



Community Experience Distilled

F# for Quantitative Finance

An introductory guide to utilizing F# for quantitative finance
leveraging the .NET platform

Johan Astborg

[PACKT] open source*
PUBLISHING community experience distilled

F# for Quantitative Finance

An introductory guide to utilizing F# for quantitative finance leveraging the .NET platform

Johan Astborg



BIRMINGHAM - MUMBAI

F# for Quantitative Finance

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2013

Production Reference: 1191213

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-462-3

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author

Johan Astborg

Reviewers

Yan Cui

Arthur Pham

Isaac Abraham

Acquisition Editors

Sam Birch

Aarthi Kumaraswamy

Kunal Parikh

Lead Technical Editor

Athira Laji

Copy Editors

Roshni Banerjee

Janbal Dharmaraj

Mradula Hegde

Gladson Monteiro

Deepa Nambiar

Karuna Narayanan

Shambhavi Pai

Alfida Paiva

Adithi Shetty

Shambhavi Pai

Technical Editors

Gauri Dasgupta

Shiny Poojary

Siddhi Rane

Sonali S. Vernekar

Project Coordinator

Mary Alex

Proofreader

Paul Hindle

Indexers

Hemangini Bari

Mariammal Chettiyar

Tejal Soni

Graphics

Ronak Dhruv

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Author

Johan Astborg is the developer and architect of various kinds of software systems and applications, financial software systems, trading systems, as well as mobile and web applications. He is interested in computer science, mathematics, and quantitative finance, with a special focus on functional programming. Johan is passionate about languages such as F#, Clojure, and Haskell, and operating systems such as Linux, Mac OS X, and Windows for his work. Most of Johan's quantitative background comes from Lund University, where he studied courses in computer science, mathematics, and physics. Currently Johan is studying pure mathematics at Lund University, Sweden, and is aiming for a PhD in the future, combining mathematics and functional programming. Professionally, Johan has worked as a part-time developer for Sony Ericsson and various smaller firms in Sweden. He also works as a part-time consultant focusing on web technologies and cloud solutions. You can easily contact him by sending an e-mail to joastbg@gmail.com or visit his GitHub page at <https://github.com/joastbg>.

About the Reviewers

Yan Cui (@theburningmonk) is a lead server-side developer at the London-based, award winning gaming company GameSys. He focuses on building highly distributed and scalable server-side solutions for GameSys's social and mobile games. Yan is a regular speaker on topics such as F#, AOP, and NoSQL at local user groups and conferences in the UK and keeps an active blog at <http://theburningmonk.com>. He is also a co-author of the upcoming book, *F# Deep Dives*, Manning Publications.

Arthur Pham is working for Thomson Reuters as a Lead Quantitative Engineer since 2006. He has spent many years designing and implementing derivatives pricing models and still loves learning new programming languages like F#, C++, Python, Flex/Actionscript, C#, Ruby, and JavaScript.

He currently lives in New York, USA, and can be contacted on Twitter @arthurpham.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Introducing F# Using Visual Studio	5
Introduction	5
Getting started with Visual Studio	6
Creating a new F# project	6
Creating a new project in Visual Studio	6
Understanding the program template	8
Adding an F# script file	9
Understanding F# Interactive	10
Language overview	12
Explaining mutability and immutability	12
Primitive types	13
Explaining type inference	15
Explaining functions	15
Learning about anonymous functions	16
Explaining higher-order functions	17
Currying	17
Investigating lists	18
Concatenating lists	20
Tuples	21
The pipe operator	22
Documenting your code	22
Your first application	22
The whole program	24
Understanding the program	24
Extending the example program	25
The entire program	26
The power of prototyping	27
Functional languages in quantitative finance	27

Understanding the imperative code and interoperability	27
Summary	28
Chapter 2: Learning More About F#	29
Structuring your F# program	30
Looking into modules	30
Using functions and values in modules	31
Namespaces	32
Looking deeper inside data structures	33
Record types	34
Discriminated unions	36
Enumerations	38
Arrays	38
Interesting functions in an array module	40
Lists	41
Pattern matching and lists	44
Interesting functions in a list module	44
Sequences	45
Interesting functions in the sequence module	48
Sets	48
Maps	51
Interesting functions in the map module	52
Options	52
Strings	53
Interesting functions in the string module	54
Choosing data structures	54
Arrays	54
Lists	54
Sets	55
Maps	55
More on functional programming	55
Recursive functions	55
Tail recursion	56
Pattern matching	57
Incomplete pattern matching	58
Using guards	58
Pattern matching in assignment and input parameters	59
Active patterns	59
Introducing generics	62
Lazy evaluation	63
Units of measure	63
Asynchronous and parallel programming	65
Events	65

Background workers	66
Threads	68
Thread pools	69
Asynchronous programming	70
The F# asynchronous workflows	71
Asynchronous binding	71
Examples of using an async workflow	71
Parallel programming using TPL	74
MailboxProcessor	74
A brief look at imperative programming	77
Object-oriented programming	77
Classes	77
Objects and members	78
Methods and properties	79
Overloaded operators	80
Using XML documentation	81
Useful XML tags	82
Typical XML documentation	82
Summary	83
Chapter 3: Financial Mathematics and Numerical Analysis	85
Understanding the number representation	86
Integers	86
Two's complement	86
Floating-point numbers	87
The IEEE 754 floating-point standard	87
Learning about numerical types in F#	88
Arithmetic operators	90
Learning about arithmetic comparisons	90
Math operators	91
Conversion functions	92
Introducing statistics	93
Aggregate statistics	93
Calculating the sum of a sequence	93
Calculating the average of a sequence	94
Calculating the minimum of a sequence	94
Calculating the maximum of a sequence	95
Calculating the variance and standard deviation of a sequence	95
Looking at an example application	97
Using the Math.NET library	98
Installing the Math.NET library	98
Introduction to random number generation	99
Pseudo-random numbers	99
Mersenne Twister	100
Probability distributions	100

Normal distribution	100
Statistics	102
Linear regression	103
Using the least squares method	103
Using polynomial regression	104
Learning about root-finding algorithms	106
The bisection method	107
Looking at an example	107
Finding roots using the Newton–Raphson method	109
Looking at an example	109
Finding roots using the secant method	110
Looking at an example	110
Summary	111
Chapter 4: Getting Started with Data Visualization	113
Making your first GUI in F#	113
Composing interfaces	114
More about agents	116
The user interface	117
The main application	118
Learning about event handling	119
Displaying data	119
Extending the form to use a table	122
Displaying financial data from Yahoo! Finance	124
Understanding the application code	125
Extending the application to use Bollinger bands	127
Using FSharp.Charting	130
Creating a candlestick chart from stock prices	130
Creating a bar chart	132
Summary	134
Chapter 5: Learning Option Pricing	135
Introduction to options	135
Looking into contract specifications	136
European options	136
American options	136
Exotic options	136
Learning about Wiener processes	136
Learning the Black-Scholes formula	139
Implementing Black-Scholes in F#	141
Using Black-Scholes together with charts	142
Introducing the greeks	145
First-order greeks	145
Second-order greeks	145

Implementing the greeks in F#	146
Delta	146
Gamma	147
Vega	147
Theta	148
Rho	148
Investigating the sensitivity of the greeks	149
Code listing for visualizing the four greeks	152
The Monte Carlo method	153
Summary	155
Chapter 6: Exploring Volatility	157
Introduction to volatility	157
Actual volatility	158
Implied volatility	158
Exploring volatility in F#	159
The complete application	162
Learning about implied volatility	164
Solving for implied volatility	165
Delta hedging using Black-Scholes	168
Exploring the volatility smile	169
Summary	174
Chapter 7: Getting Started with Order Types and Market Data	175
Introducing orders	175
Order types	175
Market orders	176
Limit orders	176
Conditional and stop-orders	176
Order properties	176
Understanding order execution	182
Introducing market data	183
Implementing simple pretrade risk analysis	185
Validating orders	185
Introducing FIX and QuickFIX/N	187
Using FIX 4.2	187
Configuring QuickFIX to use the simulator	187
Summary	202
Chapter 8: Setting Up the Trading System Project	203
Explaining automated trading	203
Understanding software testing and test-driven development	204
Understanding NUnit and FsUnit	205
Requirements for the system	205
Setting up the project	206

Installing the NUnit and FsUnit frameworks	208
Connecting to Microsoft SQL Server	210
Introducing type providers	212
Using LINQ and F#	212
Explaining sample code using type providers and LINQ	213
Creating the remaining table for our project	215
Writing test cases	216
Details about the setup	221
Summary	221
Chapter 9: Trading Volatility for Profit	223
Trading the volatility	223
Plotting payoff diagrams with FSharpCharts	224
Learning directional trading strategies	226
Trading volatility using options	226
Trading the straddle	226
Trading the butterfly spread	228
The long butterfly spread	229
The short butterfly spread	230
Trading the VIX	231
Trading the delta neutral portfolio	231
Deriving the mathematics	232
Hedging with implied volatility	233
Implementing the mathematics	233
Learning relative value trading strategies	234
Trading the slope of the smile	234
Defining the trading strategy	239
Case 1 – increasing the slope	240
Case 2 – decreasing the slope	240
Defining the entry rules	240
Defining the exit rules	241
Summary	241
Chapter 10: Putting the Pieces Together	243
Understanding the requirements	243
Revisiting the structure of the system	244
Understanding the Model-View-Controller pattern	246
The model	246
The view	246
The controller	246
Executing the trading strategy using a framework	247
Building the GUI	252
Presenting information in the GUI	254

Adding support for downloading the data	258
Looking at possible additions to the system	258
Improving the data feed	259
Support for backtesting	259
Extending the GUI	259
Converting to the client-server architecture	260
Summary	260
Index	261

Preface

F# is a functional programming language that allows you to write simple code for complex problems. Currently, it is most commonly used in the financial sector. Quantitative finance makes heavy use of mathematics to model the real world. If you are interested in using F# for your day-to-day work or research in quantitative finance, this book is for you.

This book covers everything you need to know about using functional programming for quantitative finance. Using a functional programming language for quantitative finance will enable you to concentrate more on the model itself rather than the implementation details. Tutorials and snippets are summarized into a trading system throughout this book.

F#, together with .NET, provides a wide range of tools needed to produce high quality and efficient code, from prototyping to production. The example code snippets in this book can be extended into larger blocks of code, and reused and tested easily in a functional language. F# is considered one of the default functional languages of choice for financial and trading-related applications.

What this book covers

Chapter 1, Introducing F# Using Visual Studio, introduces you to F# and its roots in functional languages. You will learn how to use F# in Visual Studio and write your first application.

Chapter 2, Learning More About F#, teaches you more about F# as a language and illustrates the many sides of this paradigm language.

Chapter 3, Financial Mathematics and Numerical Analysis, introduces the toolset we'll need throughout the book to implement financial models and algorithms.

Chapter 4, Getting Started with Data Visualization, introduces some of the most common ways to use F# to visualize data and display information in a GUI.

Chapter 5, Learning Option Pricing, teaches you about options, the Black-Scholes formula and ways of exploring options using the tools at hand.

Chapter 6, Exploring Volatility, digs deeper into the world of Black-Scholes and teaches you about implied volatility.

Chapter 7, Getting Started with Order Types and Market Data, takes a rather pragmatic approach towards finance and implements a basic order management system.

Chapter 8, Setting Up the Trading System Project, builds the foundation for the project and shows how to connect to SQL Server and use LINQ for queries.

Chapter 9, Trading Volatility for Profit, studies various ways of monetizing through movements in volatility and the arbitrage opportunity defining the trading strategy for the project.

Chapter 10, Putting the Pieces Together, shows the final steps towards the complete trading system using a volatility arbitrage strategy and FIX 4.2.

What you need for this book

Apart from an interest in F# and finance, you need a computer with Visual Studio 2012 installed. Visual Studio 2012 is the recommended IDE, supporting F# 3.0.

Who this book is for

This book is for anyone interested in writing F# code in the financial domain, with a quantitative approach. The book is mainly intended to be a source of inspiration and uses a lot of working code examples to illustrate both the concepts of finance and F# as a functional language.

At the end of the book we develop a simple trading system for volatility arbitrage. Details about orders and the FIX protocol are explained, as well as the theory behind the strategy itself. This may work as a foundation for anyone interested in developing their own trading system based on options and volatility.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input are shown as follows: "First we set a flag in the constructor, `WorkerSupportsCancellation = true`, then we check a flag every time we iterate the calculation."


A block of code is set as follows:


```
let rec getSecondLastElement = function
    | head :: tail :: [] -> head
    | head :: tail -> getSecondLastElement tail
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
SocketConnectPort=9878
SocketConnectHost=192.168.0.25
FileStorePath=temp
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "If you run this code, you will see a form with the title **Displaying data in F#**, like the one in the following screenshot."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introducing F# Using Visual Studio

In this chapter, you will learn about the history of F# and its roots in other programming languages. We will also be introducing Visual Studio and the fundamental language constructs of F#. You will be comfortable using the interactive mode for prototyping the code step-by-step. You will get a better understanding of how to build programs in F# by putting pieces together. Also, the basics of the language are covered by using and evaluating the code in the **Read Eval Print Loop (REPL)**.

In this chapter you will learn:

- How to use F# with Visual Studio 2012
- How to use F# Interactive to write the code in a new exploratory way
- The basics of F# and how to write your first non-toy application
- How functional programming will make you more productive

Introduction

Before we dive in to the language itself, we should discuss why we need it in the first place. F# is a powerful language, which may sound like a cliché, but it combines multiple paradigms into real-life productivity and supports the .NET components and libraries natively as well as the **Common Language Infrastructure (CLI)**. Functional programming has long been associated with academics and experts. F# is one of the few languages offering a complete environment that is mature enough to comfortably be integrated into an organization.

Also, F# has extensive support for parallel programming, where advanced features such as asynchronous and multi-threaded concepts are implemented as language constructs. It hides a lot of implementation details from the programmer. In F#, the functional programming paradigm is the main philosophy used to solve problems. The other paradigms, object-oriented and imperative programming, are prioritized to be used as subsidiaries and complements for this main paradigm. Reasons for them to coexist, involves compatibility and pragmatic, real-world productivity concerns.

Getting started with Visual Studio

We will start by introducing Visual Studio as the main tool of choice for this book. Although it's possible to use the standalone F# compiler and your favorite editor, you will most likely be more productive using Visual Studio 2012, as we will do throughout this book.

F# has been a part of Visual Studio since 2010. We will use the latest version of Visual Studio and F# throughout this book. This will enable us to use the latest functionality and enhancements available in Visual Studio 2012 and F# 3.0.

F# is open source, which means you can use it on any supported platform; it's not bound to Microsoft or Visual Studio. There is good support in other IDEs, such as MonoDevelop, which will run on Linux and Mac OS X.



For more information about F# and the F# Software Foundation, visit <http://fsharp.org>.

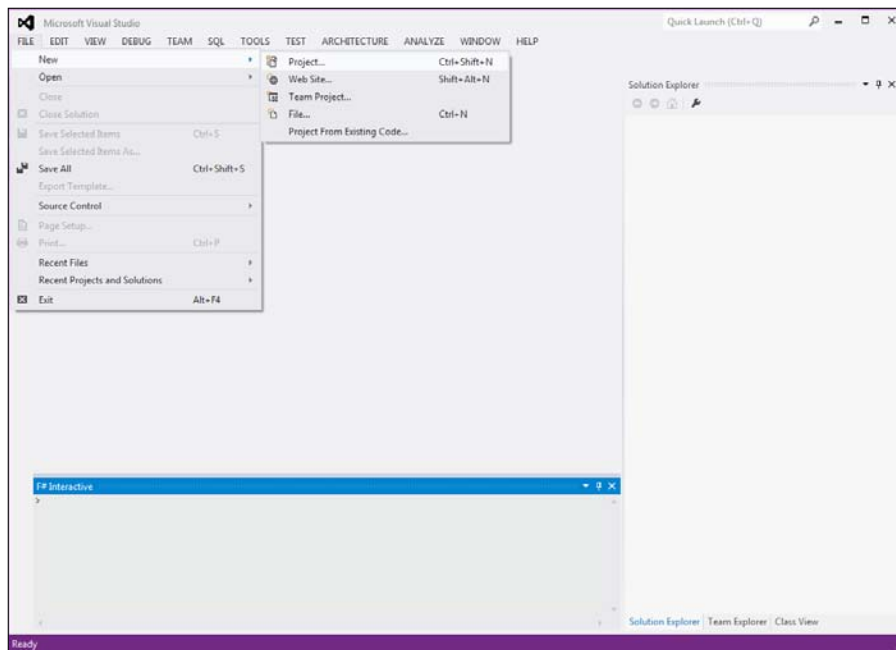
Creating a new F# project

Create a new project in Visual Studio for F#, which is to be used in this guide to explore the basics, as shown in the following sections.

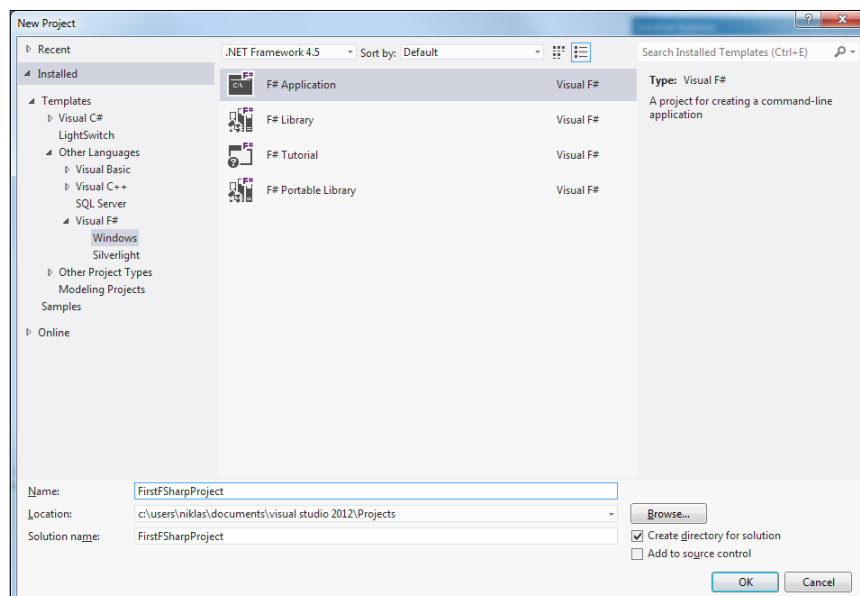
Creating a new project in Visual Studio

Using the following steps, we can create a new project in Visual Studio:

1. To create your first F# project, open Visual Studio 12 and navigate to **File | New | Project**, then, from the menu select **New Project**.



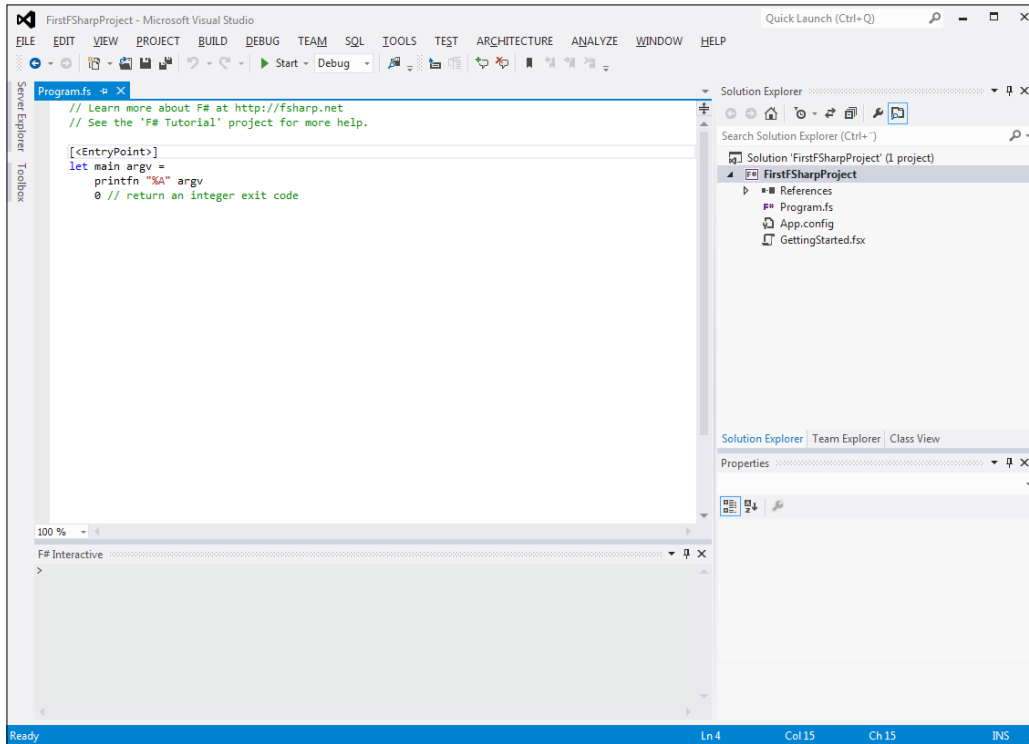
2. Now you will see the **New Project** window appear. Select **F#** in the left panel and then select **F# Application**. You can name it anything you like. Finally, click on **OK**.



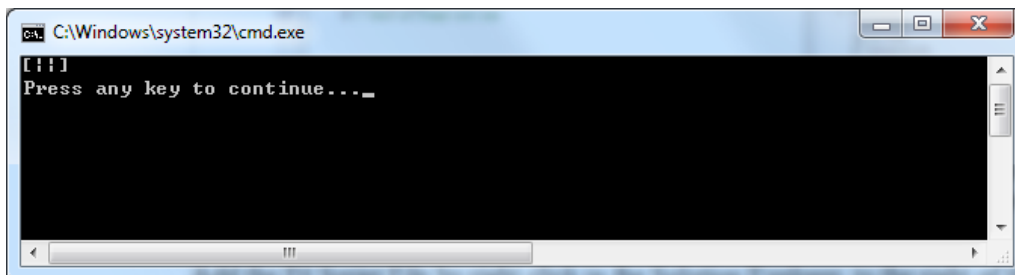
3. Now you have created your first F# application, which will just print the arguments passed to it.

Understanding the program template

Let's have a brief look at the program template generated by Visual Studio.



If you run this program, which will just print out the arguments passed to it, you will see a terminal window appear.

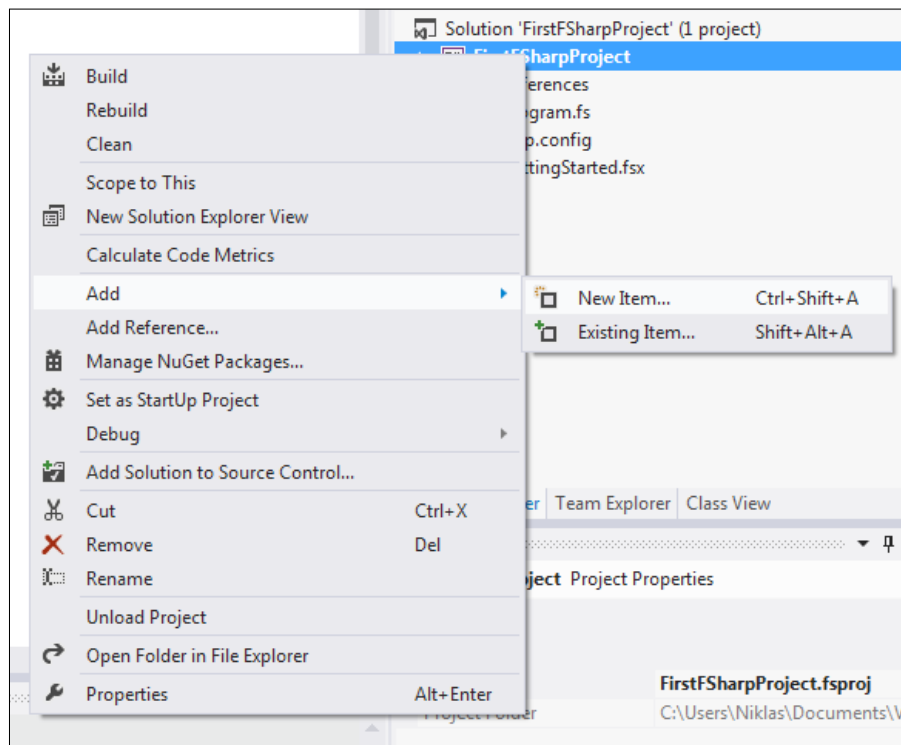


The [`<EntryPoint>`] function in the preceding screenshot is the main function, which tells Visual Studio to use that particular function as the entry point for the program executable. We will not dig any deeper into this program template for now, but we will come back to this in the last three chapters when we'll build the trading system.

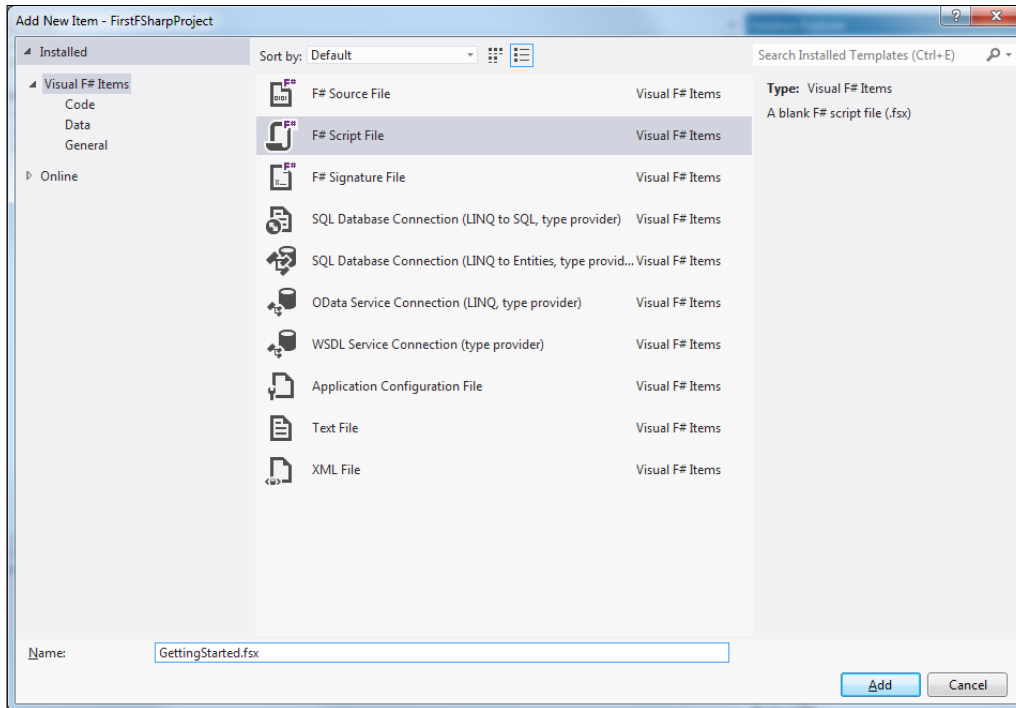
Adding an F# script file

We will use an F# script file after having looked at the standard program template instead of exploring the basics of the language in a more interactive fashion. You can think of F# script files as notebooks, where you have executable code that you can explore in pieces in an incremental style:

1. Add the F# script file by right-clicking on the **Solution Explorer** to the right of the code editor.
2. Then navigate to **Add | New Item...**, as shown in the following screenshot:



3. You can name the script file anything you like, such as `GettingStarted.fsx`.



Now that we have set up the basic project structure in Visual Studio, let's continue and explore F# Interactive.

Understanding F# Interactive

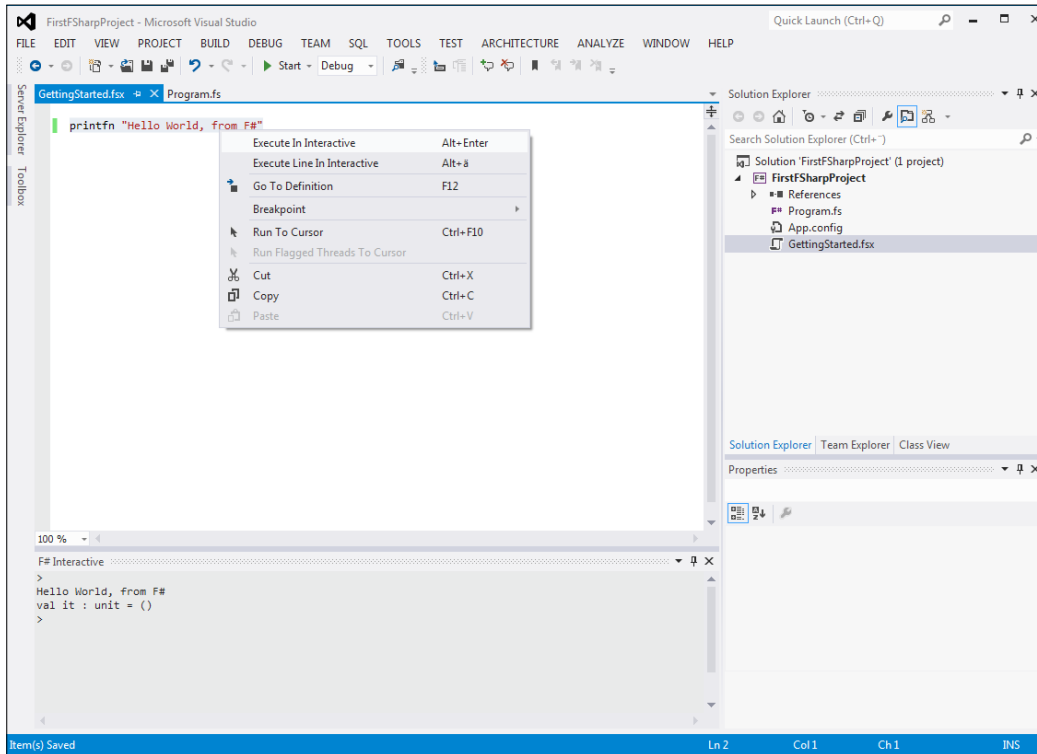
F# Interactive is a way of executing parts of a program interactively. Doing this enables you as a programmer to explore parts of the code and how it behaves. You will have a more dynamic feel for writing code. It's also more fun. F# Interactive is a REPL for F#. This means, it will read the code, evaluate it, and then print out the result. It will then do this over and over again. It's much like a command line, where the code is executed and the result is displayed to the user.

To execute a code in F# Interactive, have a look at the following steps:

1. Select the source code you are interested in and press *Alt + Enter*.
2. You can write a simple line of code that will just print a string to the REPL's output window:

```
printfn "Hello World, from F"
```

- It's also possible to right-click on the selected code and choose **Execute In Interactive**.



When executing the code using the Interactive mode, the result is shown in the **F# Interactive Evaluation** window below the code editor. It is also possible, and sometimes preferable to enter snippets into the **Interactive** window like the following example illustrates.

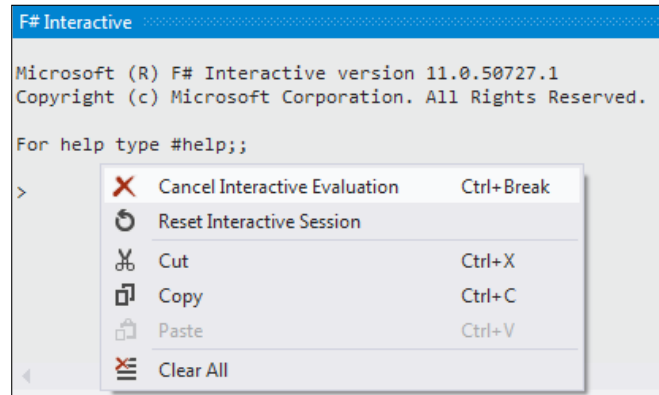
- Enter the following line in the **F# Interactive** window and press *Enter*:

```
printfn "Hello World, from F#";;
```
- This will be evaluated to the following in the REPL:

```
> printfn "Hello World, from F#";;
Hello World, from F#
val it : unit = ()
```

Using double semicolons (`;;`) after the line will terminate the input and enable you to just hit the *Enter* key, they are required if you type directly into the terminal window.

6. If you want to cancel the evaluation, it's possible to right-click on and then select **Cancel Interactive Evaluation**, or simply press *Ctrl + Break*.



Language overview

We will now start our journey into functional programming using F#, and explore its capabilities in quantitative finance applications.

Let's start by looking at how values are declared, that is, how to bind values to names, mutability, and immutability.

To initialize and create a value, use the `let` keyword. `let` will bind the value on the right-hand side to the variable name on the left-hand side of the equals sign. This is a bind operator, a lot like math.

