have evolved since the early days with functional programming as a primary focus.

Here are two simple functions written in LISP:

```
(defun calcLine (ch col line maxp)
  (let
    ((tch (if (= col (- maxp line)) (cons ch nil) (cons 46 nil))))
    (if (= col maxp) tch (append (append tch (calcLine ch (+ col 1) line maxp)) tch))
    )
  )

(defun calcLines (line maxp)
  (let*
    ((ch (+ line (char-int #\A)))
      (l (append (calcLine ch 0 line maxp) (cons 10 nil)))
      )
    (if (= line maxp) l (append (append l (calcLines (+ line 1) maxp)) l))
    )
  )
```

The dialect used here is Common Lisp, one of the main dialects of LISP. It is not important to understand precisely what this code snippet does. A much more interesting aspect of the LISP family of dialects is the structure and the syntactic simplicity exhibited. Arguably, LISP's Scheme dialects enforce this notion further than Common Lisp, Scheme being an extremely simple language with very strong extensibility features. But the general ideas become clear immediately: a minimum of syntax, few keywords and operators, and obvious blocks. Many of the elements you may regard as keywords or other built-in structures — such as `defun` or `append` — are actually macros, functions, or procedures. They may indeed come out of the box with your LISP system of choice, but they are not compiler magic. You can write your own or replace the existing implementations. Many programmers do not agree that the exclusive use of standard round parentheses makes code more readable, but it is nevertheless easy to admire the elegance of such a basic system.

The following code snippet shows an implementation of the same two functions, the same algorithm, in the much newer language Haskell:

```
calcLine :: Int -> Int -> Int -> Int -> String
calcLine ch col line maxp =
  let tch = if maxp - line == col then [chr ch] else "." in
  if col == maxp
    then tch
```

```
    else tch ++ (calcLine ch (col+1) line maxp) ++ tch

calcLines :: Int -> Int -> String
calcLines line maxp =
  let ch = (ord 'A') + line in
  let l = (calcLine ch 0 line maxp) ++ "\n" in
  if line == maxp
    then l
    else l ++ (calcLines (line+1) maxp) ++ l
```

There is a very different style to the structure of the Haskell code. Different types of brackets are used to create a list comprehension. The `if...then...else` construct is a built-in, and the `++` operator does the job of appending lists. The type signatures of the functions are a common practice in Haskell, although they are not strictly required. One very important distinction can't readily be seen: Haskell is a strongly typed language, whereas LISP is dynamically typed. Because Haskell has extremely strong type inference, it is usually unnecessary to tell the compiler about types explicitly; they are known at compile time. There are many other invisible differences between Haskell and LISP, but that's not the focus of this book.

Finally, here's an example in the language Erlang, chosen for certain Erlang specific elements:

```
add(A, B) ->
    Calc = whereis(calcservice),
    Calc ! {self(), add, A, B},
    receive
        {Calc, Result} -> Result
    end.

mult(A, B) ->
    Calc = whereis(calcservice),
    Calc ! {self(), mult, A, B},
    receive
        {Calc, Result} -> Result
    end.


loop() ->
    receive
        {Sender, add, A, B} ->
            Result = A + B,
            io:format("adding: ~p~n", [Result]),
            Sender ! {self(), Result},
            loop();
        {Sender, mult, A, B} ->
            Result = A * B,
            io:format("multiplying: ~p~n", [Result]),
            Sender ! {self(), Result},
            loop();
        Other ->
            io:format("I don't know how to do ~p~n", [Other]),
            loop()
    end.
```

This is a very simple learning sample of Erlang code. However, it uses constructs pointing at the Actor model based parallelization support provided by the language and its runtime system. Erlang is not a very strict functional language — mixing in the types of side effects provided by `io:format` wouldn't be possible this way in Haskell. But in many industrial applications, Erlang has an important role today for its stability and the particular feature set it provides.