```
let sum = 4 + 5
let newsum = sum + 3
```

The `let` binding is also used in the same way for binding functions to a name, as we will see in the following sections.

Explaining mutability and immutability

Once a variable is defined to have a particular value, it keeps that value indefinitely. There are exceptions to this, and shadowing can be used to override a previous assignment made within the same scope. Thus, variables in mathematics are immutable. Similarly, variables in F# are immutable with some exceptions.

Immutable variables are default in F#. They are useful because they are thread-safe and easier to reason about. This is one of the reasons you may have heard a lot about immutability recently. The concept is to solve the biggest issues and design flaws of concurrent programming, including shared mutable state. If the values do not change, there is no need to protect them either, which is one of the reasons for promoting immutability in concurrent programming.

If you try to alter the value of an immutable variable, you will encounter a message similar to the following:

```
let immutable = "I am immutable!"
immutable <- "Try to change it..."
... error FS0027: This value is not mutable
```

Sometimes, however, it's desirable to have variables that are mutable. Often the need arises in real-life applications when some global state is shared like a counter. Also, object-oriented programming and interoperability with other .NET languages makes the use of mutability unavoidable.

To create a mutable variable, you simply put the keyword `mutable` in front of the name as shown in the following line of code:

```
let mutable name = firstname + lastname
```

To change the variable, after it is created, use the arrow operator (`<-`) as shown in the following line of code:

```
name <- "John Johnson"
```

This is a little bit different from other languages. But once you have wrapped your head around the concept, it makes more sense. In fact, it will most likely be one of the main ways to reason about variables in the future as well.

Primitive types

It may look like F# is a dynamic typed language like JavaScript, Ruby, or Python. In fact, F# is statically typed like C#, C++, and Java. It uses type inference to figure out the correct types. Type inference is a technique which is used to automatically deduce the type used by analyzing the code. This approach works remarkably well in nearly all situations. However, there are circumstances in which you as a programmer need to make clarifications to the compiler. This is done using type annotations, a concept we will look into in the following sections.

Let's explore some of the built-in types in F# using the REPL.

```
> let anInt = 124;;  
  
val anInt : int = 124
```

This means that F# figured out the type of `anInt` to be of type `int`. It simply inferred the type on the left-hand side to be of the same type as the right-hand side of the assignment. Logically, the type must be the same on both sides of the assignment operator, right?

We can extend our analysis into floating point numbers as shown in the following lines of code:

```
> let anFloat = 124.00;;  
  
val anFloat : float = 124.0
```

Because of the decimal sign, the type is determined to be of type `float`. The same holds true for `double` as shown in the following lines of code:

```
> let anDouble : double = 1.23e10;;  
  
val anDouble : double = 1.23e+10
```

For other types, it works in the same way as expected as shown in the following:

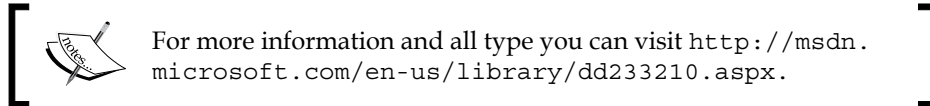
```
> let myString = "This is a string";;  
  
val myString : string = "This is a string"
```

All the primitive built-in types, except for `unit`, have a corresponding type in .NET.

The following table shows the most common primitive types in F#:

Type	.NET type	Description
<code>bool</code>	<code>Boolean</code>	true or false
<code>byte</code>	<code>Byte</code>	0 to 255
<code>int</code>	<code>Int32</code>	-128 to 127
<code>int64</code>	<code>Int64</code>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<code>char</code>	<code>Char</code>	0 to 18,446,744,073,709,551,615
<code>string</code>	<code>String</code>	Unicode text
<code>decimal</code>	<code>Decimal</code>	Floating data type
<code>unit</code>	-	Absence of an actual value

Type	.NET type	Description
void	Void	No type or value
float	Single	64-bit floating point value
double	Double	Same as above



There are also other types that are built into the language which will be covered in more detail in the next chapter, such as lists, arrays, sequences, records, and discriminated unions.

Explaining type inference

Type inference means that the compiler will automatically deduce the type of an expression used in the code, based on the information provided from the programmer about the context of the expression. Type inference analyses the code, as you have seen in the preceding section, to determine types that are often obvious to the programmer. This spares the programmer from having to explicitly define the types of every single variable. It's not always needed to have the types defined to be able to understand the code, as seen in the preceding section for simple assignments of integers and floats. Type inference will make the code easier to write, and as a consequence, easier to read, leaving a lot of ceremony where it belongs.

Explaining functions

It's now time to look at functions, the most basic and powerful building block in F#, and any other functional programming language for that matter. Functional programming languages use functions as first class constructs, in contrast to object-oriented programming, where objects and data are first class constructs. This means that in functional programming, functions will produce data based on the input and not based on state. In object-oriented programming, the state is encapsulated into objects and passed around. Functions are declared in the same way as variables were declared previously in the preceding snippets, with `let` bindings. Have a look at the following code snippet:

```
let sum (x,y) =
    x + y
> sum (7, 7)
```

If you try to evaluate the first `sum` function using *Alt + Enter*, F# Interactive will respond with a function like the following line of code:

```
val sum : x:int -> y:int -> int
```

This means that `sum` is a function that takes two values of type `int` and returns a value of type `int`. The compiler simply knows that it's just the last type that is the return type.

```
let sum (x:float, y:float) =  
    x + y  
  
> sum(7.0, 7.0);;  
val it : float = 14.0
```

Let's look at the following case where parameters of wrong types are passed to the function:

```
> sum(7, 7);;  
...  
error FS0001: This expression was expected to have type float  
but here has type int
```

As seen in the modified version of the `sum` function, the types are explicitly declared as `float`. This is a way of telling the compiler beforehand that `float` is the value to be used in the function. The first version of `sum` used type inference to calculate the types for `x` and `y` respectively and found it to be of type `int`.

Learning about anonymous functions

Since it is common to create small helper functions in F# programming, F# also provides a special syntax for creating anonymous functions. These functions are sometimes called lambdas, or lambda functions. To define an anonymous function, the keyword `fun` is used. Have a look at the following code snippet:

```
let square = (fun x -> x * x)  
> square 2  
val it : int = 4
```

Explaining higher-order functions

Now the square function can be used by itself or as an argument to other functions or higher-order functions. Have a look at the following square function:

```
let squareByFour f
  f 4
> squareByFour square
```

Here, the square function is passed as an argument to the function `squareByFour`. The function `squareByFour` is a higher-order function; it takes another function as an argument. Higher-order functions can take a function as an argument or return a function, or do both. This is an often used technique in functional programming to be able to construct new functions from existing functions and reuse them.

Currying

Though, currying is sometimes considered to be an advanced feature of programming languages, it makes the most sense on connection to functions and higher-order functions. The idea is not complicated at all, and once you have seen a couple of examples, the concept should be clear.

Let's look at the following `sum` function:

```
let sum x y =
  x + y
```

Let's assume we want to reuse the function, but we may often call it for some fixed value of `x`. That means we have a fixed `x`, let's say `2`, and we vary the `y` parameter. Have a look at the following:

```
sum 2 3
sum 2 4
sum 2 5
```

Instead of having to write out the `x` parameter every time, we can make use of the concept of currying. That means we create a new function with the first parameter fixed in this case. Take a look at the following function:

```
let sumBy2 y =
  sum 2 y

> sumBy2 3;;
val it : int = 5

> sumBy2 4;;
```



```
val it : int = 5

> sumBy2 5;;
val it : int = 5
```

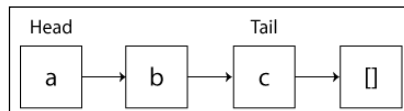
We have now saved ourselves from rewriting some arguments, but this is not the main reason. It's the ability to control the parameters and reuse functionality. More about currying will be covered in later chapters, but the basics were covered here in connection to higher-order functions.

Investigating lists

Lists in F# are very useful, they are some of the most frequently used building blocks. They are the basic building blocks in functional languages and often replace the need of other types or classes. This is because the support for manipulating and creating lists and also being able to nest lists can be enough to replace custom types. You can think of lists as a sequence of values of the same type.

Lists in F# are the following:

- A powerful way to store data
- Immutable linked lists of values of any type
- Often used as building blocks
- One of the best ways to store data



This illustrates a list in F#, with a head and a tail, where each element is linked to the next.

Let's consider the following simple list of price information which are represented as floating points:

```
let prices = [45.0; 45.1; 44.9; 46.0]
> val prices : float list = [45.0; 45.1; 44.9; 46.0]
```

Suppose you want a list with values between 0 and 100, instead of writing them yourself, F# can do it for you. Take a look at the following lines of code:

```
let range = [0 .. 100]
val range : int list =
    [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17;
    18; 19; 20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32;
    33; 34; 35; 36; 37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47;
    48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60; 61; 62;
    63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77;
    78; 79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92;
    93; 94; 95; 96; 97; 98; 99; ...]
```

This is fine if we just want a simple range with fixed size increments. Sometimes however, you may want to have a smaller increment, let's say 0.1, which is between 1.0 and 10.0. The following code shows how it is done:

```
let fineRange = [1.0 .. 0.1 .. 10.0]
val fineRange : float list =
    [1.0; 1.1; 1.2; 1.3; 1.4; 1.5; 1.6; 1.7; 1.8; 1.9; 2.0; 2.1; 2.2;
    2.3; 2.4; 2.5; 2.6; 2.7; 2.8; 2.9; 3.0; 3.1; 3.2; 3.3; 3.4; 3.5;
    3.6; 3.7; 3.8; 3.9; 4.0; 4.1; 4.2; 4.3; 4.4; 4.5; 4.6; 4.7; 4.8;
    4.9; 5.0; 5.1; 5.2; 5.3; 5.4; 5.5; 5.6; 5.7; 5.8; 5.9; 6.0; 6.1;
    6.2; 6.3; 6.4; 6.5; 6.6; 6.7; 6.8; 6.9; 7.0; 7.1; 7.2; 7.3; 7.4;
    7.5; 7.6; 7.7; 7.8; 7.9; 8.0; 8.1; 8.2; 8.3; 8.4; 8.5; 8.6; 8.7;
    8.8; 8.9; 9.0; 9.1; 9.2; 9.3; 9.4; 9.5; 9.6; 9.7; 9.8; 9.9; 10.0]
```

Lists can be of any type, and type inference works here as well. Have a look at the following code:

```
> let myList = ["One"; "Two"; "Three"];;
val myList : string list = ["One"; "Two"; "Three"]
```

However, if you mix the types in a list, the compiler will get confused about the actual type used:

```
let myList = ["One"; "Two"; 3.0];;
...
This expression was expected to have type
string but here has type float
```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Concatenating lists

Concatenating lists is useful when you want to add lists together. This is done using the `@` operator. Have a look at the following code where the `@` operator is used:

```
> let myNewList = [1;2;3] @ [4;5;6];;

val myNewList : int list = [1; 2; 3; 4; 5; 6]

> myNewList;;
val it : int list = [1; 2; 3; 4; 5; 6]
```

Let's have a look at some of the most commonly used functions in the List Module: `Length`, `Head`, `Tail`, `map`, and `filter` respectively.

The function `Length` will simply return the length of the list:

```
> myNewList.Length;;
val it : int = 6
```

If you want the first element of a list, use `Head`:

```
> myNewList.Head;;
val it : int = 1
```

The rest of the list, meaning all other elements except the `Head`, is defined as the `Tail`:

```
> myNewList.Tail;;
val it : int list = [2; 3; 4; 5; 6]
```

You can also do some more interesting things with lists, such as calculating the square of all the elements one by one. Note that it's an entirely new list returned from the `map` function, since lists are immutable. This is done using higher-order functions, where `List.map` takes a lambda function defined to return the value of `x*x` as seen in the following code:

```
> List.map (fun x -> x * x) myNewList;;
val it : int list = [1; 4; 9; 16; 25; 36]
```

Another interesting function is the `filter` function of lists, which will return a new list matching the filter criteria:

```
> List.filter (fun x -> x < 4) myNewList;;
val it : int list = [1; 2; 3]
```

Tuples

Tuples are a group of unnamed but ordered values. The values can be of different types, if needed. You can think of them as more flexible versions of the Tuple class in C#.

```
// Tuple of two floats
(1.0, 2.0)

// Tuple of mixed representations of numbers
(1, 2.0, 3, '4', "four")

// Tuple of expressions
(1.0 + 2.0, 3, 4 + 5)
```

Let's analyze the type information from the tuples in the REPL. The first tuple has the type information:

```
> (1.0, 2.0);;
val it : float * float = (1.0, 2.0)
```

The * symbol is used to separate the type elements for a tuple. It's simply a tuple of two floats. The next one is a bit more complex:

```
> (1, 2.0, 3, '4', "four");;
val it : int * float * int * char * string = (1, 2.0, 3, '4', "four")
```

But the type inference figured it out without any doubts. The last one consists of expressions:

```
> (1.0 + 2.0, 3, 4 + 5);;
val it : float * int * int = (3.0, 3, 9)
```

As you can see, the expressions are evaluated before the type data is analyzed. It may be useful to extract the values from a tuple, this can be done using simple patterns:

```
let (a, b) = (1.0, 2.0)
printfn "%f %f" a b
```

If you are not interested in the first value, use the wildcard character (the underscore) to simply ignore it. The wildcard is used throughout F#, for example, in pattern matching, which will be introduced in the next chapter.

```
let (_, b) = (1.0, 2.0)
printfn "only b %2.2f" b
```

The pipe operator

The pipe operator is used a lot and it's defined as a function which takes the values on the left-hand side of the operator and applies them to the function on the right-hand side. There is another version of the pipe operator with various numbers of arguments, and more about them will be covered later.

The pipe-forward operator (`|>`) is the most common pipe operator:

```
[0..100]
|> List.filter (fun x -> x % 2 = 0)
|> List.map (fun x -> x * 2)
|> List.sum
```

This snippet first creates a list from 0 to 100, as illustrated in the section about lists previously. Then, the list is piped to the filter function with a conditional lambda function. Every even value in the list gets passed on to the next function. The map function will execute the lambda function to square every number. Finally, all numbers are summed, with the result of:

```
val it : int = 5100
```

Documenting your code

Documenting your code is a good practice to get used to. Do you remember details about code you worked on some weeks ago? Then imagine yourself looking at the code you worked on several years ago. This is where documentation comes in. Just some hints about the logic will be sufficient for you and your colleges to grasp the main concepts behind the logic.

```
(*
This is a comment on multiple lines
*)

/// Single line comment, supporting XML-tags

// This is also a single line comment
```

Your first application

The first application for doing something useful will be this Hello World of finance, which will illustrate some powerful yet simple concepts and strengths of F# and functional languages in general.

Let's start our journey into quantitative finance by looking at a simple yet illustrative example using Yahoo finance data. First, we will just put the data into the code to get used to the basic concepts.

First, we put some data in. In F# you can declare a list of strings on multiple lines like the following code:

```
/// Sample stock data, from Yahoo Finance
let stockData = [
    "2013-06-06,51.15,51.66,50.83,51.52,9848400,51.52";
    "2013-06-05,52.57,52.68,50.91,51.36,14462900,51.36";
    "2013-06-04,53.74,53.75,52.22,52.59,10614700,52.59";
    "2013-06-03,53.86,53.89,52.40,53.41,13127900,53.41";
    "2013-05-31,54.70,54.91,53.99,54.10,12809700,54.10";
    "2013-05-30,55.01,55.69,54.96,55.10,8751200,55.10";
    "2013-05-29,55.15,55.40,54.53,55.05,8693700,55.05"
]
```

We introduce a function for splitting strings on commas; this will create an array of strings. Don't forget to evaluate the parts of the program in F# Interactive using *Alt + Enter*. Doing this as a practice will make the number of errors less, and you will also be getting more comfortable and understand the types involved.

The type of the `stockData` value is not explicitly declared, but if you evaluate it, you should see it is of type `string list`:

```
val stockData : string list =
    ["2013-06-06,51.15,51.66,50.83,51.52,9848400,51.52";
     ...
     "2013-05-29,55.15,55.40,54.53,55.05,8693700,55.05"]

// Split row on commas
let splitCommas (l:string) =
    l.Split(',')

// Get the row with lowest trading volume
let lowestVolume =
    stockData
    |> List.map splitCommas
    |> List.minBy (fun x -> (int x.[5]))
```

Evaluating the expression `lowestVolume` will parse the strings in `stockData` and extract the row with the lowest trading volume, column six. Hopefully, the result will be the row with date 2013-05-29, as in the following:

```
val lowestVolume : string [] =
    ["2013-05-29"; "55.15"; "55.40"; "54.53"; "55.05"; "8693700";
     "55.05"]
```

The whole program

The following is the code listing for the program we developed in the previous section, the Hello World program of finance. Try it out for yourself and make changes to it if you like:

```
/// Open the System.IO namespace
open System.IO

/// Sample stock data, from Yahoo Finance
let stockData = [
    "2013-06-06,51.15,51.66,50.83,51.52,9848400,51.52";
    "2013-06-05,52.57,52.68,50.91,51.36,14462900,51.36";
    "2013-06-04,53.74,53.75,52.22,52.59,10614700,52.59";
    "2013-06-03,53.86,53.89,52.40,53.41,13127900,53.41";
    "2013-05-31,54.70,54.91,53.99,54.10,12809700,54.10";
    "2013-05-30,55.01,55.69,54.96,55.10,8751200,55.10";
    "2013-05-29,55.15,55.40,54.53,55.05,8693700,55.05"
]

/// Split row on commas
let splitCommas (l:string) =
    l.Split(',')

/// Get the row with lowest trading volume
let lowestVolume =
    stockData
    |> List.map splitCommas
    |> List.minBy (fun x -> (int x.[5]))
```

Understanding the program

The pipe operator makes the logic of the program very straightforward. The program takes the list `stockData`, splits for commas, then selects specific columns and applies a mathematical operator. Then, it selects the maximum value of these calculations and finally returns the first column of the row fulfilling the `minBy` criteria. You can think of it as building blocks, where each piece is a standalone function on its own. Combining many functions into powerful programs is the philosophy behind functional programming.

Extending the example program

Let's extend the preceding program to read data from a file instead. Since having it explicitly declared in the code is not that useful in the long run, as data tends to change. During this extension, we will also introduce exceptions and how they are used in .NET.

We start by writing a simple function to read all the contents from a file, where its path is passed as an argument. The argument is of type string, as you can see in the function header using the type annotation. Annotations are used either when the compiler can't figure out the type on its own, or when you as a programmer want to clarify the type used or enforce a certain type.

```
/// Read a file into a string array
let openFile (name : string) =
    try
        let content = File.ReadAllLines(name)
        content |> Array.toList
    with
        | :? System.IO.FileNotFoundException as e -> printfn
            "Exception! %s " e.Message; ["empty"]
```

The function will catch `FileNotFoundException` if the file is not found. There is also a new operator `(:?)` before the exception type. This is a type test operator, which returns true if the value matches the specified type, otherwise, returns false.

Let's change the preceding code to use the content loaded from the file instead of the pre-coded stock prices.

```
/// Get the row with lowest trading volume, from file
let lowestVolume =
    openFile filePath
    |> List.map splitCommas
    |> Seq.skip 1
    |> Seq.minBy (fun x -> (int x.[5]))
```

There are some minor changes needed to the code to make it able to handle the input from the **Comma-Separated Values (CSV)** file. As with the input to the pipes, we use the result from the call to the `openFile` function. Then, we split for commas as before. It was necessary to have a way to skip the first line; this is easy to do in F#, and you just insert a `Seq.skip n`, where `n` is the number of elements in the sequence to skip.

```
printfn "Lowest volume, found in row: %A" lowestVolume
```


Here, we simply use `printfn` formatted with `%A`, which will just take anything and format for output (very convenient).

Let's look at one more example of this useful string formatter:

```
> printfn "This works for lists too: %A" [1..5];;
This works for lists too: [1; 2; 3; 4; 5]
val it : unit = ()
```

The entire program

Let's look at the code for the entire program, which we looked at in the previous section.

```
/// Open the System.IO namespace
open System.IO

let filePath = @"table.csv"

/// Split row on commas
let splitCommas (l:string) =
    l.Split(',')

/// Read a file into a string array
let openFile (name : string) =
    try
        let content = File.ReadAllLines(name)
        content |> Array.toList
    with
        | :? System.IO.FileNotFoundException as e -> printfn
            "Exception! %s " e.Message; ["empty"]

/// Get the row with lowest trading volume, from file
let lowestVolume =
    openFile filePath
    |> List.map splitCommas
    |> Seq.skip 1
    |> Seq.minBy (fun x -> (int x.[5]))

/// Use printfn with generic formatter, %A
printfn "Lowest volume, found in row: %A" lowestVolume
```

The power of prototyping

Using the interactive mode in Visual Studio and being able to write the program in smaller building blocks using prototyping is a great way of writing software. You have already used this exploratory way of writing programs with the first application.

The workflow is to build up programs incrementally instead of running all code at once. The REPL is a perfect place to try out snippets and experiment with different aspects of F#.

Functional languages in quantitative finance

In the preceding example code, we saw that parsing data from a file and extracting various information is straightforward, and results in code that is both easy to read and understand. That's one of the highlights of F#, not less important in quantitative finance where code can be complex and hard to follow and comprehend in many languages.

Let's illustrate the preceding statements with another example. The data in the CSV file in the previous sample application was sorted with the most recent date first. If we want the data to be ordered in a more natural way, lowest date first and so on, we can simply reverse the entire list in the following way:

```
/// Reverses the price data from the CSV-file
let reversePrices =
    openFile filePath
    |> List.map splitCommas
    |> List.rev
```

Understanding the imperative code and interoperability

Suppose we are interested in parsing the date column in the example with the stock prices. The entire row looks something like the following:

```
[|"2013-02-22"; "54.96"; "55.13"; "54.57"; "55.02"; "5087300";
"55.02"|]
```

We are interested in the first column, with index 0:

```
lowestVolume.[0];;
val it : string = "2013-02-22"
```

We can make use of the .NET classes for date and time in the `System.DateTime` namespace:

```
> let dateTime = System.DateTime.ParseExact(lowestVolume.[0], "yyyy-  
mm-dd", null);;  
  
val dateTime : System.DateTime = 2013-01-22 00:02:00
```

Now we have a `System.DateTime` object, which is compatible with C# and the other .NET languages, to work with!

Summary

In this chapter, we had a look into the basics of programming in F# using Visual Studio. We covered a wide variety of the language and scratched the surface of functional programming, where immutability plays a key role. Throughout the chapter we started out illustrating some of F#'s language features and how to make use of the .NET framework. At the end of the chapter, we put together a simple application which shows the power and elegant syntax of F#. Functions are the main building block in any functional programming language. Building new functions from existing ones is a way of abstracting away the complexity and allows for reuse.

In the next chapter, we'll dive into more details about the F# language. You'll learn more about data structures, such as Lists, Sequences, and Arrays. You'll also learn how to structure your program using modules and namespaces, things that will become useful in larger programs. The next chapter will also introduce you to threads, thread pools, asynchronous programming using .NET, and language-specific constructs for the F# language.

2

Learning More About F#

This chapter is a more detailed one about the F# language itself. The tutorial approach will continue with demonstrations using Visual Studio, together with detailed aspects of the language relevant to building the final trading system throughout the later part of the book. Most of the language building blocks will be covered with explanations and examples. This chapter is quite large, but it is essential to understand the content provided here to be able to follow along throughout the book.

In this chapter, you will learn:

- Structuring a program into modules and namespaces
- More about data structures and types
- Recursive functions and their role in functional programming
- Pattern matching
- Combining functional and object-oriented ideas
- The imperative part of F#
- The parallel and asynchronous programming model in F#



Download the F# 3.0 Language Specification from Microsoft Research and use it side-by-side as you go along this chapter and the rest of the book. The specification provides a lot of useful details and answers potential questions from you as a reader.

Structuring your F# program

When you write larger programs, it becomes essential to be able to structure the code into hierarchical abstraction levels. This makes it possible to build larger programs, reuse existing code, and let others understand the code as well. In F# there are namespaces, modules, and object orientation together with types and data structures to do this. In object orientation, there are possibilities to make functions and variables private and to disable outside access. Object orientation will be covered in a section by itself later on.

As you may have seen in Visual Studio, when you have created your F# projects, there exist various types of source code files. They have different extensions depending on their use. In this book we use `.fs` and `.fsx` files. The former is an F# source code file to be compiled and used in an executable program. The latter, `.fsx`, is used for F# scripts and interactive mode for prototyping. Scripts are excellent for prototyping and exploratory development, but not suitable for larger programs.

Next, we will cover the most common and useful techniques used in F# to structure the code into elegant, maintainable, and logical structures.

Looking into modules

Modules help in organizing and structuring the related code. They are a simple and elegant way of organizing your code into higher abstractions. They can be thought of as named collections of declarations, such as values, types, and function values. You have been using modules already without noticing. All files will automatically be declared as modules with the same name as the file. The same holds true for F# Interactive, where every execution will be wrapped into a module of its own.

For example, consider a file named `application.fs` with the following content:

```
let myval = 100
```

This file will be compiled in the same way as if it was explicitly declared as a module:

```
module application

let myval = 100
```

That means you don't have to explicitly declare modules in every file to accomplish this.

Using functions and values in modules

Modules may look a lot like classes in object orientation. They even have access annotations to specify the rules for accessing their declared members. The main difference between modules and classes is that classes can be thought of as defining new types, where as modules are grouped functionalities and may be used before knowing the exact details needed to write the class.

The following example illustrates how to declare modules and nested modules with values and functions, and how to access these members:

```
module MainModule =
  let x = 2
  let y = 3
  module NestedModule =
    let f =
      x + y

printfn "%A" MainModule.NestedModule.f
```

As you can see, modules can be nested, and this enables you as a programmer to structure your code in an elegant way. The `f` function in the module `NestedModule` can access the values `x` and `y` in the parent module, without explicitly writing out its name.

The order in which you declare and use your modules is essential because they are processed in a sequential order from top to bottom. The following snippet will not be able to find `Module2` in the first `let` statement:

```
module Module1 =
  let x = Module2.Version() // Error: Not yet declared!

module Module2 =
  let Version() =
    "Version 1.0"
```

Reverse the order and the error is solved:

```
module Module2 =
  let Version() =
    "Version 1.0"

module Module1 =
  let x = Module2.Version() // Now Module2 is found
```

Each member value of a module is made public by default. This means that each member has a default accessibility set by the compiler. Let's go back to the following example and change the accessibility of the function `Version` to private:

```
module Module2 =
    let private Version() =
        "Version 1.0"

module Module1 =
    let x = Module2.Version() // Error: Version is private!
```

As you can see, if you type it into the editor, there is an error because of the private annotation. This is quite neat. In this case, it may be more appropriate to use the annotation internally, which means the member is only accessible from within the same assembly.

```
module Module2 =
    let internal Version() =
        "Version 1.0"

module Module1 =
    let x = Module2.Version() // Now it works again, Version is set to
    be internal
```

The available modifiers are `public`, `internal`, and `private`. The `public` modifier indicates that the annotated function or value can be accessed by all code, whereas `private` indicates that the function or value can only be accessed from the enclosing module.

Namespaces

A namespace is a hierarchical categorization of modules, classes, and other namespaces. First we take a look at how we declare namespaces. Namespaces have to be the first declaration in the code file. They are useful when you want to distinguish functionality without having to put long names in front of, for example, modules or classes. They also minimize naming collisions between your code and the existing code. Here is the code discussed previously added into a namespace:

```
namespace Namespace1.Library1

    module Module2 =
        let internal Version() =
            "Version 1.0"

    module Module1 =
        let x = Module2.Version()
```

This will tell the compiler that we are in the namespace called `Namespace1`. `Library1`. Namespaces also force programmers to use patterns known to F#, C#, and other .NET language programmers. Namespaces are open, which means many source files can contribute to the same namespace. It's also possible to have multiple namespaces in the same file:

```
namespace Namespace1.Library1

    module Module2 =
        let internal Version() =
            "Version 1.0"

namespace Namespace1.Library2

    module Module1 =
        let x = Namespace1.Library1.Module2.Version()
```

The preceding example shows how to use namespaces and how to access modules within namespaces, which requires fully qualified names to be used. If you use the `open` keyword in F#, you don't need the fully qualified names and is equivalent to `using` in C#. Fully qualified names mean the full name, as used to access the function `Version` in the last `let` statement.

For namespaces, the private and public modifiers also exist and work in the same way as for modules, except that they work on the namespace level in the code. There also exists more granular control mechanisms for your code, such as `[<AutoOpen>]`, which will automatically open a module inside of a namespace. This is handy when `let` statements are needed to be defined inside namespaces to define values.

Looking deeper inside data structures

In the previous chapter we introduced some data structures of F# and scratched the surface of their functionality. In this section we will take a deeper look at several data structures and expressions used in many programs.

The following is a list of what will be covered together with a short description to summarize their main characteristics:

- **Record types:** Record types are used to represent data and group pieces of data together by combining named values and types.
- **Discriminated unions:** Discriminated unions are useful to represent heterogeneous data and support data that can be a set of named cases.

- **Enumerations:** Enumerations in F# are almost identical to enumerations in other languages and are used to map labels to constant values.
- **Arrays:** Arrays are collections of a fixed size and must contain values of the same type. Large arrays of constant values can be compiled to efficient binary representations.
- **Lists:** Lists are ordered collections with elements of the same type, implemented as linked lists.
- **Sequences:** In F#, sequences are lazy and are represented as a logical series of elements where the elements have to be of the same type. They are especially suited to represent a large ordered collection of data where all elements are not expected to be used.
- **Sets:** Sets are unordered containers for unique data elements. Sets do not preserve the order of elements as they are inserted, nor do they permit duplicates.
- **Maps:** Maps are associative containers for key/value pairs.
- **Options:** Options are an elegant way of enclosing a value that may or may not exist. It's implemented using a discriminated union. Instead of checking for null values, options are preferred.
- **Strings:** Strings are sequences of characters and are the same as the .NET string.

Record types

Record types are used to represent data and group pieces of data together by combining named values and types.

Suppose we are interested in modeling an **open-high-low-close (OHLC)** bar using record types, it may look something like this:

```
type OHLC =  
    {  
        o: float  
        h: float  
        l: float  
        c: float  
    }
```

Then, we can use the previously defined record to declare a variable:

```
let ohclBar : OHLC = {o = 1.0; h = 2.0; l = 3.0; c = 4.0}
```

Let's consider another example, where we model a quote with `bid`, `ask`, and `midpoint`. The `midpoint()` function will be calculated from the `bid` and `ask` values, that is, the average of both. This can be done using record types and a member function like this:

```
type Quote =
  {
    bid : float
    ask : float
  }
  member this.midpoint() = (this.bid + this.ask) / 2.0

let q : Quote = {bid = 100.0; ask = 200.0}
q.midpoint()
```

As you can see, it looks a lot like the first example except for the member function `midpoint`. The member functions are able to access fields of the record type.

Suppose we are interested in modifying the fields after the initialization, it's simply a matter of adding the keyword `mutable` to the particular field of interest:

```
type Quote =
  {
    mutable bid : float
    mutable ask : float
  }
  member this.midpoint() = (this.bid + this.ask) / 2.0

let q : Quote = {bid = 100.0; ask = 200.0} q.midpoint()
q.bid <- 150.0
q.midpoint()
```

This example is a lot like the one discussed previously, but here we are able to change the value of the `bid` field to `150.0`.

Let's look at how to use record types in pattern matching, because this is one of the biggest reasons for using record types:

```
let matchQuote (quote : Quote) =
  match quote with
  | { bid = 0.0; ask = 0.0 } -> printfn "Both bid and ask is zero"
  | { bid = b; ask = a } -> printfn "bid: %f, ask: %f" b a

let q1 : Quote = {bid = 100.0; ask = 200.0}
let q2 : Quote = {bid = 0.0; ask = 0.0}

matchQuote q1
matchQuote q2
```

In short, record types are:

- Used to represent data
- Useful in pattern matching
- Used to group pieces of data together
- A lot like objects in object orientation
- Powerful and useful features of F#
- A combination of named values of types
- Different from classes because they are exposed as properties and without constructors

Discriminated unions

Discriminated unions are useful to represent heterogeneous data and support data that can be a set of named cases. Discriminated unions represent a finite, well-defined set of choices. Discriminated unions are often the tool of choice to build more complicated data structures including linked lists and a wide range of trees.

Let's investigate some properties of discriminated unions and how they may be used by looking at an example. Here we define a type, `OrderSide`, which can either be buy-side or sell-side.

```
type OrderSide =
    | Buy
    | Sell

let buy = Buy
let sell = Sell

let testOrderSide() =
    printfn "Buy: %A" buy
    printfn "Sell: %A" sell

testOrderSide()
```

This is handy and enables us to write elegant and clean code, doing what we want without any boilerplate coding at all. What about if we desired to have a function to be able to toggle the side of the order? In that case, buys becomes sells and vice versa.

```
type OrderSide =
    | Buy
    | Sell
```

```
let toggle1 =
  match x with
  | Buy -> Sell
  | Sell -> Buy

let toggle2 = function
  | Buy -> Sell
  | Sell -> Buy

let buy = Buy
let sell = Sell

let testOrderSide() =
  printfn "Buy: %A" buy
  printfn "Sell: %A" sell
  printfn "Toggle Buy: %A" (toggle1 buy)
  printfn "Toggle Sell: %A" (toggle2 sell)

testOrderSide()
```

Here, there are two versions of the toggle function, `toggle1` and `toggle2`. The first version uses the match-with style, whereas the latter uses the shorthand version. The shorthand version is sometimes useful because it's shorter and easier to read.

Let's extend our analysis of discriminated unions by introducing recursive fields. They are used to refer to the type itself and will enable you as a programmer to define more complex types. Here is an example where we define an option which can be either `Put` or `Call`, or a combination.

```
type OptionT =
  | Put of float
  | Call of float
  | Combine of OptionT * OptionT
```

Another example of a recursive discriminated union is tree structures. Tree structures are useful to represent hierarchical structures.

```
type Tree =
  | Leaf of int
  | Node of tree * tree

let SimpleTree =
  Node (
    Leaf 1,
    Leaf 2
  )
```

```
// Iterate tree
let countLeaves tree =
    let rec loop sum = function
        | Leaf(_) -> sum + 1
        | Node(tree1, tree2) ->
            sum + (loop 0 tree1) + (loop 0 tree2)
    loop 0 tree
```

Enumerations

Enumerations in F# are almost identical to enumerations in other languages, and are used to map labels to constant values. Enumerations are used to associate a label with a number or predefined values.

Here we define a type, RGB, to map labels to values:

```
// Enumeration
type RGB =
    | Red = 0
    | Green = 1
    | Blue = 2
```

And we use the enumeration to bind a value to the first color, Red:

```
let coll : Color = Color.Red
```

To summarize enumerations:

- It looks like discriminated unions, but they accept that the values can be specified as constants
- They are used to represent constants with labels
- They only hold one piece of data
- They are as safe as discriminated unions, because they can be created with unmapped values

Arrays

Arrays are mutable collections of a fixed size and must contain values of the same type. Large arrays of constant types can be compiled to efficient binary representations.

In F#, arrays are created in the following way:

```
let array1 = [| 1; 2; 3 |]
```

Because arrays are mutable, one can change a value in an array like this:

```
array1.[0] <- 10
```

All elements in an array have to be of the same type. Otherwise, the compiler will complain. Suppose you create an array with one floating point number and two integers:

```
let array2 = [| 1.0; 2; 3 |]
```

The compiler will tell you that the types are inconsistent. Sometimes there is a need to initialize a data structure with values following a logical expression, called array comprehension in this case. For arrays, you can use the following expression to create one:

```
let array3 = [| for i in 1 .. 10 -> i * i |]
```

Accessing elements is straightforward and follows the same pattern for every data structure. It looks a lot like in other programming languages except for the dot in front of the indexing bracket:

```
array1.[0]
```

One convenient feature is the slice notation, used to access a range of elements:

```
array1.[0..2]
```

This can be shortened further, for example, if you select the elements from the beginning to the element with index 2: `array1[..2]`

And the same holds true for the reverse, selecting from the element with index 2 to the end of the array:

```
array1.[2..]
```

Let's look at two examples of initialization of arrays. It can be useful to initialize an array with zeroes. The array module has a function to do this. Have a look at the following code snippet:

```
let arrayOfTenZeroes : int array = Array.zeroCreate 10
```

To create a totally empty array, have a look at the following code snippet:


```
let myEmptyArray = Array.empty
```

There are many useful functions in an array module and we will just look at a few of them here. For example, appending two arrays can be done as follows:

```
printfn "%A" (Array.append [| 1; 2; 3|] [| 4; 5; 6|])
```

The `filter` function is a recurrent candidate, which is useful in many ways. The following will show you how it is applicable on arrays:

```
printfn "%A" (Array.filter (fun elem -> elem % 2 = 0) [| 1 .. 10|])
```

 Check out the MSDN page for more details and examples on how to use the F# array at <http://msdn.microsoft.com/en-us/library/dd233214.aspx>.

Interesting functions in an array module

In the following table, the most useful functions in an array module are presented. This table can be used as a short reference too:

Function	Description
<code>Array.length a</code>	Returns the length of array <code>a</code>
<code>Array.average a</code>	Calculates the average of array <code>a</code> elements. It has to be of float or double data type
<code>Array.min a</code>	Finds the minimum value of elements in array <code>a</code>
<code>Array.max a</code>	Finds the maximum value of elements in array <code>a</code>
<code>Array.filter f a</code>	Filters out the elements in array <code>a</code> matching the predicate in function <code>f</code>
<code>Array.find f a</code>	Returns the first element in list <code>a</code> to match the predicate in function <code>f</code>
<code>Array.empty</code>	Returns an empty array
<code>Array.isEmpty a</code>	Indicates whether the array <code>a</code> is empty
<code>Array.exists f a</code>	Checks whether an element exists in array <code>a</code> which matches the predicate <code>f</code>
<code>Array.sort a</code>	Sorts the elements in array <code>a</code> in increasing order, see <code>sortBy</code> to use predicate
<code>Array.zip a b</code>	Sorts the elements in array <code>a</code> in increasing order, see <code>sortBy</code> to use predicate
<code>Array.map f a</code>	Calls the function <code>f</code> for every element in the array <code>a</code> and forms a new array

Lists

In this section we will investigate lists in more detail and show how to use most of the module functions. Lists in F# are implemented as linked lists and are immutable. They work on lists, and may also be used to convert between lists and other collections.

Lists are ordered collections with elements of the same type; you cannot mix different types in the same list. First we look at how to create and initialize a list and then we investigate the module functions one by one. At the end of this section is a table summarizing all of them.

Let's make use of F# Interactive:

```
> let list1 = [1 .. 10];;  
  
val list1 : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Here we use a range to initialize a list. Expressions for lists are useful when you want a certain pattern or sequence described by a function.

```
> let list2 = [ for i in 1 .. 10 -> i * i ];;  
  
val list2 : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

We'll now investigate two useful operators in lists. We do that using the two previously defined lists: `list1` and `list2`.

```
> 10 :: [10];;  
val it : int list = [10; 10]  
  
> 10 :: list1;;  
val it : int list = [10; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

The previously used operator (`::`) is called the concatenation operator or `cons` operator. Concatenation is an efficient $O(1)$ operator, which prepends elements to the start of a list. This builds a new list from an element and a list. It's possible to append two lists using an operator:

```
> [10] @ list1;;  
val it : int list = [10; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]  
  
> list1 @ list2;;  
val it : int list =  
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```


The concat operator (@) has a performance penalty because lists are immutable and the first list has to be copied, and should be avoided when performance is concerned. This will find the *n*th value of the list:

```
> List.nth list1 3;;  
val it : int = 4
```

To calculate the average value of a list, use the function `List.average`:

```
> let list3 = [10.0 .. 20.0];;  
  
val list3 : float list =  
    [10.0; 11.0; 12.0; 13.0; 14.0; 15.0; 16.0; 17.0; 18.0; 19.0; 20.0]  
  
> List.average list3;;  
val it : float = 15.0
```

We have to use floats here, otherwise the compiler will complain:

```
List.average list1;;  
  
List.average list1;;  
-----^^^^^  
  
... error FS0001: The type 'int' does not support the operator  
'DivideByInt'
```

Minimum and maximum values of a list are found by `List.min` and `List.max`, respectively:

```
> List.min list1;;  
val it : int = 1  
  
> List.max list1;;  
val it : int = 10
```

There is a function, `List.append`, in the list module, equivalent to the operator @, which appends two lists together:

```
> List.append list1 list2;;  
val it : int list =  
    [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 1; 4; 9; 16; 25; 36; 49; 64; 81;  
    100]
```

Let's investigate three functions now: `filter`, `find`, and `exists`. They take a predicate function `f` and a list. The predicate function describes a condition:

```
> List.filter (fun elem -> elem > 10) list1;;
val it : int list = []

> List.filter (fun elem -> elem > 3) list1;;
val it : int list = [4; 5; 6; 7; 8; 9; 10]

> List.find (fun elem -> elem > 3) list1;;
val it : int = 4

> List.exists (fun elem -> elem > 3) list1;;
val it : bool = true

> List.exists (fun elem -> elem > 10) list1;;
val it : bool = false
```

The `zip` function will create a new list using a pair-wise combination of the elements found in lists provided as arguments:

```
> List.zip list1 list2;;
val it : (int * int) list =
  [(1, 1); (2, 4); (3, 9); (4, 16); (5, 25); (6, 36); (7, 49); (8,
  64); (9, 81); (10, 100)]
```

Suppose we want to summarize elements of a list. The function `List.fold` comes in handy:

```
> List.fold (+) 0 list1;;
val it : int = 55
```

This is equal to 55, as expected. We can also find the geometrical sum using `List.fold`:

```
> List.fold (*) 1 list1;;
val it : int = 3628800
```

Note that we use 1 as the initial value to fold, because it is a multiplication operation.

Pattern matching and lists

Let's look at pattern matching and lists, which is a common way to work with lists. Pattern matching provides for a powerful way to create recursive functions on lists. If we want to write a recursive function to get the length of a list, we can use pattern matching.

```
// Get the length of a list
let rec getLengthOfList l = function
    | [] -> printfn "Length of list: %d" l
    | head :: tail -> getLengthOfList (l+1) tail

let myList = [1..10]
getLengthOfList 0 myList
```

As you can see in the preceding short example, pattern matching will make the code easy to read. The first match is for an empty list and the second uses the cons operator to match a list with a head and a tail. The tail is then used in the recursive function `all`, where the length `l` is increased by one for each iteration. Let's look at one more example. If we want to get the second last element of a list, we can use pattern matching.

```
// Get the second last element of a list
let rec getSecondLastElement = function
    | head :: tail :: [] -> head
    | head :: tail -> getSecondLastElement tail

getSecondLastElement myList
```

In the previous example, we can see how pattern matching can be used to express ideas in a clear manner. Because a list is constructed using the cons operator (`::`), we can match arbitrary patterns on lists using it. The first pattern will match `head :: tail :: []`, where `head` is the second last element in the list.

Interesting functions in a list module

In the following table, the most usable functions in the list module are presented. This table can be used as a short reference too:

Function	Description
<code>List.nth a</code>	Returns the <code>nth</code> element of the list <code>a</code>
<code>List.average a</code>	Calculates the average of the list <code>a</code> ; elements have to be of float or double data type
<code>List.max a</code>	Finds the maximum value of elements in list <code>a</code>

Function	Description
List.min a	Finds the minimum value of elements in list a
List.append a b	Appends the two lists a and b
List.filter f a	Filters out the elements in list a, matching the predicate in function f
List.empty	Returns an empty list
List.length a	Returns the length of list a
List.find f a	Returns the first element in list a to match the predicate in function f
List.sort a	Sorts the elements in list a in increasing order, see sortBy to use predicate
List.zip a b	Combines list a and b element wise and forms a new list
List.exists f a	Checks whether an element exists in list a that matches the predicate f
List.fold f s a	Folds list a from left to right, using the function f with start value s
List.head a	Returns the head of list a
List.tail a	Returns the tail of list a
List.map f a	Calls the function f for every element in list a and forms a new list

Sequences

Sequences are logical series' of elements where the elements have to be of the same type. They are especially suited to represent large ordered collections of data where not all elements are expected to be used. Sequences are lazily evaluated and are suitable for large collections of data since not all elements need to be held in memory. Sequence expressions represent sequences of data computed on demand. We will investigate sequences in the same fashion as in the previous sections, using F# Interactive to get a better feel for them, and we will also see how their module functions work.

Let's start by looking at how to initialize and create sequences in various ways using F# Interactive:

```
> seq {1 .. 2}
val it : seq<int> = [1; 2]

> seq {1 .. 10}
val it : seq<int> = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

> seq {1 .. 10 .. 100}
val it : seq<int> = [1; 11; 21; 31; 41; 51; 61; 71; 81; 91]
```

```
> seq {for i in 1 .. 10 do yield i * i}
val it : seq<int> = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]

> seq {for i in 1 .. 10 -> i * i}
val it : seq<int> = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

First we create a sequence defining the elements explicitly. Then, we use ranges. They work in the same way as for arrays and lists. The range expressions are also similar to other collections.

This will find the *n*th value of the sequence:

```
> Seq.nth 3 { 1 .. 10 };;
val it : int = 4
```

Notice that the arguments are swapped in `Seq.nth` compared to, for example, the same function in the list module:

```
> Seq.average {0.0 .. 100.0};;
val it : float = 50.0
```

Minimum and maximum values of a sequence are found by `Seq.min` and `Seq.max`, respectively:

```
> Seq.min seq1;;
val it : int = 1

> Seq.max seq1;;
val it : int = 10
```

You can also append two sequences using the `Seq.append` function:

```
> Seq.append seq1 seq1;;
val it : seq<int> = seq [1; 2; 3; 4; ...]
```

This will create an empty sequence:

```
> Seq.empty;;  
val it : seq<'a> = seq []
```

Let's investigate three functions now: `filter`, `find`, and `exists`. They take a predicate function, `f`, and a list. The predicate function describes a condition. They are identical to the list functions:

```
> Seq.filter (fun elem -> elem > 3) seq1;;  
val it : seq<int> = seq [4; 5; 6; 7; ...]  
  
> Seq.filter (fun elem -> elem > 3) seq1;;  
val it : seq<int> = seq [4; 5; 6; 7; ...]  
  
> Seq.find (fun elem -> elem > 3) seq1;;  
val it : int = 4  
  
> Seq.exists (fun elem -> elem > 3) seq1;;  
val it : bool = true  
  
> Seq.exists (fun elem -> elem > 10) seq1;;  
val it : bool = false
```

The first element of a sequence, the head, is obtained by calling `Seq.head`:

```
> Seq.head seq1;;  
val it : int = 1
```

Notice that there is no `Seq.tail`, that's because of how sequences are represented (they are lazy constructs).

Interesting functions in the sequence module

In the following table, the most useful functions in the sequence module are presented. This table can be used as a short reference too:

Function	Description
<code>Seq.nth a</code>	Returns the <i>n</i> th element of sequence <i>a</i>
<code>Seq.average a</code>	Calculates the average of sequence <i>a</i> . Elements have to be of float or double data type
<code>Seq.min a</code>	Finds the maximum value of elements in sequence <i>a</i>
<code>Seq.max a</code>	Finds the minimum value of elements in sequence <i>a</i>
<code>Seq.append a b</code>	Appends the two sequences <i>a</i> and <i>b</i>
<code>Seq.filter f a</code>	Filters out the elements in sequence <i>a</i> , matching the predicate in function <i>f</i>
<code>Seq.empty</code>	Returns an empty sequence
<code>Seq.find f a</code>	Returns the first element in sequence <i>a</i> to match the predicate in function <i>f</i>
<code>Seq.sort a</code>	Sorts the elements in sequence <i>a</i> in an increasing order. See <code>sortBy</code> to use predicate
<code>Seq.zip a b</code>	Combines list <i>a</i> and <i>b</i> element-wise and forms a new list
<code>Seq.length a</code>	Returns the length of sequence <i>a</i>
<code>Seq.exists f a</code>	Checks whether an element exists in sequence <i>a</i> which matches the predicate <i>f</i>
<code>Seq.fold f s a</code>	Folds sequence <i>a</i> from left to right, using the function <i>f</i> with start value <i>s</i>
<code>Seq.head</code>	Returns the head of sequence <i>a</i>
<code>Seq.map f a</code>	Calls the function <i>f</i> for every element in sequence <i>a</i> and forms a new sequence

Sets

Sets are unordered containers for data elements. Sets do not preserve the order of elements as they are inserted, nor do they permit duplicates.

Let's create a set with three integer elements:

```
> let s1 = set [1; 2; 7];;
```

The preceding set will be of the following type:

```
val s1 : Set<int> = set [1; 2; 7]
```

The type inference works here as expected. Now let's consider adding and inspecting elements in `s1`:

```
> s1.Add(9);;
val it : Set<int> = set [1; 2; 7; 9]
```

Note that `s1` will not be modified due to its immutable characteristics. We can check whether `s1` contains 9:

```
> Set.contains 1 s1;;
val it : bool = true
```

Sometimes it's helpful to create sequences from other data structures. In this case, from a sequence:

```
> let s2 = Set.ofSeq [1..10];;
val s2 : Set<int> = set [1; 2; 3; 4; 5; 6; 7; 8; 9; ...]
```

It works the same way as creating it from an array:

```
> let s3 = Set.ofArray([| for i in 1 .. 5 -> i * i |]);;
val s3 : Set<int> = set [1; 4; 9; 16; 25]
```

To get the length of a set, or in other words, to count the number of elements used, run the following code:

```
> Set.count s1;;
val it : int = 3
```

The `fold` function is also present in the module for sets and is used in the same way as for other collections:

```
> Set.fold (fun a b -> a + b) 0 s1;;
val it : int = 10
```

```
> Set.fold (fun a b -> a * b) 1 s1;;
val it : int = 14
```

This can also be written using the shorthand version `(+)` for addition and `(*)` for multiplication:

```
> Set.fold (+) 0 s1;;
val it : int = 10
```

```
> Set.fold (*) 1 s1;;
val it : int = 14
```


The same holds true for `Set.exists`. The function takes a predicate and a set and returns true if there is any element matching the function:

```
> Set.exists (fun elem -> elem = 2) s1;;  
val it : bool = true  
  
> Set.exists ((=) 4) s1;;val it : bool = false
```

`Filter` is a variant of `exists`, which returns a new set of elements matching the function:

```
> Set.filter (fun elem -> elem > 1) s1;;  
val it : Set<int> = set [2; 7]  
  
> Set.filter (fun elem -> elem < 2) s1;;  
val it : Set<int> = set [1]
```

An interesting function is the `partition` function. It will split the set, in this case, into two new sets: a set of elements which passes the predicate and one that doesn't:

```
> Set.partition (fun elem -> elem < 2) s1;;  
val it : Set<int> * Set<int> = (set [1], set [2; 7])
```

Think for a moment about how to write this `partition` function in an ordinary imperative language. Not as elegant if you ask me. Last but not least, we cover the `map` function. This function should be familiar to you at this stage. We use the old value of `s1` and simply add 2 to every element:

```
> Set.map (fun elem -> elem + 2) s1;;  
val it : Set<int> = set [3; 4; 9]
```

Some of the interesting functions in the `Set` module are as follows:

Function	Description
<code>Set.count a</code>	Returns the number of elements in set <code>a</code>
<code>Set.empty</code>	Returns an empty set
<code>Set.fold f s a</code>	Folds set <code>a</code> from left to right, using the function <code>f</code> with start value <code>s</code>
<code>Set.exists f a</code>	Checks whether an element exists in set <code>a</code> which matches the predicate <code>f</code>
<code>Set.filter f a</code>	Filters out the elements in sequence <code>a</code> matching the predicate in function <code>f</code>
<code>Set.partition f a</code>	Creates a partition, two new sets, from set <code>a</code> using the predicate function <code>f</code>
<code>Set.map</code>	Calls the function <code>f</code> for every element in set <code>a</code> and forms a new set

Maps

Maps are a special kind of set with associative key/value pairs. They are immutable, unordered data structures. They do not preserve the order of elements as they are inserted, nor do they permit duplicates.

Maps are created in much the same way as sets, except we need a key as well:

```
> let m1 = Map.empty.Add("Age", 27);;
val m1 : Map<string,int> = map [("Age", 27)]
```

Now, we can access the value using the key, `Age`:

```
> m1.["Age"];;
val it : int = 27
```

It's possible to create maps from lists of values:

```
> let m2 = ["Year", 2009; "Month", 21; "Day", 3] |> Map.ofList;;
val m2 : Map<string,int> = map [("Day", 3); ("Month", 21); ("Year", 2009)]
```

Because maps are a little different from the collections we covered so far, we will look at some of the more interesting functions in the `map` module.

To filter a map using a predicate, you may notice the small variation here compared to other collections. To ignore either the key or the value, one can replace it with an underscore (`_`), in the same way as in pattern matching:

```
> Map.filter (fun _ v -> v = 27) m1;;
val it : Map<string,int> = map [("Age", 27)]
```

`Map.exists` works almost in the same way as filters:

```
> Map.exists (fun _ v -> v = 27) m1;;
val it : bool = true
```

Partitioning a map using a predicate can be useful. Here we do that using a fixed value:

```
> Map.partition (fun _ v -> v = 27) m1;;
val it : Map<string,int> * Map<string,int> = (map [("Age", 27)], map [])
```

Another useful module function for maps is `Map.containsKey`. This function checks whether the map contains a specific key or not:

```
> Map.containsKey "Age" m1;;  
val it : bool = true  
  
> Map.containsKey "Ages" m1;;  
val it : bool = false
```

Interesting functions in the map module

In the following table, the most useful functions in the `map` module are presented. This table can be used as a short reference too:

Function	Description
<code>Map.add(k, v)</code>	Will create a new map with contents from the map together with the new entry <code>k, v</code>
<code>Map.empty</code>	Creates an empty map
<code>Map.filter f a</code>	Filters out elements in map <code>a</code> matching the predicate in function <code>f</code>
<code>Map.exists f a</code>	Checks whether an element exists in set <code>a</code> which matches the predicate <code>f</code>
<code>Map.partition f a</code>	Creates a partition, two new maps, from map <code>a</code> using the predicate function <code>f</code>
<code>Map.containsKey k a</code>	Checks whether map <code>a</code> contains the key <code>k</code>
<code>Map.fold f s a</code>	Folds map <code>a</code> from left to right, using the function <code>f</code> with start value <code>s</code>
<code>Map.find f a</code>	Returns the first element in map <code>a</code> to match the predicate in function <code>f</code>

Options

Options are an elegant way of enclosing a value that may or may not exist. They are implemented using a discriminated union. Instead of checking for null values, options are preferred.

Here is an example where we use an integer option and pattern matching to investigate them further:

```
let evalOption (o : int option) =  
    match o with
```

```
| Some(a) -> printfn "Found value: %d" a
| None -> printfn "None"
let some : int option = Some(1)
let none : int option = None
```

We can use F# Interactive to investigate the types of options:

```
> evalOption some;;
Found value: 1
val it : unit = ()

> evalOption none;;
None
val it : unit = ()
```

The first option, *some*, contains an integer value as expected. And the other, *none*, is empty. This way, we don't need to use null values and check for them using conditions. We simply pass around option values instead. Along with options, `Nullable` is also available in F#, which explicitly represents the absence of a value.

Strings

Strings should be familiar to you already, and they work in the same way in F#. To use a more formal description of strings, we can say that strings are sequences of characters and are compatible with the .NET string:

```
let str1 = "This is a string"
let str2 = @"This is a string with \ \ //"
let str3 = "" this is "another" string""

printfn "%s" (str1.[0..2])

let str4 = "Hello, " + "world"
let str5 = str1 + str3

String.length str4
String.Compare (str1, "This is a string")
String.Compare (str1, "This is another string")

String.map (fun s -> Char.ToUpper s) str1
```

Interesting functions in the string module

In the following table, the most useful functions in the string module are presented. This table can be used as a short reference too.

Function	Description
<code>String.length s</code>	Returns the length of the string <code>s</code>
<code>String.Empty</code>	Creates an empty string
<code>String.map f s</code>	Maps function <code>f</code> to every character in the string <code>s</code>
<code>String.IsNullOrEmpty</code>	Indicates whether the string <code>s</code> is null or empty
<code>String.IsNullOrEmptyOrWhiteSpace</code>	Indicates whether the string <code>s</code> is null or consists only of white space characters
<code>String.Copy s</code>	Creates a new string with the same character sequence as the string <code>s</code>
<code>String.Concat s1 s2</code>	Concatenates the two strings <code>s1</code> and <code>s2</code> into a new string
<code>String.exists f s</code>	Checks whether any character in <code>s</code> matches the predicate function <code>f</code>
<code>String.Compare s1 s1</code>	Compares the two strings <code>s1</code> and <code>s2</code> . If they match, 0 is returned, otherwise, -1 or 1 is returned depending on the comparison.

Choosing data structures

With this many data structures to choose from, it can be hard to know which one to choose during a specific problem. There are some rules to follow, and here is a short summary of the main characteristics of the individual data structures.

Arrays

Arrays are efficient if you have to know the size of the collection beforehand. That means the size is fixed, and if you want to change the size, you have to create a new array and copy the elements over. On the other hand, random access is very fast; it can be done in constant time.

Lists

Lists are implemented using linked lists, which are items linked together using pointers. This means that traversing a linked list is not super-efficient, because a lot of pointers have to be followed. On the other hand, insertion is very fast at any position in the list. Also worth mentioning is that the lookup of the head element is a constant time operation.

Sets

Sets are implemented as binary trees, where you can't have multiple values defined in the same set. Sets are useful when you don't care about the order and don't allow duplicates.

Maps

Maps are like sets, except that they are extended to use key-value pairs instead of raw values. Maps are very efficient to lookup a value if you know the key.

More on functional programming

Here we will continue and build on the foundation from the previous chapter on functional programming basics. We will examine some of the more advanced and at the same time useful constructs of the F# language.

Recursive functions

Recursion is a fundamental building block in functional programming. Many problems can be solved in a recursive fashion, and together with pattern matching, it makes up a powerful toolkit.

To define a recursive expression, you use the keyword `rec`.

Let's start by looking at the famous Fibonacci sequence, which is defined as the sum of the two previous numbers in a recursive sequence. The first two values are set to 0 and 1, respectively, as seed values:

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

Recursion is a powerful way of solving problems, and is often preferred in functional languages before loop constructs. Let's look at three recursive functions to illustrate its flexibility and power:

```
let rec sum list =
    match list with
    | head :: tail -> head + sum tail
    | [] -> 0
```

This function will sum a list of elements recursively using the match construct on the list argument. The list is split for head and tail in the first match statement and then the function is called again with the tail part. If the list is empty, zero is returned. In the end, the sum of the list will be returned:

```
let rec len list =
    match list with
    | head :: tail -> 1 + len tail
    | [] -> 0
```

To determine the length of a list, we slightly modify the `mysum` function to add one instead of the element for every element encountered. Of course there is a built-in function to do this, as we have seen before. The built-in function `map`, can be constructed using a recursion like this:

```
let rec mymap f = function
    | [] -> []
    | x::xs -> f x::mymap f xs
```

Understanding this function will give you a better understanding about the built-in functions and functional programming in general. Many functions can be defined using recursion and pattern matching. We will learn more about pattern matching in the later part.

Tail recursion

Tail recursion is a way of optimizing the recursion and easing the callback stack. Every time a call is made to the function, a new stack frame is allocated on the stack. This will eventually cause `StackOverflowException`. In other words, tail recursion is used when you expect thousands of iterations.

Tail recursion can be described as:

- An optimization technique
- A way to ease the stack and ensure there are no stack overflows
- Sometimes harder to understand and reason about

To illustrate the concepts of tail recursion, we'll convert a traditional recursively-defined function for the factorial. The factorial, $n!$, is a product of all positive numbers less than or equal to n . For example, $4!$ is defined as $4 * 3 * 2 * 1$, which is 24:

```
let rec factorial1 n =
    match n with
```

```
| 0 | 1 -> 1
| _ -> n * factorial1(n - 1)

let factorial2 n =
  let rec tailrecfact n acc =
    match n with
    | 0 -> acc
    | _ -> trecfact (n - 1) (acc * n)
  tailrecfact n 1
```

We can now verify if the function returns the correct value for 4! as follows:

```
> factorial1 4;;
val it : int = 24

> factorial2 4;;
val it : int = 24
```

Let's try out a somewhat larger factorial. The parameter doesn't have to be big to result in a large answer:

```
> factorial1 10;;
val it : int = 3628800

> factorial2 10;;
val it : int = 3628800
```

Pattern matching

Pattern matching is used for control flow. It allows programmers to look at a value, test it against a series of conditions, and perform certain computations depending on whether that condition is met. It matches different patterns:

```
let sampleMatcher value =
  match value with
  | 0 -> "Zero"
  | 1 -> "One"
  | _ -> "Greather than one"

sampleMatcher 0
```

The preceding snippet defines a function that matches the argument provided to the function with different patterns. This illustrates the basic idea behind pattern matching. We can modify it further.

Incomplete pattern matching

Let's look at an example of incomplete pattern matching:

```
let sampleMatcher value =
    match value with
    | 0 -> "Zero"
    | 1 -> "One"
```

The preceding code snippet is incomplete pattern matching, because we don't consider values other than zero or one. The compiler will tell you this:

```
warning FS0025: Incomplete pattern matches on this expression.
For example, the value '2' may indicate a case not covered by the
pattern(s).
```

This is also true if we consider a simple string pattern matcher:

```
let nameMatcher name =
    match name with
    | "John" -> "The name is John"
    | "Bob" -> "Hi Bob!"
```

This is fixed using a wildcard operator (`_`), like in the first example explained in the preceding snippet:

```
let nameMatcher name =
    match name with
    | "John" -> "The name is John"
    | "Bob" -> "Hi Bob!"
    | _ -> "I don't know you!"

nameMatcher "John"
nameMatcher "Linda"
```

Using guards

In imperative programming, we use `if` statements with expressions to express conditions. This is accomplished using pattern matching and guards. Guards use the keyword `when` to specify the condition. Let's look at an example where this is useful:

```
let sampleMatcher value =
    match value with
    | 0 -> "Zero"
    | 1 -> "One"
    | x when x > 1 -> "Greater than one"
    | _ -> "Some strange value"
```

```
sampleMatcher 0
sampleMatcher 1
sampleMatcher 2
sampleMatcher -1
```

The name *guard* tells us something about their properties. They guard the pattern based on conditions.

Pattern matching in assignment and input parameters

Pattern matching is also useful in assignments together with tuples, like this:

```
> let (bid, ask) = (100.0, 110.0);;
val bid : float = 100.0
val ask : float = 110.0
```

As you can see in the preceding example, the pattern matching mechanism will assign the values to each name. This is handy for multiple assignments:

```
> let (x, y, z) = (3.0, 2.0, 4.0);;
val z : float = 4.0
val y : float = 2.0
val x : float = 3.0
```

It's also possible to use the wildcard to ignore a value in the assignment:

```
> let (x, y, _) = (3.0, 2.0, 4.0);;
val y : float = 2.0
val x : float = 3.0
```

Active patterns

Active patterns allow programmers to wrap ad hoc values and objects in union-like structures for use in pattern matching. First you define the partitioning of the input data using various expressions that act on the data. Each partition can have its own customized logic.

Suppose we want to verify whether an order is valid or not. Let's use the `Order` class, which will be introduced in a later section. The order can be valid and either a market or limit order, or simply invalid. Take a look at the `order` class further in the chapter if you are curious about how it is implemented or its properties.

We start by introducing an active pattern with a very simple one, namely one that figures out if a number is positive or negative:

```
let (|Negative|Positive|) number =
    if number >= 0.0 then
        Positive
    else
        Negative

let TestNumber (number:float) =
    match number with
    | Positive -> printfn "%f is positive" number
    | Negative -> printfn "%f is negative" number
```

We can use F# Interactive to explore the `TestNumber` function using different floating point numbers:

```
> TestNumber 0.0;;
0.000000 is positive
val it : unit = ()

> TestNumber 16.0;;
16.000000 is positive
val it : unit = ()

> TestNumber -7.0;;
-7.000000 is negative
val it : unit = ()
```

Next, we look at the active pattern to verify an order:

```
let (|Limit|Market|Invalid|) (order:Order) =
    if order.Type = OrderType.Limit && order.Price > 0.0 then
        Limit
    else if order.Type = OrderType.Market && order.Price = 0.0 then
        Market
    else
        Invalid

let TestOrder (order:Order) =
    match order with
    | Market -> printfn "Market order"
    | Limit -> printfn "Limit order"
    | Invalid -> printfn "Invalid order"
```

Let's call `TestOrder` with some different order values:

```
> TestOrder (Order(Buy, Limit, 5.0));;
Limit order
```

```
val it : unit = ()

> TestOrder (Order(Sell, Market, 0.0));;
Market order
val it : unit = ()

> TestOrder (Order(Sell, Limit, 0.0));;
Invalid order
val it : unit = ()

> TestOrder (Order(Buy, Market, 2.0));;
Invalid order
val it : unit = ()

> TestOrder (Order(Sell, Invalid, 2.0));;
Invalid order
val it : unit = ()
```

The code works, and you now know more about active patterns and how they can be used to make life simpler.

Partial active patterns are used when parts of the input match, which is helpful in the case where strings need to be parsed to numbers. Let's look at an example where single partial active patterns are used to make this clearer:

```
let (|Integer|_|) str =
    match System.Int32.TryParse(str) with
    | (true,num) -> Some(num)
    | _ -> None

let (|Double|_|) str =
    match System.Double.TryParse(str) with
    | (true,num) -> Some(num)
    | _ -> None

let testParse numberStr =
    match numberStr with
    | Integer num -> printfn "Parsed an integer '%A'" num
    | Double num -> printfn "Parsed a double '%A'" num
    | _ -> printfn "Couldn't parse string: %A" numberStr

> testParse "10.0"
Parsed a double '10.0'
val it : unit = ()
```

```
> testParse "11"
Parsed an integer '11'
val it : unit = ()

> testParse "abc"
Couldn't parse string: "abc"
val it : unit = ()
```

The partial active pattern is used automatically in the matching inside the `testParse` function.

Introducing generics

In this section, we are going to look briefly at generics and how to define generic functions. The F# compiler is capable of finding out whether a function can be generic or not. This function together with type inference is a very powerful combination that allows for clean and easy-to-read code. Although, sometimes you want to specify your own functions as generic. If they are generic, they will be able to handle various types. Doing so will reduce the need of writing several functions with the same logic for every type involved.

Just to illustrate the concept described previously, we can take a look at a function that will create a list based on three parameters. This function is generic, so the type can be specified when used:

```
let genericListMaker<'T>(x, y, z) =
    let list = new List<'T>()
        list.Add(x)
        list.Add(y)
        list.Add(z)
    list
```

Let's use it in F# Interactive:

```
> genericListMaker<int>(1, 2, 3);;val it : List<int> = seq [1; 2; 3]
> genericListMaker<float>(1.0, 2.0, 3.0);;val it : List<float> = seq
[1.0; 2.0; 3.0]
> genericListMaker<string>("1", "2", "3");;val it : List<string> = seq
["1"; "2"; "3"]
```

First we use integers to create a list and the function creates a list for us. Secondly, the function is used with floats. And last but not least, we use ordinary strings. The function works in the same way here in all cases indifferent of the types.

Lazy evaluation

Lazy evaluations or lazy computations are, as the name suggests, lazy. This means that they are not bothered until the last moment, which is when the value is needed. This type of evaluation can help improve the performance of the code. For example, sequences use lazy evaluation. Lazy evaluation also allows expensive computations to be defined without them being evaluated before they are needed. Lazy expressions are generic, and the type is determined when the expression is evaluated.