As you can see, functional languages, like imperative ones, can take many different shapes. From the very simplistic approach of LISP to the advanced syntax of Haskell or the specific feature set of Erlang, with many steps in between, there's a great spectrum of languages available to programmers who want to choose a language for its functional origins. All three language families are available today, with strong runtime systems, even for .NET in the case of the LISP dialect Clojure. Some of the ideas shown by those languages will be discussed further in the upcoming chapters.

THE RELATIONSHIP TO OBJECT ORIENTED PROGRAMMING

It is a common assumption that the ideas of functional programming are incompatible with those of other schools of programming. In reality, most languages available today are hybrid in the sense that they don't focus exclusively on one programming technique. There's no reason why they should, either, because different techniques can often complement one another.

Object oriented programming brings a number of interesting aspects to the table. One of them is a strong focus on encapsulation, combining data and behavior into classes and objects, and defining interfaces for their interaction. These ideas help object oriented languages promote modularization and a certain kind of reuse on the basis of the modules programmers create. An aspect that's responsible for the wide adoption object oriented programming languages have seen in mainstream programming is the way they allow modeling of real-world scenarios in computer programs. Many business application scenarios are focused on data storage, and the data in question is often related to physical items, which have properties and are often defined and distinguished by the way they interact with other items in their environments. As a result, object oriented mechanisms are not just widely applicable, but they are also easy to grasp.

When looking at a complicated industrial machine, for example, many programmers immediately come up with a way of modeling it in code as a collection of the wheels and cogs and other parts. Perhaps they consider viewing it as an abstract system that takes some raw materials and creates an end product. For certain applications, however, it may be interesting to deal with what the machine does on a rather abstract level. There may be measurements to read and analyze, and if the machine is complex enough, mathematical considerations might be behind the decisions for the parts to combine and the paths to take in the manufacturing process. This example can be abstractly extended toward any non-physical apparatus capable of generating output from input.

In reality, both the physical and the abstract viewpoints are important. Programming doesn't have a golden bullet, and programmers need to understand the different techniques at their disposal and make the decision for and against them on the basis of any problem with which they are confronted. Most programs have parts where data modeling is important, and they also have parts where algorithms are important. And of course they have many parts where there's no clear distinction,

where both data modeling and algorithms and a wide variety of other aspects are important. That's why so many modern programming languages are hybrid. This is not a new idea either — the first object oriented programming language standardized by ANSI was Common Lisp.

SUMMARY

Today's .NET platform provides one of the best possible constellations for hybrid software development. Originally a strong, modern and newly developed object oriented platform, .NET has taken major steps for years now in the functional direction. Microsoft F# is a fully supported hybrid language on the .NET platform, the development of which has influenced platform decisions since 2002. At the other end of the spectrum, albeit not all too far away, there's C#, a newly developed language strongly based in object orientation, that has been equally influenced by functional ideas almost from its invention. At the core of any program written in either language there's the .NET Framework itself, arguably the strongest set of underlying libraries that has ever been available for application development.

2

Putting Functional Programming into a Modern Context

WHAT'S IN THIS CHAPTER?

- Managing side effects
- Agile programming methodologies
- Declarative programming
- Functional programming as a mindset
- The feasibility of functional programming in C#

There have always been groups of programmers more interested in functional programming than in other schools of programming, and certain niches of the industry have provided a platform for those well versed in functional approaches and the underlying theory. At the same time, however, the mainstream of business application programming — the bread and butter of most programmers on platforms made by Microsoft and others — has evolved in a different direction. Object orientation and other forms of imperative programming have become the most widely used paradigms in this space of programming, to the extent that programmers have been neglecting other schools of thought more and more. For many, the realization that solutions to certain problems can be found by looking back to something "old" is initially a surprise.

One of the main reasons programmers become interested in functional programming today is the need for concurrency programming models. This need, in turn, comes from the evolution of the hardware toward multicore and multiprocessor setups. Programs no longer benefit very much from advances in technology like they did when increases in MHz were a main measurable reference point. Instead, programs need to be parallelized to take advantage of more than one CPU, or CPU core, available in a machine. Programmers are finding that