Let's take a look at how you can define your own lazy constructs:

```
let lazyListFolding =
    lazy
    (
        let someList = [for i in 1 .. 10 -> i * 2]
        List.fold (+) 0 someList
    )
```

We can now evaluate this function using F# Interactive:

```
> let forcedMultiply1 = lazyListFolding.Force();;

val forcedMultiply1 : int = 110
```

When you execute the function definition, the compiler will tell you that the value is not created; it's lazy in the following way:

```
val lazyMultiply : Lazy<int> = Value is not created.
```

Units of measure

Units of measure are a way to associate signed integers and floating point numbers with units. The units can describe weight, length, volume, and currencies. One useful application of units of measure is currency and currency conversion. Units of measure are used to verify the types involved to ensure they are used correctly. All information about them is removed by the compiler after verification and is not part of the resulting executable program.

First we take a look at the normal way of implementing currency conversion in F#. There is no guarantee that the actual rate is correct, or that the calculation uses the correct units:

```
/// Conversion rate representing 1 EUR in USD
let rateEurUsd = 1.28M
```

```
/// Converts amount in EUR to USD
let euroToUsds eur = eur * rateEurUsd

/// Convert 10000 EUR to USD
let usd = euroToUsds 10000.0M
```

There is no robust way to verify the correctness of the conversion made in the preceding snippet with respect to the units. Every value is just treated as floating point numbers. It can be of great importance to be able to verify the units involved. The Mars Climate Orbiter was lost at the end of September 1999 as a result of units. The software on the ground was producing pound-seconds instead of newton-seconds, which led the spacecraft to vaporize in the atmosphere of Mars.

The same holds for currencies, where it's of great importance to be able to verify the correctness of the units used. Let's convert the preceding code to use the units of the measure construct:

```
[<Measure>]
type USD

[<Measure>]
type EUR

let rateEurUsd = 1.28M<EUR/USD>

// Converts amount in EUR to USD
let euroToUsds (eur:decimal<EUR>) = eur * rateEurUsd

// Convert 10000 EUR to USD
let usd = euroToUsds 10000.0M<EUR>
```

Here the compiler will verify that the units are correct using the type information provided. Very handy indeed! What if we use the wrong units? We modify the code to include a measure for YEN:

```
[<Measure>]
type USD

[<Measure>]
type EUR

[<Measure>]
type YEN

let rateEurUsd = 1.28M<EUR/USD>
```

```
// Converts amount in EUR to USD
let euroToUsds (eur:decimal<EUR>) = eur * rateEurUsd

// Convert 10000 EUR to USD
let usd = euroToUsds 10000.0M<YEN>
```

Running this in F# Interactive will result in an error with the following message:

```
error FS0001: Type mismatch. Expecting a
    decimal<EUR>
but given a
    decimal<YEN>
The unit of measure 'EUR' does not match the unit of measure 'YEN'
```

The message is quite clear about what's wrong, and this is a great help for writing the correct code involving different units to be handled and converted in various ways.

Asynchronous and parallel programming

Asynchronous and parallel programming will be introduced in this section, together with events, thread pools, background workers, and the MailboxProcessor (agents). All these constructs exist with one purpose in life: to make life easier for the programmer. Modern computers have CPUs capable of executing several threads in parallel, which opens doors to new possibilities. These possibilities require a good toolkit for concurrent and parallel programming. F# is a very good candidate, and one of its design principles is to be a good fit in these types of situations.

Events

Events are useful when you want a certain function to execute upon a specific event that will occur sometime in the future. This is often applicable to GUI programming, where a user will interact in some way with the interface. The pattern is called event-driven programming, where events drive the execution of the program. The following example illustrates this in a simple manner:

```
open System.Windows.Forms

let form = new Form(Text="F# Events",
                   Visible = true,
                   TopMost = true)

form.Click.Add(fun evArgs -> System.Console.WriteLine("Click event
handler"))
Application.Run(form)
```


Here, a form is created, a regular .NET form. The parameters to the constructor sets the title, the visibility to `true`, and specifies it to be on top. Then, a `click` handler is installed, with a lambda function as the event handler. Upon execution, the function will simply print out a text message to the console.

In F#, you have the possibility to manipulate the event stream. This can be useful if there is a need to filter out certain events or do some manipulation. Events are first-class values in F#, which makes it possible to pass them around like other variables. Let's take a look at how to filter out events. In the following example, the `click` events are filtered out depending on their coordinates:

```
open System.Windows.Forms

let form = new Form(Text="F# Events",
                   Visible = true,
                   TopMost = true)

form.MouseDown
|> Event.filter (fun args -> args.X < 50)
|> Event.map (fun args -> printfn "%d %d" args.X args.Y)
```

Here, we modified the code to listen to the `MouseDown` event, which clicks in the form itself. The `MouseDown` events are then filtered depending on their coordinates, which are part of the event argument. If the event passes the filter, a function is called that prints the coordinates to the console. This filtering process can be very useful and makes it possible to create advanced filters and logic for events with code that's clean and easy to understand.

Background workers

Say you make a program where you need calculations to take part. Sometimes these calculations will run for quite a long time. Background workers are a solution when you want to execute long running tasks that run in the background. The code will be executed in a separate thread:

```
open System.Threading
open System.ComponentModel

let worker = new BackgroundWorker()
worker.DoWork.Add(fun args ->
    for i in 1 .. 50 do
        // Simulates heavy calculation
        Thread.Sleep(1000)
        printfn "%A" i
    )
```

```
worker.RunWorkerCompleted.Add(fun args ->
    printfn "Completed..."
)

worker.RunWorkerAsync()
```

Here is an illustrative example of how to use background workers in the simplest manner. The worker will execute a function that simulates a calculation process and finally will notify us on completion. You can schedule tasks to be executed sequentially just by using the function `Add`. Here are two jobs executed in order, and finally we are notified upon completion:

```
open System.Threading
open System.ComponentModel

let worker = new BackgroundWorker()
worker.DoWork.Add(fun args ->
    for i in 1 .. 50 do
        // Simulates heavy calculation
        Thread.Sleep(1000)
        printfn "A: %A" i
)

worker.DoWork.Add(fun args ->
    for i in 1 .. 10 do
        // Simulates heavy calculation
        Thread.Sleep(500)
        printfn "B: %A" i
)

worker.RunWorkerCompleted.Add(fun args ->
    printfn "Completed..."
)

worker.RunWorkerAsync()
```

Sometimes it's desirable to be able to cancel the execution in `Backgroundworker`. To be able to do this, we have to make some minor modifications to the preceding code. First we set a flag in the constructor, `WorkerSupportsCancellation = true`, and then we check a flag every time we iterate the calculation:

```
open System.ComponentModel

let workerCancel = new BackgroundWorker(WorkerSupportsCancellation =
true)
```

```
workerCancel.DoWork.Add(fun args ->
    printfn "apan %A" args
    for i in 1 .. 50 do
        if (workerCancel.CancellationPending = false) then
            Thread.Sleep(1000)
            printfn "%A" i
    )

workerCancel.RunWorkerCompleted.Add(fun args ->
    printfn "Completed..."
)

workerCancel.RunWorkerAsync()
```

If you run this code, you will see no cancellation. Just that the code is prepared to handle a cancellation. To do a cancellation on the preceding execution, you need to run the following line:

```
workerCancel.CancelAsync()
```

Use F# Interactive and run the main code for some iterations and then run the `CancelAsync()` function. This will terminate the background worker.

Threads

Threads are essential parts in any modern software. In F#, threads are essentially .NET threads with all of the .NET functionality. If you have previous knowledge from any other .NET language, the following code will be familiar to you:

```
open System.Threading

let runMe() =
    for i in 1 .. 10 do
        try
            Thread.Sleep(1000)
        with
            | :? System.Threading.ThreadAbortException as ex ->
                printfn "Exception %A" ex
                printfn "I'm still running..."

let thread = new Thread(runMe)
thread.Start()
```

We spawn a thread by passing a delegate to the `Thread` constructor. The thread will run the function `runMe`. The new part may be the way exceptions are handled. They are handled using pattern matching.

It's possible to spawn many threads. And they will be executed concurrently.

```
open System.Threading

let runMe() =
    for i in 1 .. 10 do
        try
            Thread.Sleep(1000)
        with
            | :? System.Threading.ThreadAbortException as ex ->
                printfn "Exception %A" ex
                printfn "I'm still running..."

let createThread() =
    let thread = new Thread(runMe)
    thread.Start()

createThread()
createThread()
```

Here, two threads are created and they run concurrently. Sometimes the output will be intervened; this is due to the fact that they have no synchronization. Spawning threads are quite costly, and will become noticeable if many threads are spawned and terminated. Every thread uses some memory, and if the threads are short lived, you better use a thread pool to enhance performance.

Thread pools

As mentioned in the previous section, spawning threads is quite costly. This is because it often involves the operating system itself to handle the task. If the threads are short lived, thread pools come to the rescue. A thread pool creates and terminates threads depending on the load. When a thread has finished executing a task, they sit in a queue waiting for the next task. F# uses the .NET `ThreadPool` class. This example will also be familiar to you if you have used the equivalent class in C# or any other .NET language:

```
open System.Threading

let runMe(arg:obj) =
    for i in 1 .. 10 do
```


```
try
    Thread.Sleep(1000)
with
    | :? System.Threading.ThreadAbortException as ex ->
printfn "Exception %A" ex
printfn "%A still running..." arg

ThreadPool.QueueUserWorkItem(new WaitCallback(runMe), "One")
ThreadPool.QueueUserWorkItem(new WaitCallback(runMe), "Two")
ThreadPool.QueueUserWorkItem(new WaitCallback(runMe), "Three")
```

In the preceding code, we enqueue three tasks to be executed by the thread pool. They will all be executed without being enqueued. This is because the thread pool will spawn up to the `ThreadPool.GetMaxThreads()` threads, which are usually 1024 threads. Over that amount, they will be queued.

Asynchronous programming

Asynchronous code performs requests that are not completed immediately. That means they are doing operations that will be completed sometime in the future without blocking the current thread. Instead of waiting for that result to be available, multiple requests can be issued and the result will be handled as soon as it becomes available. This way of programming is called asynchronous programming. The program is not blocking just because of a result that is not yet available. Instead, as mentioned, the program will be notified when the result is ready. A common application is IO, where the CPU time can be used for something better than waiting for the IO operation to complete. There are often a lot of callbacks involved in asynchronous programming. Historically, asynchronous programming has been done in .NET using the **Asynchronous Programming Model (APM)**.

 MSDN has a detailed page about APM, that is, <http://msdn.microsoft.com/en-us/library/ms228963.aspx>.

Without going into a lot of details about the APM and asynchronous callbacks, we will simply introduce the asynchronous workflows in F#. This enables us to write asynchronous code without the need for explicit callbacks.

The F# asynchronous workflows

To use asynchronous workflows, you simply wrap your code that you want to be asynchronous in the `async` block. It's that simple, but the entire truth is not covered yet. There is one more thing. The code inside the `async` block has to be asynchronous in itself to make use of the asynchronous workflow:

```
async { expression }
```

Here the expression is wrapped inside the `async` block. The expression is set up to run asynchronously by `Async.Start`, which means, without blocking the current thread. This is especially desirable if the current thread is the GUI thread.

Asynchronous binding

When you work with asynchronous code and expressions and want to bind them to values, you have to use a modified version of the original `let` keyword, `let!`.

This enables the execution to continue after the binding without blocking the current thread. It's a way of telling the binding that the value is asynchronous and will be used later, when the result is available.

Consider the following code:

```
myFunction1()  
let! response = req.AsyncGetResponse()  
myFunction2()
```

If we don't use the `let!` (let bang) operator here, `myFunction2` will have to wait for the result of the asynchronous request. We simply do not need the result in `response` right away, and can better utilize the CPU by running `myFunction2` instead of nothing.

Examples of using an async workflow

This example illustrates some of the concepts of asynchronous programming using parallel constructs. Here we will download data from Yahoo! finance using an asynchronous function in the `WebClient` class, `AsyncDownloadString`. This function takes a URL and downloads the content. In this case, the content will be CSV, containing the daily OHLC prices from 2010-01-01 until 2013-06-06. We start by just downloading the data in parallel and counting the number of bytes fetched for each stock symbol:

```
open System.Net  
open Microsoft.FSharp.Control.WebExtensions
```

```
/// Stock symbol and URL to Yahoo finance
let urlList = [ "MSFT", "http://ichart.finance.yahoo.com/table.csv?s=MSFT&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
                "GOOG", "http://ichart.finance.yahoo.com/table.csv?s=GOOG&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
                "EBAY", "http://ichart.finance.yahoo.com/table.csv?s=EBAY&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
                "AAPL", "http://ichart.finance.yahoo.com/table.csv?s=AAPL&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
                "ADBE", "http://ichart.finance.yahoo.com/table.csv?s=ADBE&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
                ]

/// Async fetch of CSV data
let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Downloaded historical data for %s, received %d characters" name html.Length
        with
            | ex -> printfn "Exception: %s" ex.Message
    }

/// Helper function to run in async parallel
let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

/// Get max closing price from 2010-01-01 for each stock
runAll()
```

Now that we have seen how to download data in parallel and verified that it actually works, we will extend the example to do some useful operations on the data as well. The code is almost identical to the earlier one, except that we added a function, `getMaxPrice`, to parse the CSV and then iterate the sequence using pipes. We will then extract the maximum closing price for the entire period for each stock. All this is done in parallel, asynchronously:

```
open System.Net
open Microsoft.FSharp.Control.WebExtensions

/// Stock symbol and URL to Yahoo finance
let urlList = [ "MSFT", "http://ichart.finance.yahoo.com/table.csv?s=M
SFT&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
               "GOOG", "http://ichart.finance.yahoo.com/table.csv?s=G
OOG&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
               "EBAY", "http://ichart.finance.yahoo.com/table.csv?s=E
BAY&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
               "AAPL", "http://ichart.finance.yahoo.com/table.csv?s=A
APL&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
               "ADBE", "http://ichart.finance.yahoo.com/table.csv?s=A
DBE&d=6&e=6&f=2013&g=d&a=1&b=1&c=2010&ignore=.csv"
               ]

/// Parse CSV and extract max price
let getMaxPrice(data:string) =
    let rows = data.Split('\n')
    rows
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> float s.[4])
    |> Seq.take (rows.Length - 2)
    |> Seq.max

/// Async fetch of CSV data
let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            let maxprice = (getMaxPrice(html.ToString()))
            printfn "Downloaded historical data for %s, max closing
            price since 2010-01-01: %f" name maxprice
        with
            | ex -> printfn "Exception: %s" ex.Message
    }
}
```



```
/// Helper function to run in async parallel
let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

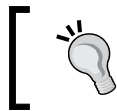
/// Get max closing price from 2010-01-01 for each stock

runAll()
```

We have now looked at two examples, or two versions of the same example. You can experiment and extend this example as much as you like to learn more about how to work with parallel and asynchronous constructs in F#.

Parallel programming using TPL

We have already covered parts of the asynchronous workflow in F# in the previous section. In this section, we'll focus on the .NET TPL (Task Parallel Library).



There are also parallel sequences that you can explore on your own if you are interested.

The TPL is more useful than an asynchronous workflow; if the threads involved are going to do CPU work often. In other words, if there is less IO work to be done, the TPL is the one to be chosen. The TPL is part of the .NET framework 4, and can be used from all the .NET languages.

The simplest way to program parallel programs is to replace for loops with `Parallel.For` and `Parallel.ForEach`, which is found in the `System.Threading` namespace. These two functions will enable loops to execute in parallel instead of in a sequence. Of course, it's not this simple in reality. For example, it's hard if the current iteration depends on another iteration, which is where you have a loop dependence.

MailboxProcessor

The `MailboxProcessor` is also called an agent. Agents are constructs to support concurrent applications without too many insights into the implementation details. They are also more suitable where no shared memory is used, for example, in distributed computer systems with many nodes.

The agent became famous from the Erlang programming language where it is called an actor. The concepts of actors have been partially implemented in various libraries for other programming languages. One major library is Akka for Scala, which is now part of the official Scala distribution.

We'll introduce agents by looking at an example which keeps track of a maximum value. Every time a message is received, the agent recalculates the maximum value. It's also possible to send a reset message. All communication with agents is handled using message passing. Messages are typically discriminated unions. This will be clear after we have walked through an example:

```
open System

// Type for our agent
type Agent<'T> = MailboxProcessor<'T>

// Control messages to be sent to agent
type CounterMessage =
    | Update of float
    | Reset

module Helpers =
    let genRandomNumber (n) =
        let rnd = new System.Random()
        float (rnd.Next(n, 100))

module MaxAgent =
    // Agent to keep track of max value and update GUI
    let sampleAgent = Agent.Start(fun inbox ->
        let rec loop max = async {
            let! msg = inbox.Receive()
            match msg with
            | Reset ->
                return! loop 0.0
            | Update value ->
                let max = Math.Max(max, value)

                Console.WriteLine("Max: " + max.ToString())

                do! Async.Sleep(1000)
                return! loop max
        }
        loop 0.0)

let agent = MaxAgent.sampleAgent
let random = Helpers.genRandomNumber 5
agent.Post(Update random)
```

In this example, we make use of modules to illustrate how to structure the program when it becomes bigger. First we redefine `MailboxProcessor` to be called `Agent`, for simplicity. The module helpers then contain a function to generate a random number.

Finally, the module `MaxAgent` is defined with the `Agent` itself. The `Agent` reacts upon received messages using pattern matching. If the message is the reset message, the value is reset; otherwise, the agent updates the maximum value and waits for one second.

The last three lines will create the agent and send it a random value. You can execute the last two lines in repetition to send multiple random values and update the agent. This will produce an output close to the following:

```
Max: 15

val random : float = 15.0
val it : unit = ()

>

val random : float = 43.0
val it : unit = ()

> Max: 43

val random : float = 90.0
val it : unit = ()

> Max: 90
```

Let's send a reset message and see what happens. Do this by using `F# Interactive` as before:

```
agent.Post(Reset)
```

Now send some updates with random values:

```
let random = Helpers.genRandomNumber 5
agent.Post(Update random)
```

Agents are powerful constructs that act as state machines. You send messages to them and they change state depending on the logic they contain.

A brief look at imperative programming

In this section, we will look at imperative programming and object orientation. It's hard to do object orientation without involving imperative programming. In other words, mutable state. Mutable state and pure functional programming is not a good combination, in fact pure functional programming forbids mutable state totally. To our rescue, F# is not a pure functional programming language, so mutable state is allowed. With this knowledge, we can continue and learn about object orientation and how it's carried out in F#.

Object-oriented programming

F# is a multi-paradigm language where object orientation makes up parts of it. This makes the language interact seamlessly with the other .NET languages in the case of objects. Features that are well known because they are used in almost every modern programming language are imperative programming, object orientation, storing, and manipulating data. Combine imperative and functional programming; F# tried this and has in many ways succeeded. F# objects can have constructors, methods, properties (getters and setters), and fields.

Classes

Classes and objects are the foundation of **object-oriented programming (OOP)**. They are used to model actions, processes, and any conceptual entities in applications. Apart from modules, classes are the most useful way in F# to represent and encapsulate data and related functionality.

Consider a class to represent orders in a trading system. The order will first have order-side, order-type, and a price. We'll continue to extend the functionality of this class in this section to explore classes and object-orientation principles:

```
type OrderSide =
    | Buy
    | Sell

type OrderType =
    | Market
    | Limit

type Order(s: OrderSide, t: OrderType, p: float) =
    member this.S = s
    member this.T = t
    member this.P = p
```

We can use the newly-defined type, `order`, and investigate the member variables using F# Interactive:

```
> let order = Order(Buy, Limit, 45.50);;

val order : Order

> order.S;;
val it : OrderSide = Buy
> order.T;;
val it : OrderType = Limit
> order.P;;
val it : float = 45.5
```

Objects and members

We'll now extend the class `order` to have a function that will enable us to toggle the order-side. The order-side has to be mutable, and we need a member function that performs the actual work. The function to do this will be called `toggleOrderSide` and will use pattern matching for the discriminated union `OrderSide`:

```
// Toggle order side
type Order(s: OrderSide, t: OrderType, p: float) =
    let mutable S = s
    member this.T = t
    member this.P = p

    member this.Side
        with get() = S
        and set(s) = S <- s

    member this.toggleOrderSide() =
        match S with
        | Buy -> S <- Sell
        | Sell -> S <- Buy
```

As seen previously, we investigate the changes to our class using F# Interactive:

```
> let order = Order(Buy, Limit, 45.50);;

val order : Order

> order.Side;;
val it : OrderSide = Buy
> order.toggleOrderSide();;
```

```

val it : unit = ()
> order.Side;;
val it : OrderSide = Sell
> order.toggleOrderSide();;
val it : unit = ()
> order.Side;;
val it : OrderSide = Buy

```

This may look like a lot of new stuff at once. But first we use the keyword `mutable`, to indicate that the value `s` is mutable. This is needed because we want to change its value on an already created object. The opposite is `immutable`, as we talked about in the previous chapter. Pattern matching takes care of the actual work, and the `<-` operator is used to assign new values to mutable variables.



Take a look at the MSDN article about values in F# at <http://msdn.microsoft.com/en-us/library/dd233185.aspx>.

Methods and properties

Wouldn't it be nice to be able to access the price and order-type fields using a better name like `T` or `P`? It's always a good tradition to name your objects, types, and functions in a clear and concise way. This enables both you and other programmers to understand the intention behind the code better.

```

type Order(s: OrderSide, t: OrderType, p: float) =
    let mutable S = s
    member this.T = t
    member this.P = p

    member this.Side
        with get() = S
        and set(s) = S <- s

    member this.Type
        with get() = this.T

    member this.Price
        with get() = this.P

    member this.toggleOrderSide() =
        match S with
        | Buy -> S <- Sell
        | Sell -> S <- Buy

```

It's now possible to access the member values using the properties `side` and `type`, respectively. Properties are just another name for getters and setters. Getters and setters retrieve and change values. You can define members to be read only by omitting the setters, as for the `type` property:

```
> order.Type;;
val it : OrderType = Limit

> order.Price;;
val it : float = 45.5
```

Overloaded operators

Overloaded operators can be useful if a specific functionality has to be exposed without calling a particular function:

```
type Order(s: OrderSide, t: OrderType, p: float) =
    let mutable S = s
    member this.T = t
    member this.P = p

    member this.Side
        with get() = S
        and set(s) = S <- s

    member this.Type
        with get() = this.T

    member this.Price
        with get() = this.P

    member this.toggleOrderSide() =
        S <- this.toggleOrderSide(S)

    member private this.toggleOrderSide(s: OrderSide) =
        match s with
        | Buy -> Sell
        | Sell -> Buy

    static member (~-) (o : Order) =
        Order(o.toggleOrderSide(o.Side), o.Type, o.Price)
```

Look at the last line in this example; here we define an overloaded operator, a unary minus. This operator is defined to toggle the order-side of the order object.

We can use F# Interactive to investigate this and how it is used in the code outside of the order object definition:

```
> let order1 = Order(Buy, Limit, 50.00);;

val order1 : Order

> let order2 = -order1;;

val order2 : Order

> order1;;
val it : Order = FSI_0263+Order {P = 50.0;
                                Price = 50.0;
                                Side = Buy;
                                T = Limit;
                                Type = Limit;}

> order2;;
val it : Order = FSI_0263+Order {P = 50.0;
                                Price = 50.0;
                                Side = Sell;
                                T = Limit;
                                Type = Limit;}
```

As you can see, we first create the `order1` object and then we create `order2` to be defined as `-order1`. This will call the overloaded operator in the `order` class and toggle the order side. Compare the order sides in the last two outputs and see for yourself.

Using XML documentation

XML documentation is useful when you want to autogenerate documentation for your code. If you just put the documentation directly in a triple-slash comment, the entire content will be a summary. The other possibility is to use specific XML tags to specify which type of documentation the specific text belongs to. Doing that enables you, as a programmer, to document the code in a more versatile way.

Useful XML tags

The following are some useful XML tags:

Tag	Description
summary	Summary describes code that is commented on
Returns	Specifies what's returned
Remark	A remark or something to notice about the code
exception	Specifies exceptions that may be thrown from the code
See also	Enables you to link your documentation to other sections for more details

Typical XML documentation

XML documentation is used by Visual Studio and its IntelliSense to provide information about the code. Let's add some XML documentation to a function that we have used before and investigate how it affects IntelliSense:

```
/// <summary>Extracts the maximum closing price from the provided CSV
string</summary>
///<param name="str">Unparsed CSV string.</param>
///<remarks>Will leave the two last lines unhandled, due to Yahoo
specific conditions</remarks>
///<returns>The maximum closing price for the entire sequence.</
returns>
let getMaxPrice(data:string) =
    let rows = data.Split('\n')
    rows
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> float s.[4])
    |> Seq.take (rows.Length - 2)
    |> Seq.max
```

The final result will look something like the following. Here you can see IntelliSense and how it includes the summary at the bottom in the tool tip box:

```
///<summary>Extracts the maximum closing price from the provided CSV string</summary>
///<param name="str">Unparsed CSV string.</param>
///<remarks>Will leave the two last lines unhandled, due to Yahoo specific conditions</remarks>
///<returns>The maximum closing price for the entire sequence.</returns>
let getMaxPrice(data:string) =
    let rows = data.Split('\n')
    rows
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> float s.[4])
    |> Seq.take (rows.Length - 2)
    |> Seq.max

let a = getMaxPrice("2;3;4;5")
    val getMaxPrice: data:string -> float
    Full name: File1.getMaxPrice
    Extracts the maximum closing price from the provided CSV string
```

Summary

In this chapter we looked into the F# language and its various features in more detail. The objective was to introduce in more detail relevant parts of the language to be used later throughout the book. F# is a big language and not all features are useful at once. There are many resources to explore on the Internet, and the F# 3.0 Specification is one of them. Some of the examples here were quite large and contain many of the features and aspects covered.

You will be well prepared for the next chapter if you have digested the material provided here. Next up is financial mathematics and numerical analysis.

3

Financial Mathematics and Numerical Analysis

In this chapter, the reader will be introduced to the basic numerical analysis and algorithm implementation in F#. We will look at how integer and floating-point numbers are implemented, and we will also look at their respective limitations. The basic statistics are covered, and the existing functions in F# are studied and compared with custom implementations.

This chapter will build up the foundation of numerical analysis that can be used when we look at option pricing and volatility later on. We'll also use some of the functionality covered in the previous chapter to implement the mathematical functions for aggregate statistics and to illustrate their usefulness in real life.

In this chapter, you will learn:

- Implementing algorithms in F#
- Numerical concerns
- Implementing basic financial equations
- Curve fitting and regression
- Matrices and vectors in F#

Understanding the number representation

In this section, we will show you how numbers are represented as integers or floating-point numbers in computers. Numbers form the foundation of computers and programming. Everything in a computer is represented by the binary numbers, ones and zeroes. Today, we have 64-bit computers that enable us to have a 64-bit representation of integers and floating-point numbers naively in the CPU. Let's take a deeper look at how integers and floating-point numbers are represented in the following two sections.

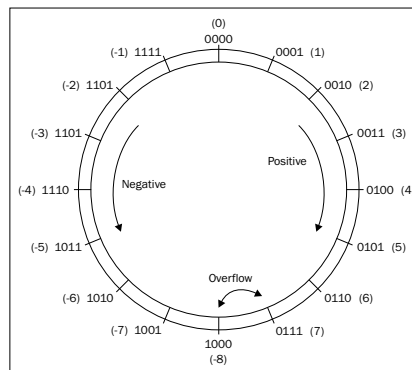
Integers

When we talk about integers, denoted as Z , we are talking specifically about machine-precision integers that are represented exactly in the computer with a sequence of bits. Also, an integer is a number that can be written without a fractional or decimal component and is denoted as Z by convention. For example, 0 is represented as 000000..., 1 is represented as ...000001, 2 is represented as ...000010, and so on. As you can see from this pattern, numbers are represented in the power of two. To represent negative numbers, the number range is divided into two halves and uses two's complement.

When we talk about integer representation without any negative numbers, that is, numbers from zero and up, we talk about unsigned integers.

Two's complement

Two's complement is a way of dividing a range of binary numbers into positive and negative decimal numbers. In this way, both positive and negative numbers can be represented in the computer. On the other hand, this means that the range of numbers is the half for two's complement in relation to the unsigned representation. Two's complement is the main representation used for signed integers.



The representation of integers in two's complement can be thought of as a ring, as illustrated in the preceding figure. The overflow occurs when the maximum allowed positive or negative value increases. Overflow simply means that we pass the barrier between positive and negative numbers.

The following table shows some integers and the representation of their two's complement:

Decimal	Two's complement
127	0111 1111
64	0100 0000
1	0000 0001
0	0000 0000
-1	1111 1111
-64	1100 0000
-127	1000 0001
-128	1000 0000

As you can see, the range for the 8-bit signed integers is from -128 to 127. In more general terms:

$$[-2^{n-1}, 2^{n-1}-1]$$

Floating-point numbers

Floating-point numbers, denoted as \mathbb{R} , represent a quantity where decimals are needed to define them. Another way of describing these numbers is to think of them as values represented as a quantity along a continuous line. They are needed to model things in real life, such as economic, statistical, and physical quantities. In the machine, floating-point numbers are represented by the IEEE 754 standard.

The IEEE 754 floating-point standard

The IEEE 754 floating-point standard describes floating-points using a mantissa and an exponent; see the following figure.

For example, a 64-bit floating point number is made up of the following bit pattern:

Sign bit	Exponent	Mantissa
1 bit	11 bits	52 bits

$$1.2345 = \underbrace{12345}_{\text{Mantissa}} \times \underbrace{10^{-4}}_{\text{Exponent}}$$

An example of floating-point numbers and their binary representations are shown in the following table:

Binary representation	Floating-point number
0x0000000000000000	0.0
0x3ff0000000000000	1.0
0xc000000000000000	-2.0
0x4000000000000000	2.0
0x402E000000000000	15.0

F# Interactive is capable of decoding the representations of floating-point numbers in hexadecimal format into floating-points:

```
> 0x402E000000000000LF;;  
val it: float = 15.0
```



Try out the preceding binary representations in F# Interactive.

Learning about numerical types in F#

In F#, as in most other modern languages, there is a variety of numerical types. The main reason for this is to enable you, as a programmer, to choose the most appropriate numerical type at any given situation. Sometimes there is no need for a 64-bit integer as 8-bit will be enough for small numbers. Another aspect is memory efficiency and consumption, that is, 64-bit integers will consume eight times as much as 8-bit integers.

The following is a table with the most common numerical types used in the F# code. They come in two main varieties; integers and floating-point numbers:

Type	Description	Example
byte	8-bit unsigned integers	10uy, 0xA0uy
sbyte	8-bit signed integers	10y
int16	16-bit signed integers	10s
uint16	16-bit unsigned integers	10us
int, int32	32-bit signed integers	10
uint32	32-bit unsigned integers	10u
int64	64-bit signed integers	10L
uint64	64-bit unsigned integers	10UL
nativeint	Hardware-sized signed integers	10n
unativeint	Hardware-sized signed integers	10un
single, float32	32-bit IEEE 754 floating-point	10.0f
double, float	64-bit IEEE 754 floating-point	10.0
decimal	High-precision decimal	10.0M
bigint	Arbitrary precision integers	10I
complex	Complex numbers using 64-bit floats	Complex(10.0, 10.0)

The following are some examples of how to use suffixes for integers:

```
> let smallestInteger = 10uy;;
val smallestInteger : byte = 10uy

> let smallerInteger = 10s;;
val smallerInteger : int16 = 10s

> let smallInteger = 10us;;
val smallInteger : uint16 = 10us

> let integer = 10L;;
val integer : int64 = 10L
```


Arithmetic operators

Arithmetic operators should be familiar to you; however, we'll cover them in this section for consistency. The operators work as expected, and the succeeding table illustrates this using an example for each one. The remainder operator that returns the remainder from an integer division is worth noticing.

Let's look at an example to see this in more detail. First, we try to divide 10 by 2, and the remainder is 0 as expected:

```
> 10 % 2;;  
val it : int = 0
```

If we try to divide 10 by 3, we get 1 as the remainder, because $3 \times 3 = 9$, and $10 - 9 = 1$:

```
> 10 % 3;;  
val it : int = 1
```

The following table shows arithmetic operators with examples and a description:

Operator	Example	Description
+	$x + y$	Addition
-	$x - y$	Subtraction
*	$x * y$	Multiplication
/	x / y	Division
%	$x \% y$	Remainder
-	$-x$	Unary minus



Arithmetic operators do not check for overflow. If you want to check for overflow, you can use the Checked module. You can find more about the Checked module at <http://msdn.microsoft.com/en-us/library/vstudio/ee340296.aspx>.

Learning about arithmetic comparisons

Arithmetic comparisons are used to compare two numbers for relationships. It's good to know all the operators that are shown in the following table:

Operator	Example	Description
<	$x < y$	Less than
<=	$x \leq y$	Less than or equal to
>	$x > y$	Greater than
>=	$x \geq y$	Greater than or equal to
(=)	$x = y$	Equality
<>	$x \neq y$	Inequality
min	min x y	Minimum
max	max x y	Maximum

Some examples of arithmetic comparisons are as follows:

```
> 5.0 = 5.0;;
val it : bool = true
> 1 < 4;;
val it : bool = true
> 1.0 > 3.0;;
val it : bool = false
```

It is also worth noticing that you can't compare numbers of different types in F#. To do this, you have to convert one of them as follows:

```
> 5.0 >= 10;;
5.0 >= 10
-----^^
stdin(10,8): error FS0001: This expression was expected to have type
float but here has type int
```

Math operators

The following table of mathematical operators covers the most basic mathematical functions that are expected to be found in a programming language or its standard libraries:

Operator	Example	Description
abs	abs x	Overloaded absolute value
acos	acos x	Overloaded inverse cosine
asin	asin x	Overloaded inverse sine
atan	atan x	Overloaded inverse tangent
ceil	ceil x	Overloaded floating-point ceil

Operator	Example	Description
cos	cos x	Overloaded cosine
exp	exp x	Overloaded exponent
floor	floor x	Overloaded floating-point floor
log	log x	Overloaded natural logarithm
log10	log10 x	Overloaded base-10 logarithm
(**)	x ** y	Overloaded exponential
pown	pown x y	Overloaded integer exponential
round	round x	Overloaded rounding
sin	sin x	Overloaded sine function
sqrt	sqrt x	Overloaded square root function
tan	tan x	Overloaded tangent function

Conversion functions

There are no implicit conversions in F# as conversions have to be done manually using conversion routines. Conversion must be made explicitly between types using the operators that are described in the following table:

Operator	Example	Description
byte	byte x	Overloaded conversion to a byte
sbyte	sbyte x	Overloaded conversion to a signed byte
int16	int16	Overloaded conversion to a 16-bit integer
uint16	uint16	Overloaded conversion to an unsigned 16-bit integer
int32, int	Int32 x, int x	Overloaded conversion to a 32-bit integer
uint32	uint32 x	Overloaded conversion to an unsigned 32-bit integer
int64	int64 x	Overloaded conversion to a 64-bit integer
uint64	uint64 x	Overloaded conversion to an unsigned 64-bit integer
nativeint	nativeint x	Overloaded conversion to a native integer
unativeint	unativeint x	Overloaded conversion to an unsigned native integer
float, double	float x, double x	Overloaded conversion to a 64-bit IEEE floating-point number
float32, single	float32 x, single x	Overloaded conversion to a 32-bit IEEE floating-point number

Operator	Example	Description
decimal	decimal x	Overloaded conversion to a System.decimal number
char	char x	Overloaded conversion to a System.Char value
enum	enum x	Overloaded conversion to a typed enum value

This means that there will never be any automatic type conversion behind the scenes that may lead to loss of precision. For example, numbers are not converted from floating-points to integers just to fit the code that you have written. The compiler will tell you that there is an error in the code before converting it (it will never be converted by the compiler). The positive side about this is that you always know the representation of your numbers.

Introducing statistics

In this section, we'll look at statistics using both built-in functions and simple custom ones. Statistics are used a lot throughout quantitative finance. Larger time series are often analyzed, and F# has great support for sequences of numbers; some of its power will be illustrated in the examples mentioned in this section.

Aggregate statistics

Aggregated statistics is all about statistics on aggregated data such as sequences of numbers collected from measurements. It's useful to know the average value in such a collection; this tells us where the values are centered. The `min` and `max` values are also useful to determine the extremes in the collection.

In F#, the `Seq` module has this functionality built-in. Let's take a look at how to use it in each case using an example.

Calculating the sum of a sequence

Consider a sequence of 100 random numbers:

```
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 100 -> rnd()]
```

We can calculate the sum of the preceding sequence, `data`, using the pipe operator and then the module function `Seq.sum`:

```
let sum = data |> Seq.sum
```

Note that the result, `sum`, will vary from time to time due to the fact that we use a random number generator:

```
> sum;;  
val it : float = 42.65793569
```

You might think that the `sum` function is not the most useful one; but there are times you need it, and knowing about its existence in the module library will save you time.

Calculating the average of a sequence

For this example, let's modify the random seed function a bit to generate 500 numbers between 0 and 10:

```
let random = new System.Random()  
let rnd() = random.NextDouble()  
let data = [for i in 1 .. 500 -> rnd() * 10.0]
```

The expected value of this sequence is 5 because of the distribution of the random function:

```
let avg = data |> Seq.average
```

The value may vary a bit due to the fact that we generate numbers randomly:

```
> avg;;  
val it : float = 4.983808457
```

As expected, the average value is almost 5. If we generate more numbers, we will soon come closer and closer to the theoretical expected value of 5:

```
let random = new System.Random()  
let rnd() = random.NextDouble()  
let data = [for i in 1 .. 10000 -> rnd() * 10.0]  
  
let avg = data |> Seq.average  
  
> avg;;  
val it : float = 5.006555917
```

Calculating the minimum of a sequence

Instead of iterating the sequence and keeping track of the minimum value using some kind of a loop construct with a temporary variable, we lend ourselves towards the functional approach in F#. To calculate the minimum of a sequence, we use the module function `Seq.min`:

```
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 10 -> rnd() * 10.0]

val data : float list =
    [5.0530272; 6.389536232; 6.126554094; 7.276151291; 0.9457452972;
     7.774030933; 7.654594368; 8.517372011; 3.924642724; 6.572755164]

let min = data |> Seq.min

> min;;
val it : float = 0.9457452972
```

This looks a lot like the preceding code seen, except that we generate 10 random numbers and inspect the values in the list. If we manually look for the smallest value and then compare it to the one calculated by F#, we see that they match.

Calculating the maximum of a sequence

In the following example, we'll use `Seq.max` to calculate the maximum number of a list:

```
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 5 -> rnd() * 100.0]

val data : float list =
    [7.586052086; 22.3457242; 76.95953826; 59.31953153; 33.53864822]

let max = data |> Seq.max

> max;;
val it : float = 76.95953826
```

Calculating the variance and standard deviation of a sequence

So far, we have already used the existing functions for our statistical analysis. Now, let's implement variance and standard deviation.

Calculating variance

Let's use the following function and calculate the variance for a dice:

```
let variance(values=
  let average = Seq.average values
  values
  |> Seq.map (fun x -> (1.0 / float (Seq.length values)) * (x -
    average) ** 2.0)
  |> Seq.sum
```

A dice has six discrete outcomes, one to six, where every outcome has equal probability. The expected value is 3.5, $(1 + 2 + 3 + 4 + 5 + 6)/6$. The variance of a dice is calculated using the following function:

```
> variance [1.0 .. 6.0];;
val it : float = 2.916666667
```

Calculating standard deviation

We start by implementing the standard deviation function using the previously defined variance function. According to statistics, standard deviation is the square root of the variance:

```
let stddev1(values:seq<float>) = sqrt (variance (values))
```

The preceding function works just fine, but to illustrate the power of sequences, we will implement the standard deviation using the `fold` function. The `fold` function will apply a given function to every element and accumulate the result. The `0.0` value in the end just means that we don't have an initial value. You may remember this from the section about `fold` in the previous chapter. If we were to fold using multiplication, `1.0` is used instead as the initial value. In the end, we just pass the sum to the square root function, `sqrt`, and we are done:

```
let stddev2(values) =
  let avg = Seq.average values
  values
  |> Seq.fold (fun acc x -> acc + (1.0 / float (Seq.length
    values)) * (x - avg) ** 2.0) 0.0
  |> sqrt
```

Let's verify using some sample data:

```
> stddev1 [2.0; 4.0; 4.0; 4.0; 5.0; 5.0; 7.0; 9.0];;
val it : float = 2.0

> stddev2 [2.0; 4.0; 4.0; 4.0; 5.0; 5.0; 7.0; 9.0];;
val it : float = 2.0
```

Now, we can go back and analyze the random data that was generated and used earlier when we looked at the build in the sequence functions:

```
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 100 -> rnd() * 10.0]

let var = variance data
let std = stddev2 data
```

We can check the fact that the square of the standard deviation is equal to the variance:

```
> std * std = var;;
val it : bool = true
```

Looking at an example application

The example application that we'll look at in this section is a combination of the parts that we looked at in this chapter and will simply produce an output with statistics about the given sequence of data:

```
/// Helpers to generate random numbers
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 500 -> rnd() * 10.0]

/// Calculates the variance of a sequence
let variance(values:seq<float>) = values
  |> Seq.map (fun x -> (1.0 / float (Seq.length values)) * (x -
    (Seq.average values)) ** 2.0)
  |> Seq.sum

/// Calculates the standard deviation of a sequence
let stddev(values:seq<float>) = values
  |> Seq.fold (fun acc x -> acc + (1.0 / float (Seq.length
    values)) * (x - (Seq.average values)) ** 2.0) 0.0
  |> sqrt

let avg = data |> Seq.average
let sum = data |> Seq.sum
let min = data |> Seq.min
let max = data |> Seq.max
let var = data |> variance
let std = data |> stddev
```


Evaluating the code results in an output of the statistical properties of the random sequence generated is shown as follows:

```
val avg : float = 5.150620541
val sum : float = 2575.310271
val min : float = 0.007285140458
val max : float = 9.988292227
val var : float = 8.6539651
val std : float = 2.941762244
```

This means that the sequence has an approximate mean value of 5.15 and an approximate standard deviation of 2.94. Using these facts, we can rebuild the distribution assuming that the numbers are distributed according to any of the known distributions, such as normal distribution.

Using the Math.NET library

Instead of implementing your own functions for numerical support, there is an excellent library called **Math.NET**. It's an open-source library covering fundamental mathematics such as linear algebra and statistics.

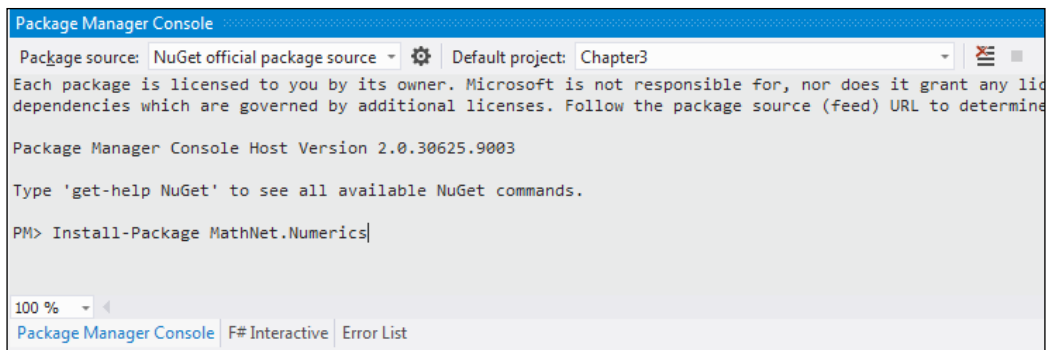
The Math.NET library consists of the following libraries:

- **Math.NET numerics**: Numerical computing
- **Math.NET neodym**: Signal processing
- **Math.NET linq algebra**: Computer algebra
- **Math.NET yttrium**: Experimental network computer algebra

In this section, we'll look at Math.NET numerics, and see how it can help us in our F# programming. First, we need to make sure that Math.NET is installed on our system.

Installing the Math.NET library

The Math.NET library can be installed using the built-in **Package Manager**.



```
Package Manager Console
Package source: NuGet official package source | Default project: Chapter3
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any li
dependencies which are governed by additional licenses. Follow the package source (feed) URL to determin
Package Manager Console Host Version 2.0.30625.9003
Type 'get-help NuGet' to see all available NuGet commands.
PM> Install-Package MathNet.Numerics|
```

1. Open the **Package Manager Console** by going to **View | Other Windows | Package Manager Console**.
2. Type in the following command:
Install-Package MathNet.Numerics
3. Wait for the installation to complete.



You can read more about the Math.NET project on the project's website: <http://www.mathdotnet.com/>.

Introduction to random number generation

Let's start by looking at the various ways of generating random numbers. Random numbers are frequently used in statistics and simulations. They are used extensively in the Monte Carlo simulations. Before we start looking at Math.NET and how to generate random numbers, we need a little bit of the background theory.

Pseudo-random numbers

In computers and programming, random numbers often refer to pseudo-random numbers. Pseudo-random numbers appear to be random, but they are not. In other words, they are deterministic if some properties of the algorithm and the seed that is used are known. The seed is the input to the algorithm to generate the number. Often, one chooses the seed to be the current time of the system or another unique number.

The random number generator in the `System.Random` class that is provided with the .NET platform is based on a subtractive random number generator algorithm by *Donald E. Knuth*. This algorithm will generate the same series of numbers if the same seed is used.

Mersenne Twister

The Mersenne Twister pseudo random number generator is capable of producing less deterministic numbers in an efficient way. These properties make this algorithm one of the most popular ones used today. The following is an example of how to use this algorithm in Math.NET:

```
open MathNet.Numerics.Random

let mersenneTwister = new MersenneTwister(42);
let a = mersenneTwister.NextDouble();
```

In F# Interactive, we can generate some numbers using Mersenne Twister:

```
> mersenneTwister.NextDouble();;
val it : float = 0.7965429842

> mersenneTwister.NextDouble();;
val it : float = 0.9507143116

> mersenneTwister.NextDouble();;
val it : float = 0.1834347877
> mersenneTwister.NextDouble();;
val it : float = 0.7319939383

> mersenneTwister.NextDouble();;
val it : float = 0.7796909974
```

Probability distributions

Probability distributions are commonly used in statistics and finance. They are used to analyze and categorize a set of samples to investigate their statistical properties.

Normal distribution

Normal distribution is one of the most commonly used probability distributions.

In Math.NET, normal distribution can be used in the following way:

```
open MathNet.Numerics.Distributions

let normal = new Normal(0.0, 1.0)
let mean = normal.Mean
let variance = normal.Variance
let stddev = normal.StdDev
```

By using the preceding example, we create a normal distribution with zero mean and a standard deviation of one. We can also retrieve the mean, variance, and standard deviation from a distribution:

```
> normal.Mean;;
val it : float = 0.0

> normal.Variance;;
val it : float = 1.0

> normal.StdDev;;
val it : float = 1.0
```

In this case, the mean and standard deviation is the same as we specified in the constructor of the `Normal` class. It's also possible to generate random numbers from a distribution. We can use the preceding distribution to generate some random numbers, from the properties that are defined:

```
> normal.Sample();;
val it : float = 0.4458429471

> normal.Sample();;
val it : float = 0.4411828389

> normal.Sample();;
val it : float = 0.9845689791

> normal.Sample();;
val it : float = -1.733795869
```

In the Math.NET library, there are also other distributions such as:

- Poisson
- Log normal
- Erlang
- Binomial

Statistics

In Math.NET, there is also great support for descriptive statistics that can be used to determine the properties of a collection of samples. The samples can be numbers from a measurement or generated by the same library.

In this section, we'll look at an example where we'll analyze a collection of samples with known properties, and see how the `DescriptiveStatistics` class can help us out.

We start by generating some data to be analyzed:

```
let dist = new Normal(0.0, 1.0)
let samples = dist.Samples() |> Seq.take 1000 |> Seq.toList
```

Notice the conversion from `Seq` to `List`; this is done because otherwise, `samples` will be a lazy collection. This means that the collection will be a set of different numbers every time it's used in the program, which is not what we want in this case. Next we instantiate the `DescriptiveStatistics` class:

```
let statistics = new DescriptiveStatistics(samples)
```

It will take the samples that were previously created and create an object that describes the statistical properties of the numbers in the `samples` list. Now, we can get some valuable information about the data:

```
// Order Statistics
let maximum = statistics.Maximum
let minimum = statistics.Minimum

// Central Tendency
let mean = statistics.Mean

// Dispersion
let variance = statistics.Variance
let stdDev = statistics.StandardDeviation
```

If we look closer at the mean, variance, and standard deviation respectively, we see that they correspond well with the expected values for the collection:

```
> statistics.Mean;;
val it : float = -0.002646746232

> statistics.Variance;;
val it : float = 1.000011159

> statistics.StandardDeviation;;
val it : float = 1.00000558
```

Linear regression

Linear regression is heavily used in statistics where sample data is analyzed. Linear regression tells the relationship between two variables. It is currently not part of Math.NET but can be implemented using it. Regression in Math.NET is an asked-for feature; and hopefully, it will be supported natively by the library in the future.

Using the least squares method

Let's look at one of the most commonly used methods in linear regression, the least squares method. It's a standard approach to find the approximate solution using the least squares method. The least squares method will optimize the overall solution with respect to the squares of the error, which means that it will find the solution that best fits the data.

The following is an implementation of the least squares method in F# using Math.NET for the linear algebra part:

```
open System
open MathNet.Numerics
open MathNet.Numerics.LinearAlgebra
open MathNet.Numerics.LinearAlgebra.Double
open MathNet.Numerics.Distributions

/// Linear regression using least squares

let X = DenseMatrix.ofColumnsList 5 2 [ List.init 5 (fun i ->
    1.0); [ 10.0; 20.0; 30.0; 40.0; 50.0 ] ] X
let y = DenseVector [| 8.0; 21.0; 32.0; 40.0; 49.0 |]
let p = X.QR().Solve(y)
printfn "X: %A" X
printfn "y: %s" (y.ToString())
printfn "p: %s" (p.ToString())

let (a, b) = (p.[0], p.[1])
```

The independent data y and the dependent data x are used as inputs to the solver. You can use any linear relationship here, between x and y . The regression coefficients will tell us the properties of the regression line, $y = ax + b$.

Using polynomial regression

In this section, we're going to look at a method for fitting a polynomial to data points. This method is useful when the relationship in the data is better described by a polynomial, such as a second or third degree one. We'll use this method in *Chapter 6, Exploring Volatility*, where we'll fit a second degree polynomial to a option data to construct a graph over the volatility smile. We'll lay out the foundations needed for that use case in this section.

We'll continue to use Math.NET for our linear algebra calculations, to solve for the coefficients for a polynomial of a second degree.

We'll start out by generating some sample data for a polynomial:

$$y = x^2 - 3x + 5$$

Then, we generate x-values from -10.0 to 10.0 with increments of 0.2, and y-values using these x-values and the preceding equation with added noise. To accomplish this, the normal distribution with zero mean and 1.0 in standard deviation is used:

```
let noise = (Normal.WithMeanVariance(0.0,0.5))
// Sample points for x^2-3x+5
let xdata = [-10.0 .. 0.2 .. 10.0]
let ydata = [for x in xdata do yield x ** 2.0 - 3.0*x + 5.0 +
             noise.Sample()]
```

Next, we use the linear algebra functions from Math.NET to implement the least square estimation for the coefficients. In mathematical terms, this can be expressed as:

$$\hat{c} = (A^T A)^{-1} A^T \hat{y}$$

This means we will use the matrix A, which stores the x-values and the y-vector to estimate the coefficient vector c. Let's look at the following code to see how this is implemented:

```
let N = xdata.Length
let order = 2

// Generating a Vandermonde row given input v
let vandermondeRow v = [for x in [0..order] do yield v ** (float
x)]
```

```
/// Creating Vandermonde rows for each element in the list
let vandermonde = xdata |> Seq.map vandermondeRow |> Seq.toList

/// Create the A Matrix
let A = vandermonde |> DenseMatrix.ofRowsList N (order + 1)
A.Transpose()

/// Create the Y Matrix
let createYVector order l = [for x in [0..order] do yield l]
let Y = (createYVector order ydata |> DenseMatrix.ofRowsList
  (order + 1) N).Transpose()

/// Calculate coefficients using least squares
let coeffs = (A.Transpose() * A).LU().Solve(A.Transpose()
  * Y).Column(0)

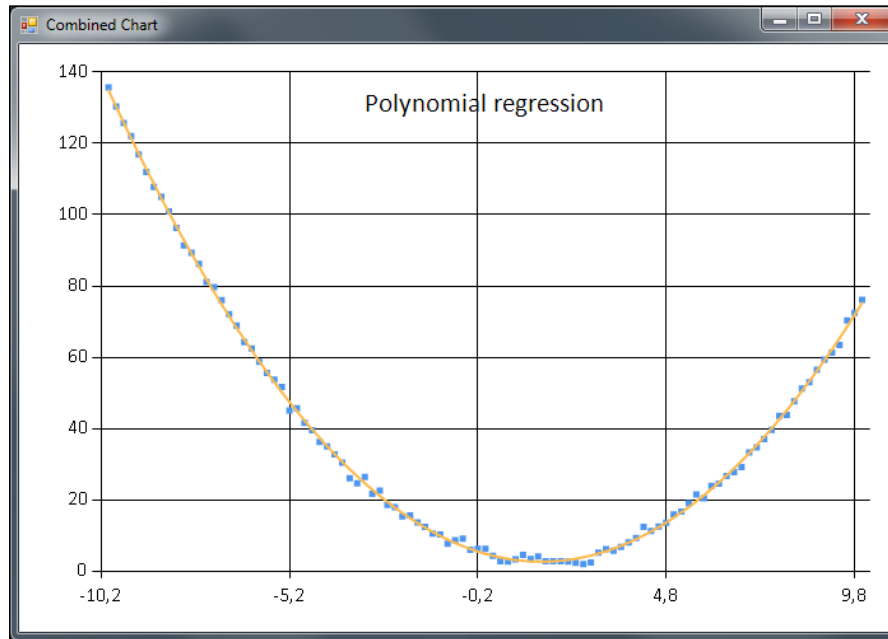
let calculate x = (vandermondeRow(x) |> DenseVector.ofList) *
  coeffs

let fitxs = [(Seq.min xdata).. 0.02 ..(Seq.max xdata)]
let fitys = fitxs |> List.map calculate
let fits = [for x in [(Seq.min xdata).. 0.2 ..(Seq.max xdata)] do
  yield (x, calculate x)]
```

The values in the coefficient vector are in reverse order, which means they correspond to a polynomial that fits the data, but the coefficient is reversed:

```
> coeffs;;
val it = seq [4.947741224; -2.979584718; 1.001216438]
```


The values are pretty close to the polynomial we used as the input in the preceding code. The following is a graph of the sample data points together with the fitted curve. The graph is made using FSharpChart, which we'll look into in the next chapter.



Polynomial regression using Math.net

The curious reader can use the following snippet to produce the preceding graph:

```
open FSharp.Charting
open System.Windows.Forms.DataVisualization.Charting

fsi.AddPrinter(fun (ch:FSharp.Charting.ChartTypes.GenericChart) ->
    ch.ShowChart(); "FSharpCharting")
let chart = Chart.Combine [Chart.Point(List.zip xdata ydata );
    Chart.Line(fits).WithTitle("Polynomial regression")]
```

Learning about root-finding algorithms

In this section, we'll learn about the different methods used in numerical analysis to find the roots of functions. Root-finding algorithms are very useful, and we will learn more about their applications when we talk about volatility and implied volatility.

The bisection method

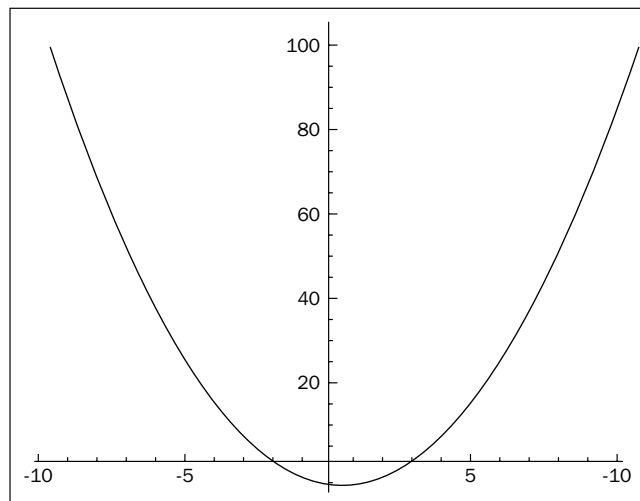
In this section, we will look at a method for finding the roots of a function using the bisection method. This method will be used later in this book to numerically find the implied volatility for an option that is given a certain market price. The bisection method uses iteration and repeatedly bisects an interval for the next iteration.

The following function implements bisection in F#:

```
let rec bisect n N (f:float -> float) (a:float) (b:float)
  (t:float) : float =
  if n >= N then -1.0
  else
    let c = (a + b) / 2.0
    if f(c) = 0.0 || (b - a) / 2.0 < t then
      // Solution found
      c
    else
      if sign(f(c)) = sign(f(a)) then
        bisect (n + 1) N f c b t
      else
        bisect (n + 1) N f a c t
```

Looking at an example

Now, we will look at an example of solving the roots of a quadratic equation. The equation, $x^2 - x - 6$, is plotted in the following figure:



The roots of the preceding quadratic equation can easily be seen in the figure. Otherwise, there are analytical methods of solving it; for example, the method of completing the square. The roots of the equation are -2 and 3.

Next, we create an anonymous function in F# to describe the one that we are interested in solving the roots for:

```
let f = (fun x -> (x**2.0 - x - 6.0))
```

We can test the preceding function using the roots that we found in the preceding figure:

```
> f(-2.0);;  
val it : float = 0.0
```

```
> f(3.0);;  
val it : float = 0.0
```

The results are as expected. Now, we can continue and use the lambda function as an argument to the `bisect` function:

```
// First root, on the positive side  
let first = bisect 0 25 f 0.0 10.0 0.01  
  
// Second root, on the negative side  
let second = bisect 0 25 f -10.0 0.0 0.01
```

The first two arguments, 0 and 25, are used to keep track of the iterations. We pass in 0 because we want to start from iteration 0 and then iterate 25 times. The next argument is the function itself that we defined in the preceding code as `f`. The next two arguments are limits, that is, the range within which we can look for the root. And the last one is just a value for the accuracy used for comparison inside the iteration.

We can now inspect the two variables and see if we find the roots:

```
> first;;  
val it : float = -2.001953125  
  
> second;;  
val it : float = 2.998046875
```

They are almost equal to the analytical solution of -2 and 3 respectively. This is something that is typical for numerical analysis. The solutions will almost never be exact. In every step, some inaccuracy is added due to the floating-point numbers, rounding, and so on.

Finding roots using the Newton–Raphson method

The Newton-Raphson method, or simply Newton's method, usually converges faster than the bisection method. The Newton-Raphson method also needs the derivative of the function, which can be a problem in some cases. This is especially true when there is no analytical solution available. The following implementation is a modification of the bisection method using the derivative of the function to determine if a solution has been found. Let's look at the implementation of Newton's method in F#:

```
// Newton's Method
let rec newtonraphson n N (f:float -> float) (fprime:float ->
float) (x0: float) (tol:float) : float =
  if n >= N then -1.0
  else
    let d = fprime(x0)
    let newtonX = x0 - f(x0) / d
    if abs(d) < tol then
      -1.0
    else
      if abs(newtonX - x0) < tol then
        newtonX // Solution found
      else
        newtonraphson (n +1) N f fprime newtonX tol
```

One of the drawbacks of using the preceding method is that we use a fixed point convergence criteria, `abs(newtonX - x0) < tol`, which means that we can be far from the actual solution when this criteria is met.

Looking at an example

We can now try to find the square root of two, which is expected to be 1.41421. First, we need the function itself, `fun x -> (x**2.0 - 2.0)`. We also need the derivative of the same function, `x -> (2.0*x)`:

```
let f = (fun x -> (x**2.0 - 2.0))
let fprime = (fun x -> (2.0*x))
let sqrtOfTwo = newtonraphson 0 25 f fprime 1.0 10e-10
```

Now, we use the Newton-Raphson method to find the root of the function, $x^2 - 2$. Using F# Interactive, we can investigate this as follows:

```
> newtonraphson 0 25 f fprime 1.0 10e-10;;
val it : float = 1.414213562
```

This is the answer we would expect, and the method works for finding roots! Notice that if we change the starting value, x_0 , from 1.0 to -1.0, we'll get the negative root:

```
> newtonraphson 0 25 f fprime -1.0 10e-10;;  
val it : float = -1.414213562
```

This is also a valid solution to the equation, so be aware of this when you use this method for solving the roots. It can be helpful to plot the function, as we did in the section about the bisection method, to get a grip on where to start from.

Finding roots using the secant method

The secant method, which doesn't need the derivative of the function, is an approximation of the Newton-Raphson method. It uses the finite difference approximation in iterations. The following is a recursive implementation in F#:

```
// Secant method  
let rec secant n N (f:float -> float) (x0:float) (x1:float)  
  (x2:float) : float =  
  if n >= N then x0  
  else  
    let x = x1 - (f(x1))*((x1 - x0)/(f(x1) - f(x0)))  
    secant (n + 1) N f x x0
```

Looking at an example

Let's look at an example where we use the secant method to find one root of a function. We'll try to find the positive root of 612, which is a number just under the square of 25 ($25 \times 25 = 625$):

```
let f = (fun x -> (x**2.0 - 612.0))  
  
> secant 0 10 f 0.0 10.0 30.0;;  
val it : float = 24.73863375
```

Summary

In this chapter, we looked deeper into F# and numerical analysis and how well the two fit together because of the functional syntax of the language. We covered algorithm implementation, basic numerical concerns, and the Math.NET library. After reading this chapter, you will be more familiar with both F# and numerical analysis and be able to implement algorithms yourself. At the end of the chapter, an example using the bisection method was covered. This method will prove to be very useful later on when we talk about Black-Scholes and implied volatility.

In the next chapter, we will build on what we learned so far and extend our current knowledge with data visualization, basic GUI programming, and plotting financial data.

4

Getting Started with Data Visualization

In this chapter, you will learn how to get started with data visualization and build graphical user interfaces (GUIs) in F#. In quantitative finance, it is essential to be able to plot and visualize time series. F# is a great tool for this and we'll learn how to use F# as an advanced graphical calculator using F# Interactive.

The content in this chapter will be used throughout the book whenever user interfaces are needed. In this chapter you will learn:

- Programming of basic GUI in F# and .NET
- Plotting data using Microsoft Charts
- Plotting financial data
- Building an interactive GUI

Making your first GUI in F#

F# leverages the .NET platform and GUI programming is no exception. All classes from the .NET platform are available in this section. We will concentrate on the one from the `System.Windows.Forms` namespace.

It's possible to use the same code from F# Interactive and modify the GUI on the fly. We will look at this in more detail in the *Displaying data* section.

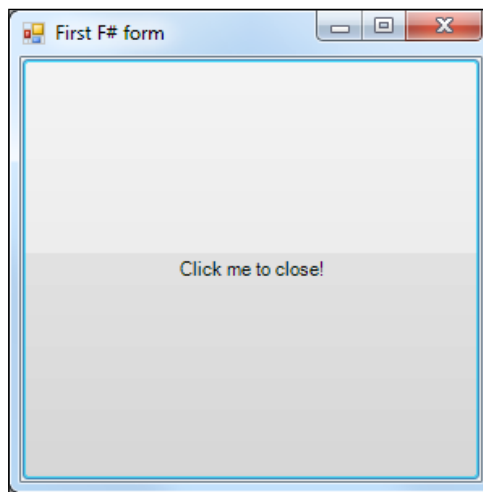
Let's look at an example where we make use of a .NET form and a button. The button will be connected to an event handler called for every click on the button. As you can see when you read the code, event handlers are higher-order functions that result in a clean and compact code.


```
open System.Windows.Forms

let form = new Form(Text = "First F# form")
let button = new Button(Text = "Click me to close!", Dock =
DockStyle.Fill)

button.Click.Add(fun _ -> Application.Exit() |> ignore)
form.Controls.Add(button)
form.Show()
```

The screenshot for the output of the preceding code is as follows:



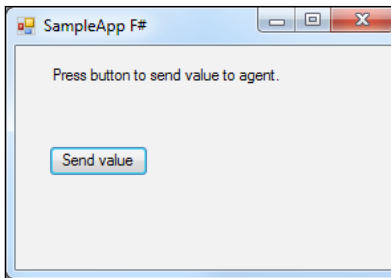
The first GUI application in F# consisting of a form and a button

Composing interfaces

Now, we have looked at the first code to generate a form and composed a very simple interface consisting of a button. As you may have noticed, F# has no visual designer as present in the other .NET languages. There are several ways of composing interfaces in F#:

- Writing interface code manually
- Using the C# visual designer and converting the code into F#
- Building a library using other .NET language and using it from F#
- Building your own visual designer to output F# code

In this book, we will mainly use the first alternative – writing interface code manually. This may seem tedious, but the upside is total control over the layout. We'll now look at a larger example using an agent to keep track of the highest number and a user interface with a button. When a user clicks on that button, a random number is sent to the agent (see the following screenshot). The agent then outputs the highest number every second. Also, the example illustrates the use of namespaces and modules in a realistic fashion. This gives the reader an idea of when to use namespaces and modules, and how to structure the programs when they become larger.

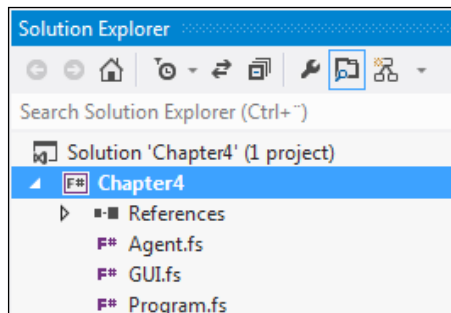


The form of the agent application, with a button to send values to the agent.

The order of the files in the project is as follows:

- Agent
- GUI
- Program

Otherwise, you will see some errors due to references. See the following figure showing **Solution Explorer** and notice the order:



More about agents

First we start with the agent. The agent is much the same as the agent in the section about agents in *Chapter 2, Learning More About F#*, except for some modifications and the namespace `Agents`. The code is as follows:

```
namespace Agents

    open System

    // Type for our agent
    type Agent<'T> = MailboxProcessor<'T>

    // Control messages to be sent to agent
    type CounterMessage =
        | Update of float
        | Reset

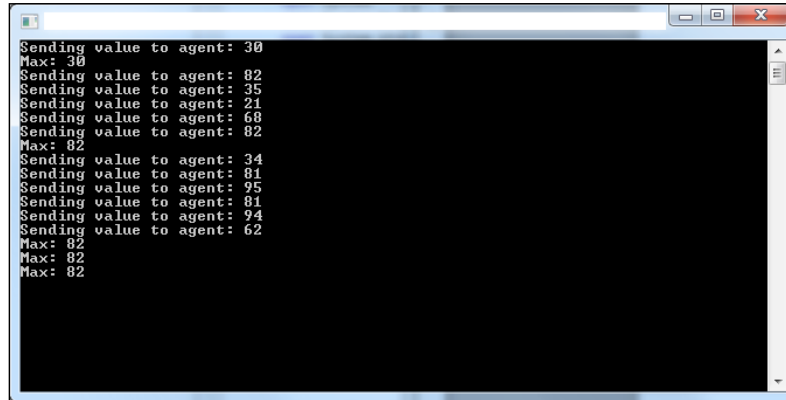
    module Helpers =
        let genRandomNumber (n) =
            let rnd = new System.Random()
            float (rnd.Next(n, 100))

    module MaxAgent =
        // Agent to keep track of max value and update GUI
        let sampleAgent = Agent.Start(fun inbox ->
            let rec loop m = async {
                let! msg = inbox.Receive()
                match msg with
                | Reset ->
                    return! loop 0.0
                | Update value ->
                    let m = max m value

                    Console.WriteLine("Max: " + m.ToString())

                    do! Async.Sleep(1000)
                    return! loop m
            }
            loop 0.0)
```

The screenshot for the preceding code is as follows:



```
Sending value to agent: 30
Max: 30
Sending value to agent: 82
Sending value to agent: 35
Sending value to agent: 21
Sending value to agent: 68
Sending value to agent: 82
Max: 82
Sending value to agent: 34
Sending value to agent: 81
Sending value to agent: 95
Sending value to agent: 81
Sending value to agent: 94
Sending value to agent: 62
Max: 82
Max: 82
Max: 82
```

The console window with the output from the agent

The user interface

The user interface is placed in the GUI namespace. `SampleForm` inherits from the `Form` .NET class. If you are familiar with other .NET languages, you will see some of the common steps involved. All the layout code is also a part of the code. There is no visual designer in F#, as mentioned earlier. To use `System.Windows.Forms`, you have to add a reference to the assembly with the same name. The code is as follows:

```
namespace GUI

    open System
    open System.Drawing
    open System.Windows.Forms
    open Agents

    // User interface form
    type public SampleForm() as form =
        inherit Form()

        let valueLabel = new Label(Location=new Point(25,15))
        let startButton = new Button(Location=new Point(25,50))
        let sendButton = new Button(Location=new Point(25,75))
        let agent = MaxAgent.sampleAgent
```

```
let initControls =
    valueLabel.Text <- "Sample Text"
    startButton.Text <- "Start"
    sendButton.Text <- "Send value to agent"
do
    initControls

    form.Controls.Add(valueLabel)
    form.Controls.Add(startButton)

    form.Text <- "SampleApp F#"

    startButton.Click.AddHandler(new System.EventHandler
        (fun sender e ->
            form.eventStartButton_Click(sender, e)))

// Event handler(s)
member form.eventStartButton_Click(sender:obj,
e:EventArgs) =
    let random = Helpers.genRandomNumber 5
    Console.WriteLine("Sending value to agent: " +
        random.ToString())
    agent.Post(Update random)
    ()
```

The main application

This is the main application entry point. It is annotated to tell the runtime environment (.NET platform) from where to start out. This is done by using the [`<STAThread>`] annotation. Here, we simply kick off the application and its GUI. The code for `SampleForm` is as follows:

```
// Main application entry point
namespace Program

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

    open GUI

    module Main =
        [<STAThread>]
        do
```

```
Application.EnableVisualStyles()
Application.SetCompatibleTextRenderingDefault(false)
let view = new SampleForm()
Application.Run(view)
```

Learning about event handling

Event driven programming and events from user are common ways of building GUIs. Event handlers are easy in F#, and lambda functions are easy to read and understand. Compact code is always preferable and makes things such as maintenance and understanding of the code easier for everyone involved.

If we look more closely at the code previously used for event handling, you can see that we first use a lambda function and inside the lambda we call a member function of the class:

```
startButton.Click.AddHandler(new System.EventHandler
    (fun sender e ->
      form.eventStartButton_Click(sender, e)))

// Event handler(s)
member form.eventStartButton_Click(sender:obj,
e:EventArgs) =
    let random = Helpers.genRandomNumber 5
    Console.WriteLine("Sending value to agent: " +
      random.ToString())
    agent.Post(Update random)
    ()
```

This is just a way of making the code more readable and easier to understand. It's of course possible to include all the logic in the lambda function directly; but this approach is leaner, especially for larger projects.

Displaying data

Displaying and visualizing data is essential to get a better understanding of its characteristics. Also, data is at its core in quantitative finance. F# is a sharp tool for data analysis and visualization. A majority of the functionalities of visualization and user interfaces comes from the .NET platform. Together with the exploratory characteristics of F#, especially through F# Interactive, the combination becomes very efficient and powerful.

Let's start out by using F# Interactive to create a form that will display data feed to it. This means we will have a form that can change the content at runtime, without the need for recompiling. The controls in the form are also interchangeable:

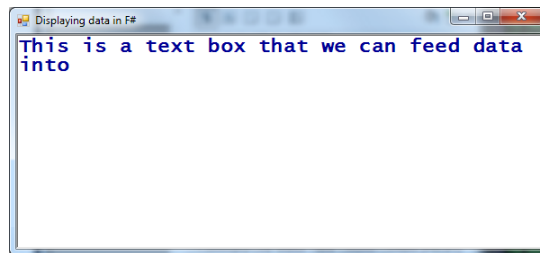
```
// The form
open System
open System.Drawing
open System.Windows.Forms

let form = new Form(Visible = true, Text = "Displaying data in F#",
  TopMost = true, Size = Drawing.Size(600,600))

let textBox =
  new RichTextBox(Dock = DockStyle.Fill, Text = "This is a text
  box that we can feed data into", Font = new Font("Lucida
  Console",16.0f,FontStyle.Bold), ForeColor = Color.DarkBlue)

form.Controls.Add textBox
```

If you run this code, you will see a form with the title **Displaying data in F#** as the one shown in the following screenshot:



The window with a RichTextBox control to display data

We need a function to send data to the textbox in the window and display it. Here is the one to do the job for us:

```
let show x =
  textBox.Text <- sprintf "%30A" x
```

Now, we can use the function and it will send the formatted data to our textbox (textBox). Here are some examples that show you how to use the function; it's useful to utilize the pipe function as illustrated in the later snippets:

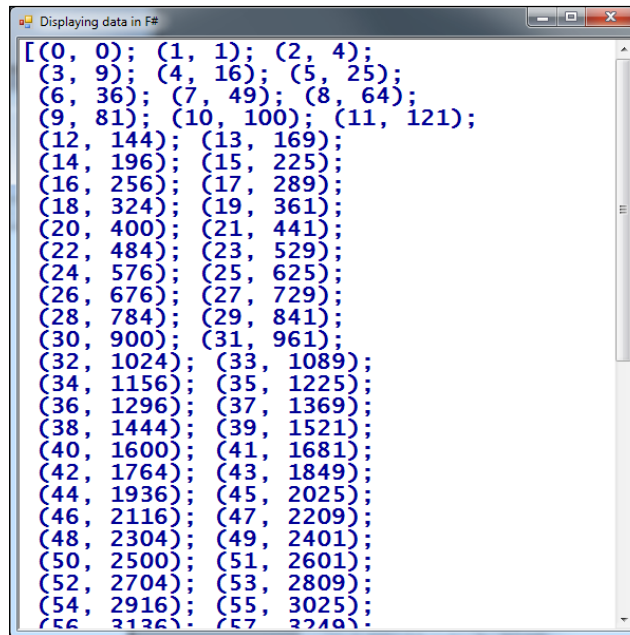
```
show (1,2)
show [ 0 .. 100 ]
show [ 0.0 .. 2.0 .. 100.0 ]
```

```
// Using the pipe operator
(1,2,3) |> show
[ 0 .. 99 ] |> show
[ for i in 0 .. 99 -> (i, i*i) ] |> show
```

If you want to clear the contents of the textbox, you can write:


```
textBox.Clear()
```

The output is as follows:



```
[(0, 0); (1, 1); (2, 4);
(3, 9); (4, 16); (5, 25);
(6, 36); (7, 49); (8, 64);
(9, 81); (10, 100); (11, 121);
(12, 144); (13, 169);
(14, 196); (15, 225);
(16, 256); (17, 289);
(18, 324); (19, 361);
(20, 400); (21, 441);
(22, 484); (23, 529);
(24, 576); (25, 625);
(26, 676); (27, 729);
(28, 784); (29, 841);
(30, 900); (31, 961);
(32, 1024); (33, 1089);
(34, 1156); (35, 1225);
(36, 1296); (37, 1369);
(38, 1444); (39, 1521);
(40, 1600); (41, 1681);
(42, 1764); (43, 1849);
(44, 1936); (45, 2025);
(46, 2116); (47, 2209);
(48, 2304); (49, 2401);
(50, 2500); (51, 2601);
(52, 2704); (53, 2809);
(54, 2916); (55, 3025);
(56, 3136); (57, 3249);
```

This is how the form looks with the content generated from the previous snippet

 Try this out for yourself and see which work flow is best suited for you.

Extending the form to use a table

Now that we have looked at how to use F# Interactive and feed data to a form on the fly, we can extend the concept and use a table as follows:

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

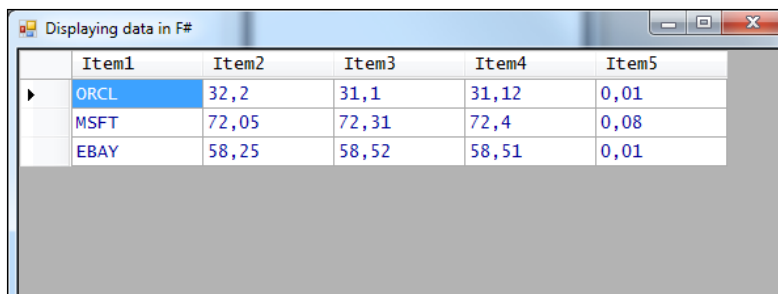
// The form
let form2 = new Form(Visible = true, Text = "Displaying data in F#",
    TopMost = true, Size = Drawing.Size(600,600))

// The grid
let data = new DataGridView(Dock = DockStyle.Fill, Text = "Data
    grid", Font = new Drawing.Font("Lucida Console", 10.0f), ForeColor
    = Drawing.Color.DarkBlue)

form2.Controls.Add(data)

// Some data
data.DataSource <- [| ("ORCL", 32.2000, 31.1000, 31.1200, 0.0100);
    ("MSFT", 72.050, 72.3100, 72.4000, 0.0800);
    ("EBAY", 58.250, 58.5200, 58.5100, 0.0100) |]
```

The preceding code will add `DataGridView` to the form, with some styling added to it. The last lines of code will populate `DataGridView` with some sample data. It will finally look something like the following figure:



	Item1	Item2	Item3	Item4	Item5
▶	ORCL	32,2	31,1	31,12	0,01
	MSFT	72,05	72,31	72,4	0,08
	EBAY	58,25	58,52	58,51	0,01

DataGridView added to the form with sample data

Let's extend the example and use code to set the column headers together with code to use a collection:

```
open System
open System.Drawing
open System.Windows.Forms
open System.Collections.Generic

// The form
let form2 = new Form(Visible = true, Text = "Displaying data in F#",
    TopMost = true, Size = Drawing.Size(600,600))

// The grid
let data = new DataGridView(Dock = DockStyle.Fill, Text = "Data grid",
    Font = new Drawing.Font("Lucida Console", 10.0f), ForeColor =
    Drawing.Color.DarkBlue)

form2.Controls.Add(data)

// Generic list
let myList = new List<(string * float * float * float * float)>()

// Sample data
myList.Add(("ORCL", 32.2000, 31.1000, 31.1200, 0.0200))
myList.Add(("MSFT", 72.050, 72.3100, 72.4000, 0.0100))

data.DataSource <- myList.ToArray()

// Set column headers
do data.Columns.[0].HeaderText <- "Symb"
do data.Columns.[1].HeaderText <- "Last sale"
do data.Columns.[2].HeaderText <- "Bid"
do data.Columns.[3].HeaderText <- "Ask"
do data.Columns.[4].HeaderText <- "Spread"

do data.Columns.[0].Width <- 100
```

The result will look something like the window in the following screenshot:



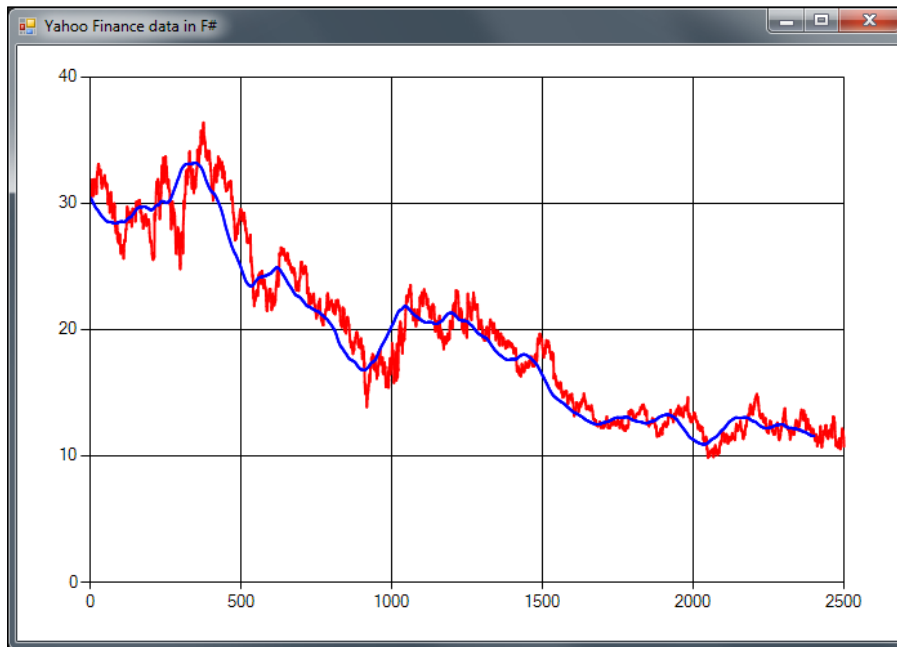
	Symb	Last sale	Bid	Ask	Spread
▶	ORCL	32,2	31,1	31,12	0,02
	MSFT	72,05	72,31	72,4	0,01
	MSFT	72,05	72,31	72,4	0,01
	ORCL	32,2	31,1	31,12	0,02

A formatted DataGridView using a collection as data source

Displaying financial data from Yahoo! Finance

Now we will look at a bigger example application, where we will use the concepts introduced this far and extend the functionality to cover visualization of financial data. Here, we will download data from Yahoo! Finance and display the closing prices together with a calculated moving average in the same chart window.

The application will finally look something like the one in the following screenshot:



An example application to visualize data form Yahoo! Finance

Understanding the application code

The application will use some code introduced in previous sections, as well as in previous chapters. If you don't recognize any of the content, please go back and refresh on that particular topic. The main building block here is `Systems.Windows.Forms` and `System.Windows.Forms.DataVisualization.Charting`. A lot more information is available online at MSDN: <http://msdn.microsoft.com/en-us/library/system.windows.forms.datavisualization.charting.aspx>.

Let's look at the code needed to provide the preceding functionality:

```
#r "System.Windows.Forms.DataVisualization.dll"

open System
open System.Net
open System.Windows.Forms
open System.Windows.Forms.DataVisualization.Charting
open Microsoft.FSharp.Control.WebExtensions
```

We'll first create a chart and initialize it by setting `style` and `ChartAreas`:

```
// Create chart and form
let chart = new Chart(Dock = DockStyle.Fill)
let area = new ChartArea("Main")
chart.ChartAreas.Add(area)
```

Then the form is created and displayed. After that, the title of the program is set and the chart control is added to the form:

```
let mainForm = new Form(Visible = true, TopMost = true,
                        Width = 700, Height = 500)

do mainForm.Text <- "Yahoo Finance data in F#"
mainForm.Controls.Add(chart)
```

Then, there is some code to create the two charting series needed and some style is set to the two charting series to distinguish them from each other. The stock price series will be red, and the moving average will be blue:

```
// Create series for stock price
let stockPrice = new Series("stockPrice")
do stockPrice.ChartType <- SeriesChartType.Line
do stockPrice.BorderWidth <- 2
do stockPrice.Color <- Drawing.Color.Red
chart.Series.Add(stockPrice)
```

```
// Create series for moving average
let movingAvg = new Series("movingAvg")
do movingAvg.ChartType <- SeriesChartType.Line
do movingAvg.BorderWidth <- 2
do movingAvg.Color <- Drawing.Color.Blue
chart.Series.Add(movingAvg)

// Synchronous fetching (just one stock here)
```

Now, the code for the fetching of data is the same as we used in the previous chapter, *Chapter 3, Financial Mathematics and Numerical Analysis*.

```
let fetchOne() =
    let uri = new
        System.Uri("http://ichart.finance.yahoo.com/table.csv?
            s=ORCL&d=9&e=23&f=2012&g=d&a=2&b=13&c=1986&ignore=.csv")
    let client = new WebClient()
    let html = client.DownloadString(uri)
    html

// Parse CSV
let getPrices() =
    let data = fetchOne()
    data.Split('\n')
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> float s.[4])
    |> Seq.truncate 2500
```

The interesting part here is how data is added to the chart. This is done by iterating the time series and using the `series.Points.Add` method. It's an elegant and concise way of doing it. The `ignore` operator is used to simply skip the resulting value from the `Points.Add` method, ignoring it.

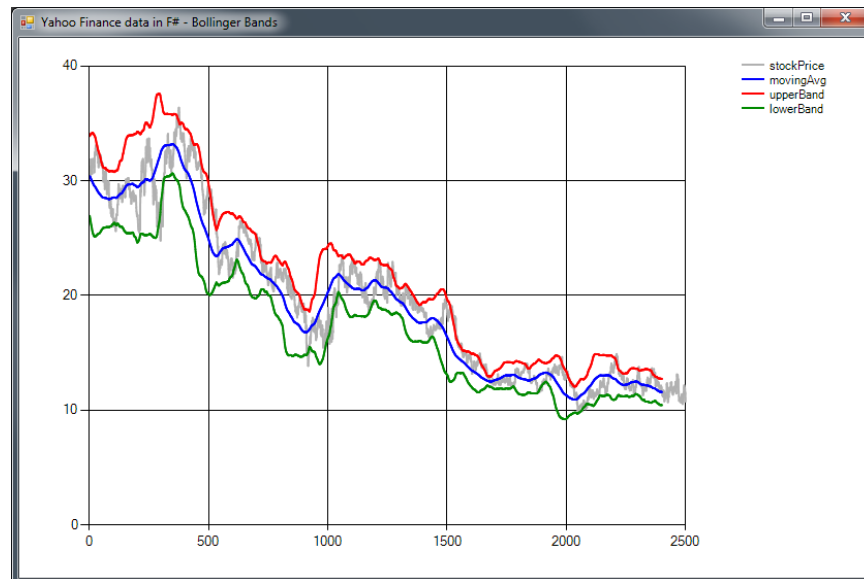
```
// Calc moving average
let movingAverage n (prices:seq<float>) =
    prices
    |> Seq.windowed n
    |> Seq.map Array.sum
    |> Seq.map (fun a -> a / float n)

// The plotting
let sp = getPrices()
do sp |> Seq.iter (stockPrice.Points.Add >> ignore)

let ma = movingAverage 100 sp
do ma |> Seq.iter (movingAvg.Points.Add >> ignore)
```

Extending the application to use Bollinger bands

We'll now extend the application we used in the previous section to use Bollinger bands. Bollinger bands is an extension of moving average, where two bands are added – one upper band and one lower band. The bands are typically K (where $K=2.0$) times a moving standard deviation above and below the moving average. We need to add a function to calculate the moving standard deviation. We can use the standard deviation from the previous chapter and use it with the `Seq.windowed` function, as shown in the following code. In this example, we also add legends to specify which data series corresponds to which color. The screenshot is as follows:



Example application extended to use Bollinger Bands

The code looks pretty much like the code used in the preceding example; except for the calculation of the upper and lower band, and the moving standard deviation:

```
/// Another example with Bollinger Bands
#r "System.Windows.Forms.DataVisualization.dll"

open System
open System.Net
open System.Windows.Forms
open System.Windows.Forms.DataVisualization.Charting
open Microsoft.FSharp.Control.WebExtensions
```

```
// Create chart and form
let chart = new Chart(Dock = DockStyle.Fill)
let area = new ChartArea("Main")
chart.ChartAreas.Add(area)
```

Legends are easily added to a chart using `chart.Legends.Add`:

```
// Add legends
chart.Legends.Add(new Legend())

let mainForm = new Form(Visible = true, TopMost = true,
                        Width = 700, Height = 500)

do mainForm.Text <- "Yahoo Finance data in F# - Bollinger Bands"
mainForm.Controls.Add(chart)

// Create series for stock price
let stockPrice = new Series("stockPrice")
do stockPrice.ChartType <- SeriesChartType.Line
do stockPrice.BorderWidth <- 2
do stockPrice.Color <- Drawing.Color.DarkGray
chart.Series.Add(stockPrice)

// Create series for moving average
let movingAvg = new Series("movingAvg")
do movingAvg.ChartType <- SeriesChartType.Line
do movingAvg.BorderWidth <- 2
do movingAvg.Color <- Drawing.Color.Blue
chart.Series.Add(movingAvg)
```

We'll need two new data series for our upper and lower bands respectively:

```
// Create series for upper band
let upperBand = new Series("upperBand")
do upperBand.ChartType <- SeriesChartType.Line
do upperBand.BorderWidth <- 2
do upperBand.Color <- Drawing.Color.Red
chart.Series.Add(upperBand)

// Create series for lower band
let lowerBand = new Series("lowerBand")
do lowerBand.ChartType <- SeriesChartType.Line
do lowerBand.BorderWidth <- 2
do lowerBand.Color <- Drawing.Color.Green
chart.Series.Add(lowerBand)
```

```

// Synchronous fetching (just one stock here)
let fetchOne() =
    let uri = new
        System.Uri("http://ichart.finance.yahoo.com/table.csv?
s=ORCL&d=9&e=23&f=2012&g=d&a=2&b=13&c=1986&ignore=.csv")
    let client = new WebClient()
    let html = client.DownloadString(uri)
    html

// Parse CSV
let getPrices() =
    let data = fetchOne()
    data.Split('\n')
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> float s.[4])
    |> Seq.truncate 2500

// Calc moving average
let movingAverage n (prices:seq<float>) =
    prices
    |> Seq.windowed n
    |> Seq.map Array.sum
    |> Seq.map (fun a -> a / float n)

```

The code to calculate the moving standard deviation is a modification of the code used in the previous chapter, to work with the `Seq.windowed` function:

```

// Stddev
let stddev2(values:seq<float>) =
    let avg = Seq.average values
    values
    |> Seq.fold (fun acc x -> acc + (1.0 / float (Seq.length
values)) * (x - avg) ** 2.0) 0.0
    |> sqrt

let movingStdDev n (prices:seq<float>) =
    prices
    |> Seq.windowed n
    |> Seq.map stddev2

// The plotting
let sp = getPrices()
do sp |> Seq.iter (stockPrice.Points.Add >> ignore)

```



```
let ma = movingAverage 100 sp
do ma |> Seq.iter (movingAvg.Points.Add >> ignore)
```

This section is pretty interesting. Here, we add and subtract the result from the moving standard deviation to the moving average, which is multiplied with the K-value to form the upper and lower band:

```
// Bollinger bands, K = 2.0
let ub = movingStdDev 100 sp
// Upper
Seq.zip ub ma |> Seq.map (fun (a,b) -> b + 2.0 * a) |> Seq.iter
  (upperBand.Points.Add >> ignore)
// Lower
Seq.zip ub ma |> Seq.map (fun (a,b) -> b - 2.0 * a) |> Seq.iter
  (lowerBand.Points.Add >> ignore)
```

You can extend this application and implement other technical indicators if you like. The nice thing about using F# Interactive is that the application itself doesn't have to be restarted to show new data. In other words, you can use `movingAvg.Points.Add` and the point will be added to the chart.

Using FSharp.Charting

FsChart is a commonly used F# chart library implemented as a functional wrapper over the Microsoft Chart Control. This control can save you some work because there is no need for boilerplate code as in the preceding examples for Microsoft Chart Control. **FsChart** is also designed to work with F# and integrate better with F# Interactive.

The library can be installed using the Package Manager Console by typing:

```
Install-Package FSharp.Charting
```

Creating a candlestick chart from stock prices

Let's look at the code for displaying a candlestick chart of the same stock as used before (Oracle) with data from Yahoo! Finance. This time there is less boilerplate code needed to set up the charting. The main part of the program consists of downloading, parsing, and converting the data:

```

open System
open System.Net
open FSharp.Charting
open Microsoft.FSharp.Control.WebExtensions
open System.Windows.Forms.DataVisualization.Charting

```

To use `FSharp.Charting`, first we need to set up the chart as follows:

```

module FSharpCharting =
    fsi.AddPrinter(fun (ch:FSharp.Charting.ChartTypes.GenericChart) ->
        ch.ShowChart();
        "FSharpCharting")

    // Synchronous fetching (just one stock here)
    let fetchOne() =
        let uri = new
            System.Uri("http://ichart.finance.yahoo.com/table.csv?
            s=ORCL&d=9&e=23&f=2012&g=d&a=2&b=13&c=2012&ignore=.csv")
        let client = new WebClient()
        let html = client.DownloadString(uri)
        html

```

We need to reorder the data from open, high, low close to high, low, and open close. This is done when we parse the strings to floating point numbers. Also, we include the date as the first value. The date will be used by `FSharpCharts` to order the candles.

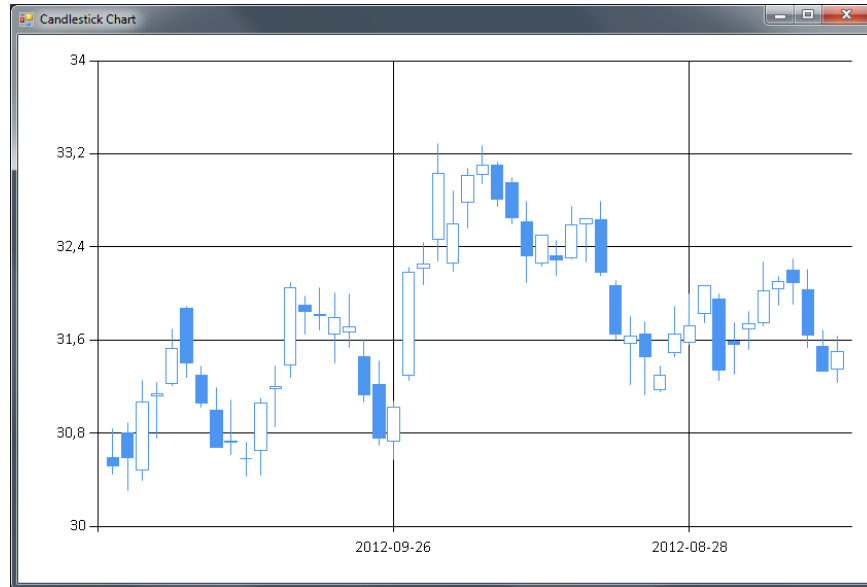
```

// Parse CSV and re-arrange O,H,L,C - > H,L,O,C
let getOHLCPrices() =
    let data = fetchOne()
    data.Split('\n')
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> s.[0], float s.[2], float s.[3], float
    s.[1], float s.[4])
    |> Seq.truncate 50

// Candlestick chart price range specified
let ohlcPrices = getOHLCPrices() |> Seq.toList
Chart.Candlestick(ohlcPrices).WithYAxis(Max = 34.0, Min = 30.0)

```

The data will be downloaded, parsed, and displayed in the chart and the final result will look something like the following screenshot:



Using FSharpCharts to display a candlestick chart

Creating a bar chart

In this example, we'll learn how to plot a histogram of a distribution generated by Math.NET. Histograms are useful to visualize statistical data and get a grip of their properties. We'll use a simple normal distribution with a zero mean and a standard deviation of one.

```
open System
open MathNet.Numerics
open MathNet.Numerics.Distributions
open MathNet.Numerics.Statistics
open FSharp.Charting

module FSharpCharting2 =
    fsi.AddPrinter(fun ch:FSharp.Charting.ChartTypes.GenericChart)
    -> ch.ShowChart(); "FSharpCharting")
```

Next we'll create the normal distribution that will be used in the histogram:

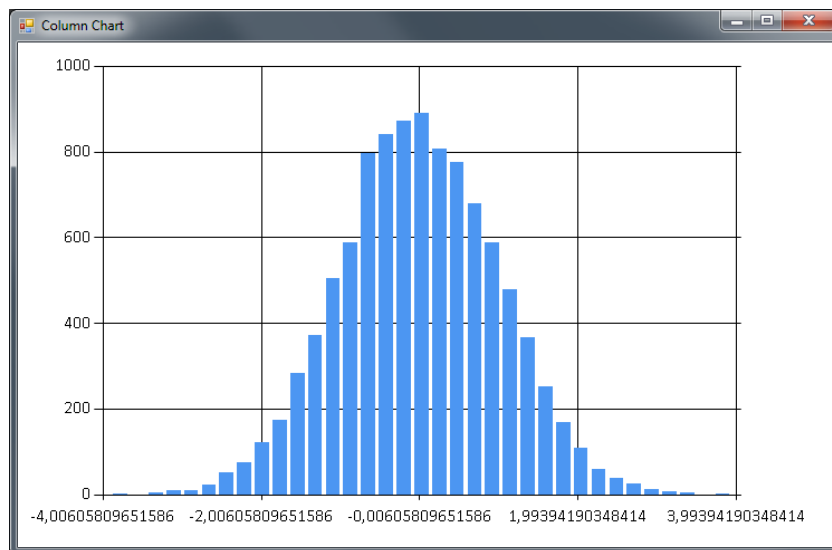
```
let dist = new Normal(0.0, 1.0)
let samples = dist.Samples() |> Seq.take 10000 |> Seq.toList
let histogram = new Histogram(samples, 35);
```

Unfortunately, Math.NET and FSharpCharting is not compatible out of the box. We need to convert the histogram from Math.NET to use it with the `Chart.Column` function:

```
let getValues =
    let bucketWidth = Math.Abs(histogram.LowerBound -
        histogram.UpperBound) / (float histogram.BucketCount)
    [0..(histogram.BucketCount-1)]
    |> Seq.map (fun i -> (histogram.Item(i).LowerBound +
        histogram.Item(i).UpperBound)/2.0, histogram.Item(i).Count)
```

```
Chart.Column getValues
```

As you can see in the following screenshot, the distribution looks a lot like the normal distribution. You can experiment with the number of buckets yourself to see how this behavior will change with the number of buckets. Also, you can increase the number of samples used.



Using FSharpCharts to display a histogram

Summary

In this chapter, we looked at data visualization in F# and learned to build user interfaces. We have looked at how to use F# to create user interfaces without a visual designer. There are pros and cons of using this approach of course. The main benefit is total control and there is no hidden magic. On the other side, it can be time consuming when we talk about larger GUI applications.

In the next chapter, we'll use the data visualization tools introduced in this chapter to study some interesting properties of options. We'll talk about the basics of the options and how they are priced using the Black Scholes formula. Also, the Black Scholes formula will be implemented in F# and discussed in detail.

5

Learning Option Pricing

In this chapter, you will learn how to get started with option pricing using the Black-Scholes formula and the Monte Carlo method. We'll compare the two methods and see where they are most suitable in real-world applications.

In this chapter you will learn:

- The Black-Scholes option pricing formula
- How to use the Monte Carlo method to price options
- European, American, and Exotic options
- How to use real market data from Yahoo! Finance in option pricing
- Plotting the greeks in F#
- The basics of Wiener processes and the Brownian motion
- The basics of stochastic differential equations

Introduction to options

Options come in two variants, puts and calls. The call option gives the owner of the option the right, but not the obligation, to buy the underlying asset at the strike price. The put option gives the holder of the contract the right, but not the obligation, to sell the underlying asset. The Black-Scholes formula describes the European option, which can only be exercised on the maturity date, in contrast to, for example, American options. The buyer of the option pays a premium for this in order to cover the risk taken from the counterpart's side. Options have become very popular and they are used in the major exchanges throughout the world, covering most asset classes.

The theory behind options can become complex pretty quickly. In this chapter, we'll look at the basics of options and how to explore them using the code written in F#.

Looking into contract specifications

Options come in a wide number of variations, some of which will be covered briefly in this section. The contract specifications for options will also depend on their type. Generally, there are some properties that are more or less general to all of them. The general specifications are as follows:

- Side
- Quantity
- Strike price
- Expiration date
- Settlement terms

The contract specifications, or known variables, are used when we value options.

European options

European options are the basic form of options that other types of options extend. American options and Exotic options are some examples. We'll stick to European options in this chapter.

American options

American options are options that may be exercised on any trading day on or before expiry.

Exotic options

Exotic options are options belonging to the broad category of options, which may include complex financial structures, and may be combinations of other instruments as well.

Learning about Wiener processes

Wiener processes are closely related to stochastic differential equations and volatility. A Wiener process, or the geometric Brownian motion, is defined as follows:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

The preceding formula describes the change in the stock price or the underlying asset with a drift, μ , and a volatility, σ , and the Wiener process, Wt . This process is used to model the prices in Black-Scholes.

We'll simulate market data using a Brownian motion or a Wiener process implemented in F# as a sequence. Sequences can be infinite and only the values used are evaluated, which suits our needs. We'll implement a generator function to generate the Wiener process as a sequence as follows:

```
// A normally distributed random generator
let normd = new Normal(0.0, 1.0)
let T = 1.0
let N = 500.0
let dt:float = T / N

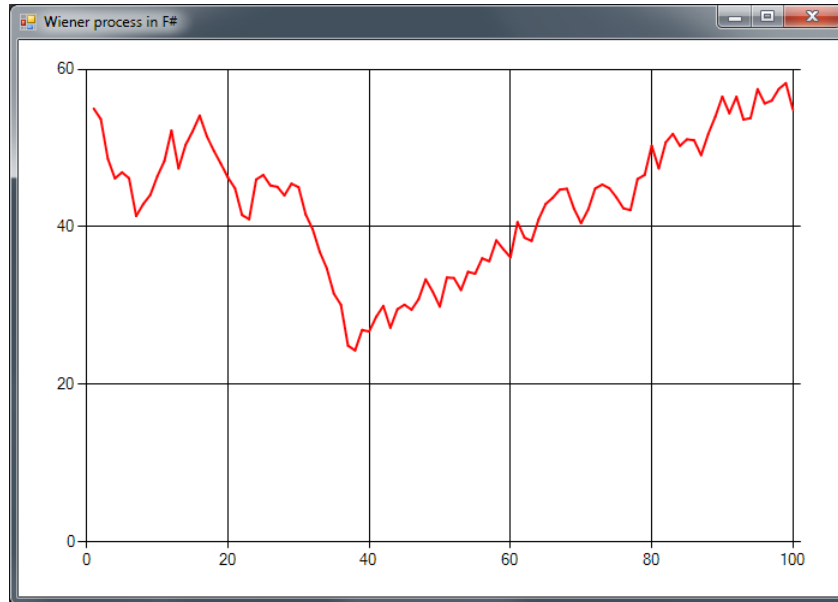
/// Sequences represent infinite number of elements
// p -> probability mean
// s -> scaling factor
let W s = let rec loop x = seq { yield x; yield! loop (x +
    sqrt(dt)*normd.Sample()*s) }
    loop s;;
```

Here, we use the random function in `normd.Sample()`. Let's explain the parameters and the theory behind the Brownian motion before looking at the implementation. The parameter `T` is the time used to create a discrete time increment `dt`. Notice that `dt` will assume there are 500 `N:s`, 500 items in the sequence; this of course is not always the case, but it will do fine here. Next, we use recursion to create the sequence, where we add an increment to the previous value ($x+\dots$), where x corresponds to $xt-1$.

We can easily generate an arbitrary length of the sequence as follows:

```
> Seq.take 50 (W 55.00);;
val it : seq<float> = seq [55.0; 56.72907873; 56.96071054;
    58.72850048; ...]
```


Here, we create a sequence of length 50. Let's plot the sequence to get a better understanding about the process as shown in the following screenshot:



A Wiener process generated from the preceding sequence generator

Next, we'll look at the following code to generate the graph shown in the preceding screenshot:

```
open System
open System.Net
open System.Windows.Forms
open System.Windows.Forms.DataVisualization.Charting
open Microsoft.FSharp.Control.WebExtensions
open MathNet.Numerics.Distributions;

// A normally distributed random generator
let normald = new Normal(0.0, 1.0)

// Create chart and form
let chart = new Chart(Dock = DockStyle.Fill)
let area = new ChartArea("Main")
chart.ChartAreas.Add(area)

let mainForm = new Form(Visible = true, TopMost = true,
                        Width = 700, Height = 500)
```

```

do MainForm.Text <- "Wiener process in F#"
MainForm.Controls.Add(chart)

// Create series for stock price
let wienerProcess = new Series("process")
do wienerProcess.ChartType <- SeriesChartType.Line
do wienerProcess.BorderWidth <- 2
do wienerProcess.Color <- Drawing.Color.Red
chart.Series.Add(wienerProcess)

let random = new System.Random()
let rnd() = random.NextDouble()
let T = 1.0
let N = 500.0
let dt:float = T / N

/// Sequences represent infinite number of elements
let W s = let rec loop x = seq { yield x; yield! loop (x +
    sqrt(dt)*normd.Sample()*s) }
    loop s;;

do (Seq.take 100 (W 55.00)) |> Seq.iter (wienerProcess.Points.Add
    >> ignore)

```

Most of the code will be familiar to you at this stage, but the interesting part is the last line, where we can simply feed a chosen number of elements from the sequence into `Seq.iter`, which will plot the values elegantly and efficiently.

Learning the Black-Scholes formula

The Black-Scholes formula was developed by *Fischer Black* and *Myron Scholes* in the 1970s. The Black-Scholes formula is a stochastic partial differential equation which estimates the price of an option. The main idea behind the formula is the delta neutral portfolio. They created the theoretical delta neutral portfolio to reduce the uncertainty involved.

This was a necessary step to be able to come to the analytical formula, which we'll cover in this section. The following are the assumptions made under the Black-Scholes formula:

- No arbitrage
- Possible to borrow money at a constant risk-free interest rate (throughout the holding of the option)

- Possible to buy, sell, and shortlist fractional amounts of underlying assets
- No transaction costs
- Price of the underlying asset follows a Brownian motion, constant drift, and volatility
- No dividends paid from underlying security

The simplest of the two variants is the one for call options. First, the stock price is scaled using the cumulative distribution function with d_1 as a parameter. Then, the stock price is reduced by the discounted strike price scaled by the cumulative distribution function of d_2 . In other words, it's the difference between the stock price and the strike price using probability scaling of each and discounting the strike price:

$$C(S,t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

The formula for the put option is a little more involved, but follows the same principles, shown as follows:

$$\begin{aligned} P(S,t) &= Ke^{-r(T-t)} - S + C(S,t) \\ &= N(-d_2)Ke^{-r(T-t)} - N(-d_1)S \end{aligned}$$

The Black-Scholes formula is often separated into parts, where d_1 and d_2 are the probability factors describing the probability of the stock price being related to the strike price:

$$\begin{aligned} d_1 &= \frac{1}{\sigma\sqrt{T-t}} \left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t) \right] \\ d_2 &= \frac{1}{\sigma\sqrt{T-t}} \left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t) \right] \\ &= d_1 - \sigma\sqrt{T-t} \end{aligned}$$

The parameters used in the preceding formula can be summarized as follows:

- **N**: The cumulative distribution function
- **T**: Time to maturity, expressed in years
- **S**: The stock price or other underlying assets
- **K**: The strike price
- **r**: The risk-free interest rate
- **σ** : The volatility of the underlying assets

Implementing Black-Scholes in F#

Now that we've looked at the basics behind the Black-Scholes formula and the parameters involved, we can implement it ourselves. The cumulative distribution function is implemented here to avoid dependencies and to illustrate that it's quite simple to implement it yourself too. The Black-Scholes formula is implemented in F# by using the following code. It takes six arguments; the first is a call-put-flag that determines if it's a call or put option. The constants a1 to a5 are the Taylor series coefficients used in the approximation for the numerical implementation:

```

let pow x n = exp(n * log(x))

type PutCallFlag = Put | Call

/// Cumulative distribution function
let cnd x =
    let a1 = 0.31938153
    let a2 = -0.356563782
    let a3 = 1.781477937
    let a4 = -1.821255978
    let a5 = 1.330274429
    let pi = 3.141592654
    let l = abs(x)
    let k = 1.0 / (1.0 + 0.2316419 * l)
    let w = (1.0-1.0/sqrt(2.0*pi))*exp(-
        1*1/2.0)*(a1*k+a2*k*k+a3*(pow k 3.0)+a4*(pow k 4.0)+a5*(pow k
        5.0)))
    if x < 0.0 then 1.0 - w else w

/// Black-Scholes
// call_put_flag: Put | Call
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes call_put_flag s x t r v =
    let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
    let d2=d1-v*sqrt(t)
    //let res = ref 0.0

    match call_put_flag with
    | Put -> x*exp(-r*t)*cnd(-d2)-s*cnd(-d1)
    | Call -> s*cnd(d1)-x*exp(-r*t)*cnd(d2)

```

Let's use the `black_scholes` function using some numbers for the `call` and `put` options. Suppose we want to know the price of an option where the underlying asset is a stock traded at \$58.60 with an annual volatility of 30 percent. The risk-free interest rate is, let's say, one percent. We can use our formula that we defined previously to get the theoretical price according to the Black-Scholes formula of a `call` option with six months to maturity (0.5 years):

```
> black_scholes Call 58.60 60.0 0.5 0.01 0.3;;  
val it : float = 4.465202269
```

We will get the value for the `put` option just by changing the flag of the function:

```
> black_scholes Put 58.60 60.0 0.5 0.01 0.3;;  
val it : float = 5.565951021
```

Sometimes, it's more convenient to express the time to maturity in number of days instead of years. Let's introduce a helper function for that purpose:

```
/// Convert the nr of days to years  
let days_to_years d = (float d) / 365.25
```

Note the number `365.25`, which includes the factor for leap years. This is not necessary in our examples, but is used for correctness. We can now use this function instead when we know the time in days:

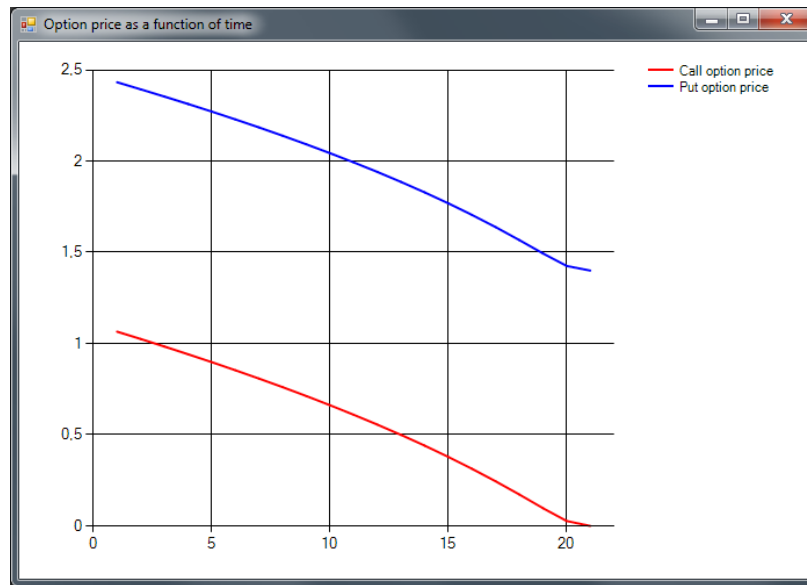
```
> days_to_years 30;;  
val it : float = 0.08213552361
```

Let's use the same preceding example, but now with 20 days to maturity:

```
> black_scholes Call 58.60 60.0 (days_to_years 20) 0.01 0.3;;  
val it : float = 1.065115482  
  
> black_scholes Put 58.60 60.0 (days_to_years 20) 0.01 0.3;;  
val it : float = 2.432270266
```

Using Black-Scholes together with charts

Sometimes, it's useful to be able to plot the price of an option until expiration. We can use our previously defined functions and vary the time left and plot the values coming out. In this example, we'll make a program that outputs the graph shown in the following screenshot:



A chart showing prices for call and put options as a function of time

The following code is used to generate the graph in the preceding screenshot:

```
/// Plot price of option as function of time left to maturity
#r "System.Windows.Forms.DataVisualization.dll"

open System
open System.Net
open System.Windows.Forms
open System.Windows.Forms.DataVisualization.Charting
open Microsoft.FSharp.Control.WebExtensions

/// Create chart and form
let chart = new Chart(Dock = DockStyle.Fill)
let area = new ChartArea("Main")
chart.ChartAreas.Add(area)
chart.Legends.Add(new Legend())

let mainForm = new Form(Visible = true, TopMost = true,
                        Width = 700, Height = 500)
```

```
do mainForm.Text <- "Option price as a function of time"
mainForm.Controls.Add(chart)

/// Create series for call option price
let optionPriceCall = new Series("Call option price")
do optionPriceCall.ChartType <- SeriesChartType.Line
do optionPriceCall.BorderWidth <- 2
do optionPriceCall.Color <- Drawing.Color.Red
chart.Series.Add(optionPriceCall)

/// Create series for put option price
let optionPricePut = new Series("Put option price")
do optionPricePut.ChartType <- SeriesChartType.Line
do optionPricePut.BorderWidth <- 2
do optionPricePut.Color <- Drawing.Color.Blue
chart.Series.Add(optionPricePut)

/// Calculate and plot call option prices
let opc = [for x in [(days_to_years 20)..(-(days_to_years
  1))..0.0] do yield black_scholes Call 58.60 60.0 x 0.01 0.3]
do opc |> Seq.iter (optionPriceCall.Points.Add >> ignore)

/// Calculate and plot put option prices
let opp = [for x in [(days_to_years 20)..(-(days_to_years
  1))..0.0] do yield black_scholes Put 58.60 60.0 x 0.01 0.3]
do opp |> Seq.iter (optionPricePut.Points.Add >> ignore)
```

The preceding code is just a modified version of the code shown in the previous chapter with the options parts added. We have two series in this chart, one for call options and one for put options. We also add a legend for each of the series. The last part is the calculation of the prices and the actual plotting. List comprehensions are used for compact code, and the Black-Scholes formula is called everyday until expiration, where the days are counted down by one day at each step.

It's up to you as the reader to modify the code to plot various aspects of the option, such as the option price as a function of an increase in the underlying stock price, and so on.

Introducing the greeks

The **greeks** are partial derivatives of the Black-Scholes formula with respect to a particular parameter, such as time, rate, volatility, or stock price. The greeks can be divided into two or more categories with respect to the order of the derivatives. In this section, we'll look at the first-order and second-order greeks.

First-order greeks

In this section, we'll present the first-order greeks using the following table:

Name	Symbol	Description
Delta	Δ	Rate of change of option value with respect to the change in the price of the underlying asset.
Vega	v	Rate of change of option value with respect to the change in the volatility of the underlying asset. It is referred to as the volatility sensitivity.
Theta	Θ	Rate of change of option value with respect to time. The sensitivity with respect to time will decay as time elapses, and this phenomenon is referred to as the time decay.
Rho	ρ	Rate of change of option value with respect to the interest rate.

Second-order greeks

In this section, we'll present the second-order greeks using the following table:

Name	Symbol	Description
Gamma	Γ	Rate of change of delta with respect to a change in the price of the underlying asset.
Veta	-	Rate of change in Vega with respect to time.
Vera	-	Rate of change in Rho with respect to volatility.



Some of the second-order greeks are omitted for clarity; we'll not cover these in this book.

Implementing the greeks in F#

Let's implement the greeks: Delta, Gamma, Vega, Theta, and Rho. First, we look at the formulas for each greek. In some of the cases, they vary for calls and puts respectively as shown in the following table:

Name	Symbol	Formula for Calls	Formula for Puts
Delta	Δ	$N(d_1)$	$N(d_1) - 1$
Gamma	Γ	$\frac{N'(d_1)}{S\sigma\sqrt{T-t}}$	
Vega	ν	$SN'(d_1)\sqrt{T-t}$	
Theta	Θ	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}N(d_2)$	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} + rKe^{-r(T-t)}N(-d_2)$
Rho	ρ	$K(T-t)e^{-r(T-t)}N(d_2)$	$-K(T-t)e^{-r(T-t)}N(-d_2)$

We need the derivative of the cumulative distribution function, which in fact is the normal distribution with a mean of 0 and standard deviation of 1:

```

/// Normal distribution
open MathNet.Numerics.Distributions;

let normd = new Normal(0.0, 1.0)

```

Delta

Delta is the rate of change of the option price with respect to the change in the price of the underlying asset:

```

/// Black-Scholes Delta
// call_put_flag: Put | Call

```

```

// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes_delta call_put_flag s x t r v =
  let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
  match call_put_flag with
  | Put -> cnd(d1) - 1.0
  | Call -> cnd(d1)

```

Gamma

Gamma is the rate of change of delta with respect to the change in the price of the underlying asset. This is the second derivative with respect to the price of the underlying asset. It measures the acceleration of the price of the option with respect to the underlying price:

```

/// Black-Scholes Gamma
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes_gamma s x t r v =
  let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
  normd.Density(d1) / (s*v*sqrt(t))

```

Vega

Vega is the rate of change of the option value with respect to the change in the volatility of the underlying asset. It is referred to as the sensitivity of the volatility:

```

/// Black-Scholes Vega
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes_vega s x t r v =
  let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
  s*normd.Density(d1)*sqrt(t)

```

Theta

Theta is the rate of change of the option value with respect to time. The sensitivity with respect to time will decay as time elapses, and this phenomenon is referred to as the time decay:

```
/// Black-Scholes Theta
// call_put_flag: Put | Call
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes_theta call_put_flag s x t r v =
  let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
  let d2=d1-v*sqrt(t)
  let res = ref 0.0
  match call_put_flag with
  | Put -> -(s*normd.Density(d1)*v)/(2.0*sqrt(t))+r*x*exp(-
    r*t)*cnd(-d2)
  | Call -> -(s*normd.Density(d1)*v)/(2.0*sqrt(t))-r*x*exp(-
    r*t)*cnd(d2)
```

Rho

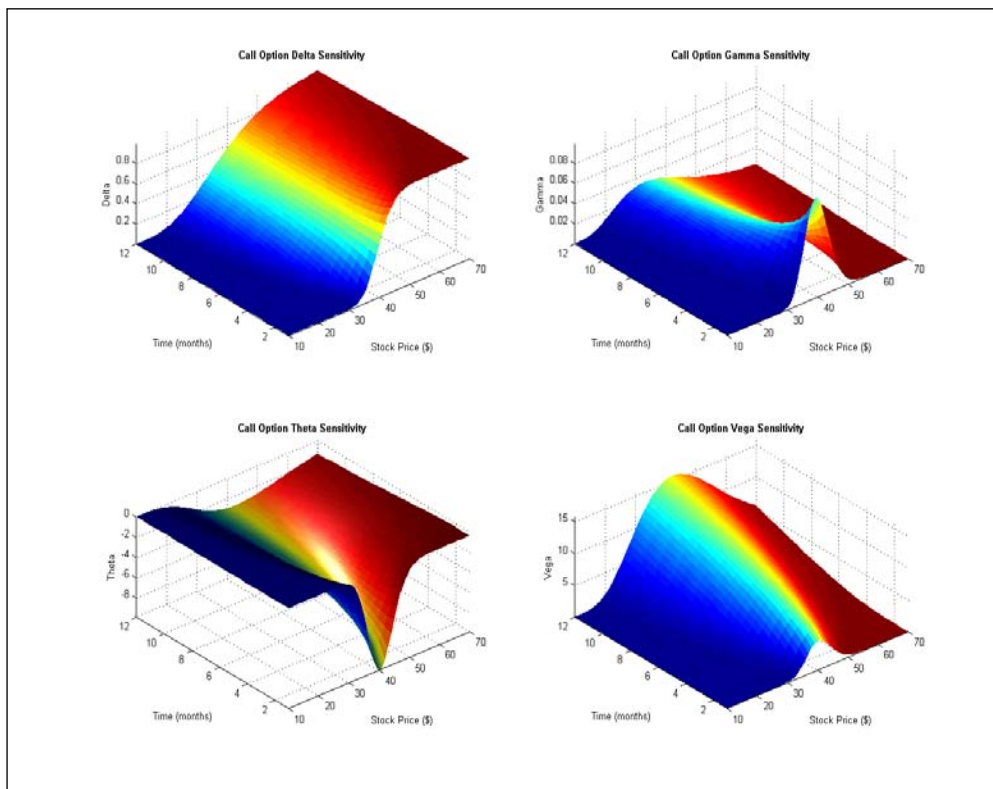
Rho is the rate of change of the option value with respect to the interest rate:

```
/// Black-Scholes Rho
// call_put_flag: Put | Call
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes_rho call_put_flag s x t r v =
  let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
  let d2=d1-v*sqrt(t)
  let res = ref 0.0
  match call_put_flag with
  | Put -> -x*t*exp(-r*t)*cnd(-d2)
  | Call -> x*t*exp(-r*t)*cnd(d2)
```

Investigating the sensitivity of the greeks

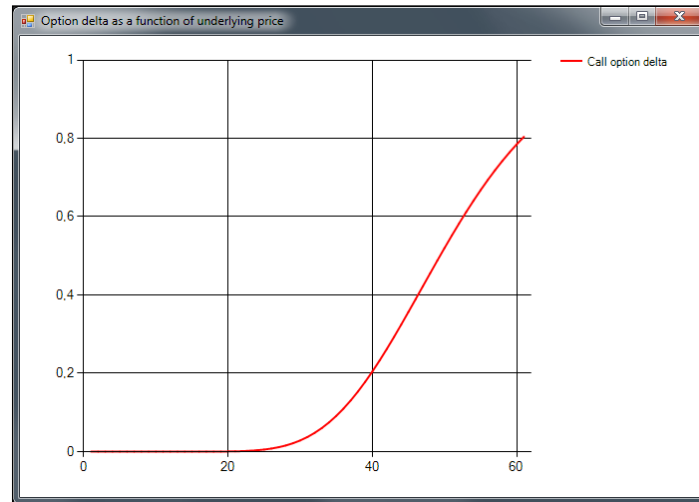
Now that we have all the greeks implemented, we'll investigate the sensitivity of some of them and see how they vary when the underlying stock price changes.

The following screenshot is a surface plot with four of the greeks where the time and the underlying price is changing. This figure is generated in MATLAB, and will not be generated in F#. We'll use a 2D version of the graph to study the greeks as shown in the following screenshot:



The surface plot of Delta, Gamma, Theta, and Rho of a call option

In this section, we'll start by plotting the value of Delta for a call option where we vary the price of the underlying asset. This will result in the following 2D plot as shown in the following screenshot:



A plot of call option delta versus the price of the underlying asset

The result in the plot shown in the preceding screenshot will be generated by the code presented next. We'll reuse most of the code from the example where we looked at the option prices for calls and puts. A slightly modified version is presented in the following code, where the price of the underlying asset varies from \$10.0 to \$70.0:

```
/// Plot delta of call option as function of underlying price
#r "System.Windows.Forms.DataVisualization.dll"

open System
open System.Net
open System.Windows.Forms
open System.Windows.Forms.DataVisualization.Charting
open Microsoft.FSharp.Control.WebExtensions

/// Create chart and form
let chart = new Chart(Dock = DockStyle.Fill)
let area = new ChartArea("Main")
chart.ChartAreas.Add(area)
chart.Legends.Add(new Legend())
```

```

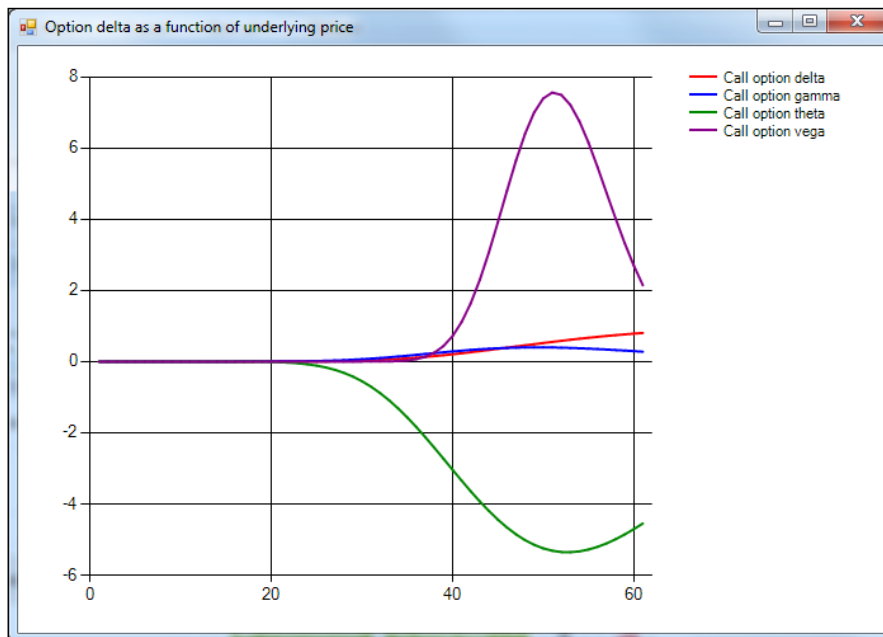
let mainForm = new Form(Visible = true, TopMost = true,
                        Width = 700, Height = 500)
do mainForm.Text <- "Option delta as a function of underlying
  price"
mainForm.Controls.Add(chart)

/// Create series for call option delta
let optionDeltaCall = new Series("Call option delta")
do optionDeltaCall.ChartType <- SeriesChartType.Line
do optionDeltaCall.BorderWidth <- 2
do optionDeltaCall.Color <- Drawing.Color.Red
chart.Series.Add(optionDeltaCall)

/// Calculate and plot call delta
let opc = [for x in [10.0..1.0..70.0] do yield black_scholes_delta
  Call x 60.0 0.5 0.01 0.3]
do opc |> Seq.iter (optionDeltaCall.Points.Add >> ignore)

```

We can extend the code to plot all four greeks, as shown in the screenshot with the 2D surface plots. The result will be a graph as shown in the following screenshot:



A graph showing greeks for a call option with respect to a price change (x axis)

Code listing for visualizing the four greeks

The following is the code listing for the entire program used to create the preceding graph:

```
#r "System.Windows.Forms.DataVisualization.dll"

open System
open System.Net
open System.Windows.Forms
open System.Windows.Forms.DataVisualization.Charting
open Microsoft.FSharp.Control.WebExtensions

/// Create chart and form
let chart = new Chart(Dock = DockStyle.Fill)
let area = new ChartArea("Main")
chart.ChartAreas.Add(area)
chart.Legends.Add(new Legend())

let mainForm = new Form(Visible = true, TopMost = true,
                        Width = 700, Height = 500)

do mainForm.Text <- "Option delta as a function of underlying
  price"
mainForm.Controls.Add(chart)
```

We'll create a series for each greek as follows:

```
/// Create series for call option delta
let optionDeltaCall = new Series("Call option delta")
do optionDeltaCall.ChartType <- SeriesChartType.Line
do optionDeltaCall.BorderWidth <- 2
do optionDeltaCall.Color <- Drawing.Color.Red
chart.Series.Add(optionDeltaCall)

/// Create series for call option gamma
let optionGammaCall = new Series("Call option gamma")
do optionGammaCall.ChartType <- SeriesChartType.Line
do optionGammaCall.BorderWidth <- 2
do optionGammaCall.Color <- Drawing.Color.Blue
chart.Series.Add(optionGammaCall)

/// Create series for call option theta
let optionThetaCall = new Series("Call option theta")
do optionThetaCall.ChartType <- SeriesChartType.Line
do optionThetaCall.BorderWidth <- 2
```

```

do optionThetaCall.Color <- Drawing.Color.Green
chart.Series.Add(optionThetaCall)

/// Create series for call option vega
let optionVegaCall = new Series("Call option vega")
do optionVegaCall.ChartType <- SeriesChartType.Line
do optionVegaCall.BorderWidth <- 2
do optionVegaCall.Color <- Drawing.Color.Purple
chart.Series.Add(optionVegaCall)

```

Next, we'll calculate the values to plot for each greek:

```

/// Calculate and plot call delta
let opd = [for x in [10.0..1.0..70.0] do yield black_scholes_delta
  Call x 60.0 0.5 0.01 0.3]
do opd |> Seq.iter (optionDeltaCall.Points.Add >> ignore)

/// Calculate and plot call gamma
let opg = [for x in [10.0..1.0..70.0] do yield black_scholes_gamma
  x 60.0 0.5 0.01 0.3]
do opg |> Seq.iter (optionGammaCall.Points.Add >> ignore)

/// Calculate and plot call theta
let opt = [for x in [10.0..1.0..70.0] do yield black_scholes_theta
  Call x 60.0 0.5 0.01 0.3]
do opt |> Seq.iter (optionThetaCall.Points.Add >> ignore)

/// Calculate and plot call vega
let opv = [for x in [10.0..1.0..70.0] do yield black_scholes_vega
  x 60.0 0.1 0.01 0.3]
do opv |> Seq.iter (optionVegaCall.Points.Add >> ignore)

```

The Monte Carlo method

The Monte Carlo method is used to sample numerical integration using random numbers and to study the mean value of a large number of samples. The Monte Carlo method is especially useful when there is no closed form solution available.

In this section, we'll look at the simplest case, where we have path-dependent European options. We are going to sample numerical integration using a random drifting parameter. This will lead to various average values for the stochastic process, which makes up the movement of the underlying asset. We'll do this using 1,000 and 1,000,000 samples respectively and compare the results. Let's dig into the following code:


```
/// Monte Carlo implementation

/// Convert the nr of days to years
let days_to_years d =
  (float d) / 365.25

/// Asset price at maturity for sample rnd
// s: stock price
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
// rnd: sample
let price_for_sample s t r v rnd =
  s*exp((r-v*v/2.0)*t+v*rnd*sqrt(t))

/// For each sample we run the monte carlo simulation
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
// samples: random samples as input to simulation
let monte_carlo s x t r v (samples:seq<float>) = samples
  |> Seq.map (fun rnd -> (price_for_sample s t r v rnd) - x)
  |> Seq.average

/// Generate sample sequence
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 1000 -> rnd() * 1.0]
```

Let's test it with the same values used for the Black-Scholes formula for a put option:

```
> black_scholes 'c' 58.60 60.0 0.5 0.01 0.3;;
val it : float = 4.465202269

/// Monte carlo for call option
> monte_carlo 58.60 60.0 0.5 0.01 0.3 data
val it : float = 4.243545757
```

This is close to being spot on; we can increase the number of samples and see if we get another value:

```
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 1000000 -> rnd() * 1.0]

// Monte carlo for call option
> monte_carlo 58.60 60.0 0.5 0.01 0.3 data;;
val it : float = 4.146170039
```

The preceding code uses the following formula to accomplish the task of estimating the price. In a short summary, the Monte Carlo method here can be thought of as a random pick of the drifting parameter $\sigma \cdot \text{rnd} \cdot \text{sqrt}(t)$. The mean of all these samples generated will then represent the estimated value of the option at maturity. In reality, the Monte Carlo method is not used for European options, in contrast to what's illustrated in this section. The choice of European options was mainly due to simplicity, to illustrate the concepts:

$$S(T) = S(0) \exp \left[\left(r - \frac{\sigma^2}{2} \right) T + \sigma \epsilon \sqrt{T} \right]$$

The formula to estimate the sample value of a price at maturity for an asset

Summary

In this chapter, we looked at option pricing in F# using the famous Black-Scholes formula together with the Monte Carlo method for European options. Once again, F# has proven itself powerful, and in numerical implementations, this is especially true. The code is almost identical to the mathematical functions, which makes it easy to implement without any extra ceremony needed. The lessons learned in this chapter will be used in the next chapter to dig deeper into options and volatility.

6

Exploring Volatility

In this chapter, you will learn about volatility and using numerical methods in F# to explore the properties of options. We'll solve for the intrinsic volatility, called implied volatility, in the Black-Scholes model using the code from the previous chapter and extending it with numerical methods covered in *Chapter 3, Financial Mathematics and Numerical Analysis*.

In this chapter you will learn:

- Actual volatility and implied volatility
- Using F# to calculate actual volatility
- Solving for implied volatility in Black-Scholes
- Using numerical methods for options
- Delta hedging
- Briefly about volatility arbitrage

Introduction to volatility

In the previous chapter we looked at the basics behind Black-Scholes for European options. We'll continue to explore options in this chapter and look at volatility and how to use F# to help us out. Volatility measures changes in price as annualized standard deviation, which is the rate at which the price of a financial instrument fluctuates up or down. Higher volatility means larger dispersion and lower volatility means, of course, smaller dispersion. Volatility relates to variance and variance equals the square of the standard deviation, as covered previously.

Black-Scholes assumes normal distributed movements in stock prices, which is not really the case in reality according to observations. In real life, the distribution is more fat-tailed, which means that negative price movements tend to be larger when they occur, but positive movements are more common, and smaller on average.



Figure 1: Courtesy of Yahoo! Finance. Shows S&P 500 Index with low volatility (9.5 % annualized) and Apple with high volatility (31 % annualized)

Actual volatility

Actual volatility is the current observed volatility over a specific period of time, typically the last month or year. Actual volatility uses the current market price and several previous observations. In simple terms, it is the standard deviation of the logarithmic returns of a series of today's and existing price data.

Implied volatility

Implied volatility is the volatility encapsulated in option prices. If we use Black-Scholes, we need to provide several inputs: stock price, strike price, risk free interest rate, volatility, and time to expiration. This will output a theoretical price for that option in terms of the assumptions made. We can use Black-Scholes backwards to get the implied volatility. That means we can extract the volatility from the market price of that option if it's traded on an exchange as a fair market price. Doing so requires us to use a numerical method for root solving, which has been covered in the chapter about numerical analysis.

Implied volatility using current prices will solve for implied volatility in the Black-Scholes model using a bisection solver, which we will study in a later section in this chapter.

Exploring volatility in F#

Let's look at a sample program in F# that will cover some aspects of volatility and how to calculate the volatility from real market data. We'll look at some tech stocks from NASDAQ and calculate the annualized volatility for each of them. First, we need to define a function to do the calculations. Annualized volatility is defined as follows:

$$\sigma = \frac{\sigma_{SD}}{\sqrt{P}}$$

Where P is the time period in years, σ is the annualized volatility, and σ_{SD} is the standard deviation during the time period P .

Here we use P as $\sigma_{SD} \frac{1}{\#days}$, this means we can rewrite the formula as:

$$\sigma = \sigma_{SD} \sqrt{\#days}$$

We start by using the function to calculate the standard deviation as mentioned in *Chapter 3, Financial Mathematics and Numerical Analysis*:

```
/// Calculate the standard deviation
let stddev(values:seq<float>) =
    values
    |> Seq.fold (fun acc x -> acc + (1.0 / float (Seq.length
    values)) * (x - (Seq.average values)) ** 2.0) 0.0
    |> sqrt
```

Then we need a function to calculate the logarithmic returns. This is done using the `Seq.pairwise` function, since we need a window of size 2. This is the same as using the `Seq.windowed` function with a size of 2.

```
/// Calculate logarithmic returns
let calcDailyReturns(prices:seq<float>) =
    prices
    |> Seq.pairwise
    |> Seq.map (fun (x, y) -> log (x / y))
```

Last but not least, we have our function to calculate annualized volatility from the return series:

```
/// Annualized volatility
let annualVolatility(returns:seq<float>) =
    let sd = stddev(calcDailyReturns(returns))
    let days = Seq.length(returns)
    sd * sqrt(float days)
```

This function uses the mathematical equation described previously, with the number of days squared and multiplied by the standard deviation for the return series. This can be interpreted as a scaling factor.

These functions are the main building blocks we need to proceed with. The next step is to reuse the functionality to obtain prices from Yahoo! Finance, slightly modified to use the preceding functions. Next, we introduce two helper functions. The first is to format a number as a string with a leading zero if the number is smaller than ten. The second function is to help us construct the URL needed to request data from Yahoo! Finance:

```
let formatLeadingZero(number:int):String =
    String.Format("{0:00}", number)

/// Helper function to create the Yahoo-finance URL
let constructURL(symbol, fromDate:DateTime, toDate:DateTime) =
    let fm = formatLeadingZero(fromDate.Month-1)
    let fd = formatLeadingZero(fromDate.Day)
    let fy = formatLeadingZero(fromDate.Year)
    let tm = formatLeadingZero(toDate.Month-1)
    let td = formatLeadingZero(toDate.Day)
    let ty = formatLeadingZero(toDate.Year)
    "http://ichart.finance.yahoo.com/table.csv?s=" + symbol +
    "&d=" + tm + "&e=" + td + "&f=" + ty + "&g=d&a=" + fm + "&b="
    + fd + "&c=" + fy + "&ignore=.csv"

/// Synchronous fetching (just one request)
let fetchOne symbol fromDate toDate =
    let url = constructURL(symbol, fromDate, toDate)
    let uri = new System.Uri(url)
    let client = new WebClient()
    let html = client.DownloadString(uri)
    html

/// Parse CSV
let getPrices stock fromDate toDate =
    let data = fetchOne stock fromDate toDate
    data.Trim().Split('\n')
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> float s.[4])
    |> Seq.takeWhile (fun s -> s >= 0.0)

/// Returns a formatted string with volatility for a stock
let getAnnualizedVol stock fromStr toStr =
```

```

let prices = getPrices stock (System.DateTime.Parse fromStr)
(System.DateTime.Parse toStr)
let vol = Math.Round(annualVolatility(prices) * 100.0, 2)
sprintf "Volatility for %s is %.2f %" stock vol

```

Let's try it out with a few stocks from NASDAQ using F# Interactive:

```

> getAnnualizedVol "MSFT" "2013-01-01" "2013-08-29";;
val it : string = "Volatility for MSFT is 21.30 %"

> getAnnualizedVol "ORCL" "2013-01-01" "2013-08-29";;
val it : string = "Volatility for ORCL is 20.44 %"

> getAnnualizedVol "GOOG" "2013-01-01" "2013-08-29";;
val it : string = "Volatility for GOOG is 14.80 %"

> getAnnualizedVol "EBAY" "2013-01-01" "2013-08-29";;
val it : string = "Volatility for EBAY is 20.82 %"

> getAnnualizedVol "AAPL" "2013-01-01" "2013-08-29";;
val it : string = "Volatility for AAPL is 25.16 %"

> getAnnualizedVol "AMZN" "2013-01-01" "2013-08-29";;
val it : string = "Volatility for AMZN is 21.10 %"

> getAnnualizedVol "^GSPC" "2013-01-01" "2013-08-29";;
val it : string = "Volatility for ^GSPC is 9.15 %"

```

The result of the preceding code can be summarized in the following table:

Symbol	Name	Annualized volatility
MSFT	Microsoft Corp.	21.30 percent
ORCL	Oracle	20.44 percent
GOOG	Google Inc.	14.80 percent
EBAY	eBay	20.82 percent
AAPL	Apple Computer	25.16 percent
AMZN	Amazon	21.10 percent
^GSPC	S&P 500	9.15 percent

From the preceding table, we can see and compare annualized volatility for the selected stocks and the S&P 500 index. It's clear from the data which one has the highest respectability and the lowest volatility. AAPL and ^GSPC are compared in *Figure 1* in the introduction of this chapter. Sometimes, volatility can tell you something about the risk involved in investing in that particular instrument. But keep in mind that this data is historical and will not interpret the future price movements of an instrument.

The complete application

Following is the code listing for the entire program used earlier. You can modify the parameters to the Yahoo! Finance web service that returns CSV data. The parameters are a, b, c for the `from-date` parameter and d, e, f for the `to-date` parameter together with the symbol of the stock, see the following table:

Parameter	Description	Example
s	Stock symbol	MSFT
d	To month of year	07
e	To day of month	29
f	To year	2013
a	From month of year	00
b	To day of month	1
c	To year	2013

Let's have a look at an example where we downloaded data from Yahoo! for a couple of stocks listed on NASDAQ as well as the S&P500 index. We'll look at a timespan from January 01, 2013 to August 2, 2013:

```
open System
open System.Net

/// Calculate the standard deviation
let stddev(values:seq<float>) =
    values
    |> Seq.fold (fun acc x -> acc + (1.0 / float (Seq.length
    values)) * (x - (Seq.average values)) ** 2.0) 0.0
    |> sqrt

/// Calculate logarithmic returns
let calcDailyReturns(prices:seq<float>) =
    prices
    |> Seq.pairwise
```

```
|> Seq.map (fun (x, y) -> log (x / y))

/// Annualized volatility
let annualVolatility(returns:seq<float>) =
    let sd = stddev(calcDailyReturns(returns))
    let days = Seq.length(returns)
    sd * sqrt(float days)

let formatLeadingZero(number:int):String =
    String.Format("{0:00}", number)

/// Helper function to create the Yahoo-finance URL
let constructURL(symbol, fromDate:DateTime, toDate:DateTime) =
    let fm = formatLeadingZero(fromDate.Month-1)
    let fd = formatLeadingZero(fromDate.Day)
    let fy = formatLeadingZero(fromDate.Year)
    let tm = formatLeadingZero(toDate.Month-1)
    let td = formatLeadingZero(toDate.Day)
    let ty = formatLeadingZero(toDate.Year)
    "http://ichart.finance.yahoo.com/table.csv?s=" + symbol +
    "&d=" + tm + "&e=" + td + "&f=" + ty + "&g=d&a=" + fm + "&b="
    + fd + "&c=" + fy + "&ignore=.csv"

/// Synchronous fetching (just one request)
let fetchOne symbol fromDate toDate =
    let url = constructURL(symbol, fromDate, toDate)
    let uri = new System.Uri(url)
    let client = new WebClient()
    let html = client.DownloadString(uri)
    html

/// Parse CSV
let getPrices stock fromDate toDate =
    let data = fetchOne stock fromDate toDate
    data.Trim().Split('\n')
    |> Seq.skip 1
    |> Seq.map (fun s -> s.Split(','))
    |> Seq.map (fun s -> float s.[4])
    |> Seq.takeWhile (fun s -> s >= 0.0)

/// Returns a formatted string with volatility for a stock
let getAnnualizedVol stock fromStr toStr =
    let prices = getPrices stock (System.DateTime.Parse fromStr)
    (System.DateTime.Parse toStr)
```

```
let vol = Math.Round(annualVolatility(prices) * 100.0, 2)
sprintf "Volatility for %s is %.2f %%" stock vol

getAnnualizedVol "MSFT" "2013-01-01" "2013-08-29"
// val it : string = "Volatility for MSFT is 21.30 %"

getAnnualizedVol "ORCL" "2013-01-01" "2013-08-29"
// val it : string = "Volatility for ORCL is 20.44 %"

getAnnualizedVol "GOOG" "2013-01-01" "2013-08-29"
// val it : string = "Volatility for GOOG is 14.80 %"

getAnnualizedVol "EBAY" "2013-01-01" "2013-08-29"
// val it : string = "Volatility for EBAY is 20.82 %"

getAnnualizedVol "AAPL" "2013-01-01" "2013-08-29"
// val it : string = "Volatility for AAPL is 25.16 %"

getAnnualizedVol "AMZN" "2013-01-01" "2013-08-29"
// val it : string = "Volatility for AMZN is 21.10 %"

getAnnualizedVol "^GSPC" "2013-01-01" "2013-08-29"
// val it : string = "Volatility for ^GSPC is 9.15 %"
```

In this section we looked at actual volatility for some instruments using data from Yahoo! Finance. In the next section we'll look into implied volatility and how to extract that information from the Black-Scholes formula.

Learning about implied volatility

Here we'll use the bisection method introduced in *Chapter 3, Financial Mathematics and Numerical Analysis*. This is a numerical method for finding roots. The implied volatility is the root where the function value is zero for the Black-Scholes function for different input parameters. The volatility of an underlying instrument is the input to Black-Scholes which gives the same price as the current price of the option.

Vega tells us about the sensitivity in the option price for the changes in the volatility of the underlying asset. Look at Yahoo! Finance and find the option data. Take that option data into the following solve function:

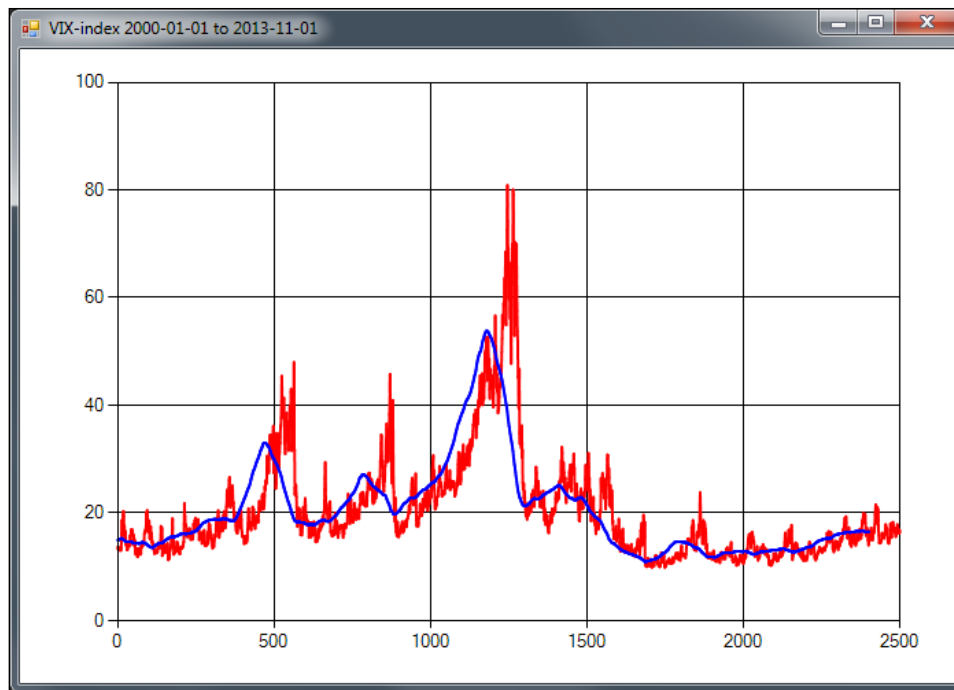


Figure 2: The VIX-index for the S&P500 index options from 2000-01-01 to 2013-11-01

The VIX-index, as seen in the preceding screenshot, is an index which combines the implied volatility of S&P 500 index options. This can be interpreted as an indication of future volatility.

Solving for implied volatility

Next we'll use a method for solving for implied volatility for European options. This can be done by numerically solving for the root using the bisection method.

To be able to understand why we use the bisection solver to find the root of the Black-Scholes equation, we need some tools. First we recapture the definition of the call and put price as a function of the estimated volatility and a set of parameters (denoted):

$$\begin{cases} C = f(\sigma, \cdot) \\ P = f(\sigma, \cdot) \end{cases}$$

To extract the implied volatility, we need an inverse function of the Black-Scholes formula. Unfortunately, there is no analytical inverse of that function. Instead, we can say that the Black-Scholes formula, with the implied volatility minus the current market price of that option, has a call option in this case of zero. Following \bar{C} is the current market price for the call option studied in this section:

$$g = f^{-1} \Rightarrow \sigma_{\bar{C}} = g(\bar{C}, .)$$
$$f(\sigma_{\bar{C}}, .) - \bar{C} = 0$$

This enables us to use a numerical root solver to find the implied volatility. Following is an implementation of the bisection solver in F#. We'll also use the earlier Black-Scholes implementation

```
/// Solve for implied volatility

let pow x n = exp (n * log(x) )

type PutCallFlag = Put | Call

/// Cumulative distribution function
let cnd x =
    let a1 = 0.31938153
    let a2 = -0.356563782
    let a3 = 1.781477937
    let a4 = -1.821255978
    let a5 = 1.330274429
    let pi = 3.141592654
    let l = abs(x)
    let k = 1.0 / (1.0 + 0.2316419 * l)
    let w = (1.0-1.0/sqrt(2.0*pi)*exp(-
1*1/2.0)*(a1*k+a2*k*k+a3*(pow k 3.0)+a4*(pow k 4.0)+a5*(pow k
5.0)))
    if x < 0.0 then 1.0 - w else w

/// Black-Scholes
// call_put_flag: Call | Put
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes call_put_flag s x t r v =
    let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
```

```

let d2=d1-v*sqrt(t)
//let res = ref 0.0

match call_put_flag with
| Put -> x*exp(-r*t)*cnd(-d2)-s*cnd(-d1)
| Call -> s*cnd(d1)-x*exp(-r*t)*cnd(d2)
/// Bisection solver
let rec bisect n N (f:float -> float) (a:float) (b:float) (t:float) :
float =
  if n >= N then -1.0
  else
    let c = (a + b) / 2.0
    if f(c) = 0.0 || (b - a) / 2.0 < t then
      // Solution found
      c
    else
      if sign(f(c)) = sign(f(a)) then
        bisect (n + 1) N f c b t
      else
        bisect (n + 1) N f a c t

Let's use it!

/// Calculate implied volatility for an option
bisect 0 25 (fun x -> (black_scholes Call 58.60 60.0 0.05475 0.0095 x)
- 1.2753) 0.0 1.0 0.001

```

Running the preceding code will result in an implied volatility of 0.3408203125, approximately 34.1 percent volatility. Note that we have to subtract for the current market price of the option (1.2753) as well, because we are solving for roots. The last three inputs are the start and stop values, 0.0 and 1.0 means 0 percent and 100 percent respectively in terms of volatility. The stepping size is set to be 0.001, which is 0.1 percent. A simple way to test whether the values are correct is to first use the Black-Scholes formula using an actual volatility to get a theoretical price for that option.

Let's assume we have a call option with a strike of \$75.00, stock price of \$73.00, 20 days to maturity (approximately 0.05475 years), a volatility of 15 percent, and a fixed rate of 0.01 percent; this will result in an option price of:

```

let option_price = black_scholes Call 73.00 75.0 (days_to_years 20)
0.01 0.15

```

We can now use this price to see if the bisection method works and solve for the implied volatility. In this case we can expect an implied volatility to be exactly the same as the volatility we put into the Black-Scholes formula; that is 15 percent:

```
> bisect 0 25 (fun x -> (black_scholes Call 73.00 75.0 (days_to_years
20) 0.01 x) - option_price) 0.0 1.0 0.001;;
val it : float = 0.1494140625
```

Delta hedging using Black-Scholes

A delta neutral portfolio is constructed by an option and the underlying instrument. The portfolio will, in theory, be immune against small changes in the underlying price. When talking about delta hedging, the hedge ratio of a derivative is used to define the amount of underlying price needed for each option. Delta hedging is the trading strategy that maintains a delta neutral portfolio for small changes in the underlying price.

Briefly, let's look at how to do this in practice. Suppose we have N derivatives. This needs to be hedged to protect against price movements. We then need to buy the underlying stock to create the hedge. The whole procedure can be described in three steps:

1. N derivatives need to be delta hedged
2. Buy underlying stock to protect derivatives
3. Rebalance hedge position on a regular basis

To determine how many stocks we need, we use the delta of the option, Δ . This tells us how much the option price changes for a change in price of the underlying price. The portfolio uses ΔN shares to be delta neutral; often there are 100 shares for each options contract.

The price of the underlying stock is constantly fluctuating, which leads to changes in the option prices. This means we also have to rebalance our portfolio. This is due to the time value of options and the change in the underlying price.

Let's use F# to calculate the number of stocks we need for a particular option to delta hedge:

```
/// Assumes 100 shares of stock per option
let nr_of_stocks_delta_hedge N =
    (black_scholes_delta 'c' 58.0 60.0 0.5 0.01 0.3) * 100.0 *
    (float N)

/// Calculate nr of shares needed to cover 100 call options
nr_of_stocks_delta_hedge 100
```

If we evaluate the last row, we'll obtain the number of shares needed to create a delta neutral hedge with 100 calls.

```
> nr_of_stocks_delta_hedge 100;;
val it : float = 4879.628104
```

The answer is approximately 4880 shares needed to hedge the call options.

Exploring the volatility smile

The volatility smile is a phenomenon frequently observed in the markets.

This phenomenon is mostly explained by the assumptions made in the Black-Scholes formula. Black-Scholes assumes constant volatility throughout the life of an option.

If the Black-Scholes formula was corrected for this behavior, by taking into account the nature of volatility being non-constant, we would end up with a flat volatility surface.

Further, the volatility smile describes the volatility for a certain price of the option relative to the strike price of the same. The volatility surface is often referring to a three-dimensional graph of the volatility smile, where time to maturity and moneyness.

Moneyness is the ratio between the spot price of the underlying asset, S , and the strike price of the option, K :

$$M = \frac{S}{K}$$

Next, we'll look at how to use F# to provide a graph for us where the volatility smile is computed from parametric regression from real market data.

The following data is from Ericsson B options, from the OMX exchange in Sweden:

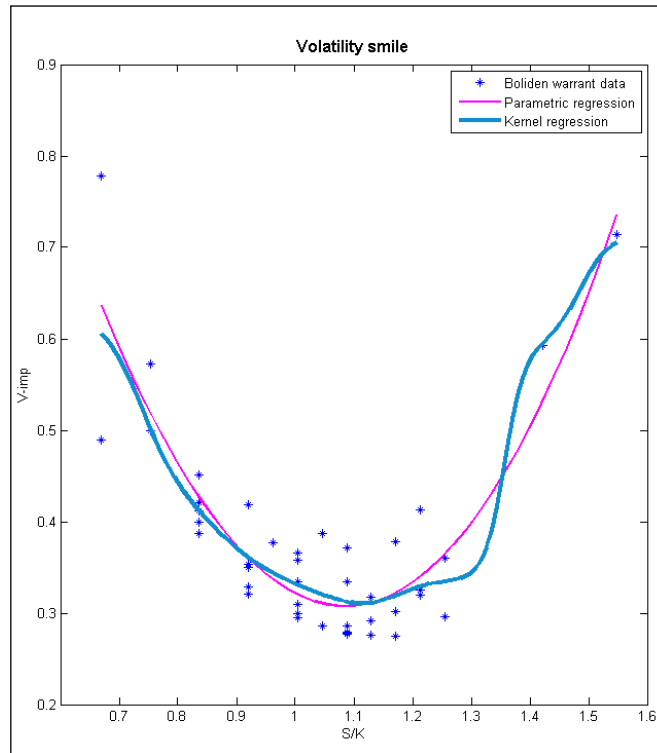


Figure 3: Volatility smile for a warrant

As you can see in the following screenshot, the smile comes from the different implied volatility for the strike prices:

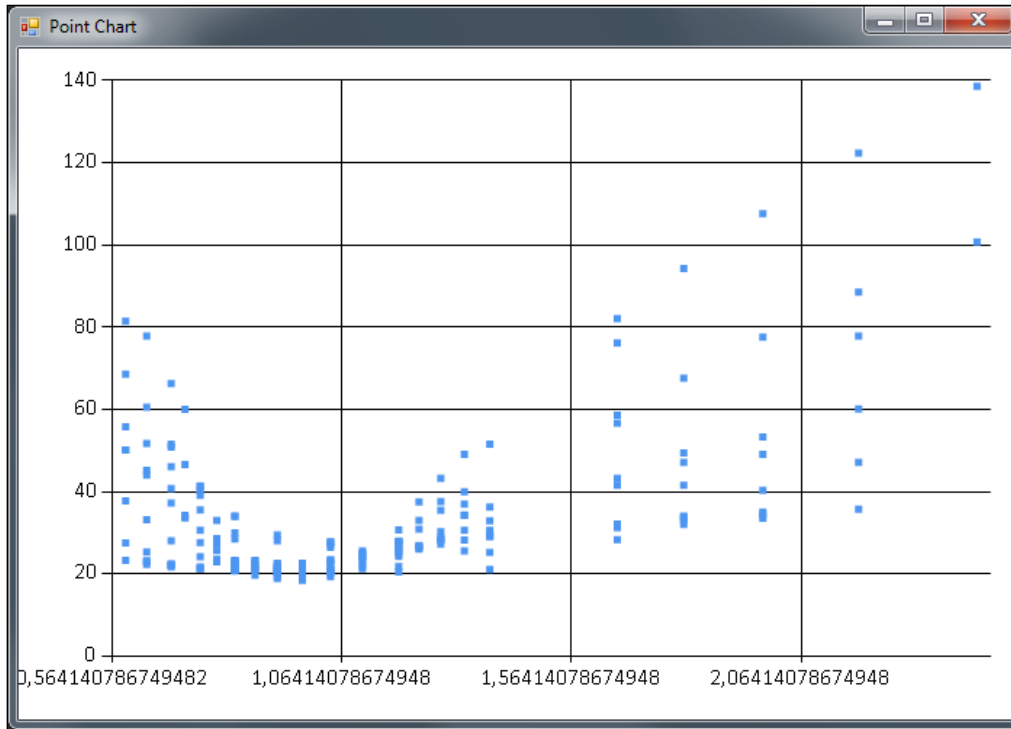


Figure 4: Volatility smile as points from market data

We can use polynomial regression to estimate the volatility smile from the points in the graph. This method was introduced in *Chapter 3, Financial Mathematics and Numerical Analysis*; we'll build on the code from there. The polynomial will be of order two, which means a second degree polynomial, that will describe the expected volatility smile well.

Let's look at an application that will produce the plot in *Figure 5* using Math.NET and FSharpChart:

```
open System.IO
open FSharp.Charting
open System.Windows.Forms.DataVisualization.Charting

open MathNet.Numerics
open MathNet.Numerics.LinearAlgebra
open MathNet.Numerics.LinearAlgebra.Double
open MathNet.Numerics.Distributions

let filePath = @"smile_data.csv"

/// Split row on commas
let splitCommas (l:string) =
    l.Split(',')

/// Read a file into a string array
let openFile (name : string) =
    try
        let content = File.ReadAllLines(name)
        content |> Array.toList
    with
        | :? System.IO.FileNotFoundException as e -> printfn
            "Exception! %s " e.Message; ["empty"]

/// Read the data from a CSV file and returns
/// a tuple of strike price and implied volatility%
let readVolatilityData =
    openFile filePath
    |> List.map splitCommas
    |> List.map (fun cols -> (cols.[2], cols.[3]))

/// Calculates moneyness and parses strings into numbers
let calcMoneyness spot list =
    list
    |> List.map (fun (strike, imp) -> (spot / (float strike),
        (float imp)))
```

Now that we have our data in a tuple, we'll use the spot price of the underlying price, which was 83.2 at the time the data was collected. The `mList` is the list of the converted tuples with moneyness calculated for each one:

```
let list = readVolatilityData
let mList = calcMoneyness 83.2 list

/// Plot values using FSharpChart
fsi.AddPrinter(fun (ch:FSharp.Charting.ChartTypes.GenericChart) ->
    ch.ShowChart(); "FSharpChartingSmile")
```

If you want to reproduce the previous plot, you can run the following line in F# Interactive:

```
Chart.Point(mlist)
```

The final step is to calculate the regression coefficients and use these to calculate the points for our curve. Then we will use a combined plot with the points and the fitted curve:

```
/// Sample points
//let xdata = [ 0.0; 1.0; 2.0; 3.0; 4.0 ]
//let ydata = [ 1.0; 1.4; 1.6; 1.3; 0.9 ]

let xdata = mlist |> Seq.map (fun (x, _) -> x) |> Seq.toList
let ydata = mlist |> Seq.map (fun (_, y) -> y) |> Seq.toList

let N = xdata.Length
let order = 2

/// Generating a Vandermonde row given input v
let vandermondeRow v = [for x in [0..order] do yield v ** (float x)]

/// Creating Vandermonde rows for each element in the list
let vandermonde = xdata |> Seq.map vandermondeRow |> Seq.toList

/// Create the A Matrix
let A = vandermonde |> DenseMatrix.ofRowsList N (order + 1)
A.Transpose()

/// Create the Y Matrix
let createYVector order l = [for x in [0..order] do yield l]
let Y = (createYVector order ydata |> DenseMatrix.ofRowsList (order +
1) N).Transpose()

/// Calculate coefficients using least squares
let coeffs = (A.Transpose() * A).LU().Solve(A.Transpose() *
Y).Column(0)

let calculate x = (vandermondeRow(x) |> DenseVector.ofList) * coeffs

let fitxs = [(Seq.min xdata).. 0.01 ..(Seq.max xdata)]
let fitys = fitxs |> List.map calculate
let fits = [for x in [(Seq.min xdata).. 0.01 ..(Seq.max xdata)] do
yield (x, calculate x)]
```

This is the code line to produce the combined plot with a title. The result will look as shown in the following screenshot:

```
let chart = Chart.Combine [Chart.Point(mlist); Chart.Line(fits)].  
WithTitle("Volatility Smile")]
```

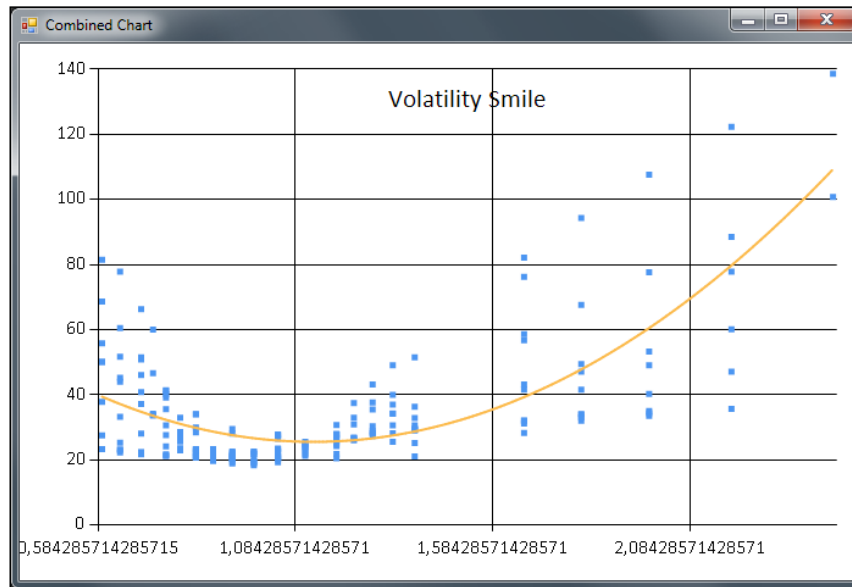


Figure 5: Volatility smile with polynomial regression curve

Summary

In this chapter, we looked into using F# for investigating different aspects of volatility. Volatility is an interesting dimension of finance where you quickly dive into complex theories and models. Here it's very much helpful to have a powerful tool such as F# and F# Interactive. We've just scratched the surface of options and volatility in this chapter. There is a lot more to cover, but that's outside the scope of this book.

Most of the content here will be used in the trading system that we will develop in the following chapters of this book. In the next chapter, we'll begin to look at the first pragmatic building blocks of our trading system: orders and order types. We'll also cover some aspects of pre-trade risk and how F# can help us model these.

7

Getting Started with Order Types and Market Data

In this chapter you will learn how to model order types and market data and various types of feeds in F#.

In this chapter you will learn:

- Modeling Order Types
- Modeling Market data
- Implementing simple pretrade risk analysis
- Using function composition and currying

Introducing orders

Typically, orders are instructions from the buyer's side to the seller's side regarding how to buy and sell financial instruments. These orders are standardized to some extent. In most situations, the orders and order types are determined by the broker used and the exchange.

Order types

Order types are the types of orders that are found in trading systems and on exchanges in general. Often you look at market orders, limit orders, and conditional orders. In the conditional orders category, we have stop-orders and other orders with special conditions for execution.

Other types exist as well, but they are considered to be synthetic orders because they are combinations of the types previously described. Order types can be one of the following:

- Market order
- Limit order
- Conditional and stop-orders

Market orders

Market orders are orders to be executed at the current market price on an exchange. These orders will accept the current bid/ask (top of book) price for the particular instrument. Furthermore, market orders are the simplest of the order types presented here. Due to the uncertainty in the price for which the order will be executed, market orders are not used where a more sophisticated risk profile is maintained. This means that their uses are quite limited in reality.

Limit orders

Limit orders are the most commonly used order type. They are, as the name suggests, limited to a fixed price. This means that the order has a limit price for which they will be executed. For buy orders, this means that there is a limit, and they are bounded above the limit price. All prices below this limit will be accepted for buy orders. For sell orders, the opposite holds; there is a lower limit and the prices are bounded below. All prices above this limit will be accepted for sell orders.

Conditional and stop-orders

Conditional orders, especially stop-orders, are orders that will be activated on the exchange on certain conditions. Stop-orders are activated when a certain price is touched by the order book (top of book). When this condition is met, the order will be converted into a market order. Certain stop-orders that will be converted into limit orders also exist.

Order properties

Order properties refer to the properties of an order describing what to do and under which conditions. The following table presents the most basic properties with descriptions and examples for each one.

Property	Description	Examples
Order side	Whether the order is a buy or sell order with short sell orders included	Buy, Sell
Order type	Type of the order	Market, Limit, and Stop
Order status	The status for the order in the execution flow	Created, New, Filled, Canceled, and so on
Tif	Time in force; how long an order will remain active	GoodForDay, FillOrKill
Quantity	Number of units to buy or sell of the specific instrument	10, 25, and 200
Price	The limit price for the order	26.50, 55.10
Instrument	The instrument to be used in the order	MSFT, GOOG, AAPL
Stop price	The stop price used in the order	26.50, 55.10
Timestamp	The date and time when the order was created	2013-09-14 11:30:44

Next, we'll continue the work we started out in *Chapter 2, Learning More About F#*, and extend the order class and describe some properties as discriminated unions. First off, we will revisit the order side:

```
/// Order side
type OrderSide =
    Buy | Sell | Sellshort
```

In this context, the order side will have three alternatives; buy, sell, and sell short. Then we have the order type, which will either be buy, sell, stop, or stop limit. The stop limit is a stop order that will be converted into a limit order.

```
/// Order type
type OrderType =
    Market | Limit | Stop | StopLimit
```

Discriminated unions are elegant ways of describing data with different values. The values can be many, without losing the readability. Next, we have the order status:

```
type OrderStatus =
    Created | New | Filled | PartiallyFilled | DoneForDay | Cancelled
    | Replaced | PendingCancel | Stopped | Rejected | Suspended |
    PendingNew | Calculated | Expired
```


There are many values for the order status; they are from the FIX 4.2 specification and will be familiar to most of you who are familiar to FIX. They are here for illustrative purposes and will be used in the trading system. **Time in force (Tif)** has the most common alternatives used in trading:

```
type Tif =  
    GoodForDay | GoodTilCancelled | ImmediateOrCancel | FillorKill
```

We'll introduce a new type here to be used along with the functionality introduced in this section. First, we will look at the following code:

```
/// Validation result, ok or failed with message  
type Result = Valid of Order | Error of string
```

Note the use of the `Result` in the preceding snippet; this is used so that we identify the outcome of the validation function(s), which we'll introduce later on. For now, you can think of them as a way of representing alternative return types.

The following order class is an extension on the work we did in *Chapter 2, Learning More About F#*. It's implemented using some mutability, which is allowed in F# because it's not a pure functional language in that sense. Mutability provides more flexibility and compromises for more pragmatic solutions. Here, we use private member fields and some new fields:

```
/// Order class  
type Order(side: OrderSide, t: OrderType, p: float, tif: Tif, q: int,  
i: string, sp: float) =  
    // Init order with status created  
    let mutable St = OrderStatus.Created  
    let mutable S = side  
    member private this.Ts = System.DateTime.Now  
    member private this.T = t  
    member private this.P = p  
    member private this.tif = tif  
    member private this.Q = q  
    member private this.I = i  
    member private this.Sp = sp  
  
    member this.Status  
        with get() = St  
        and set(st) = St <- st  
  
    member this.Side  
        with get() = S  
        and set(s) = S <- s
```

```
member this.Timestamp
    with get() = this.Ts

member this.Type
    with get() = this.T

member this.Qty
    with get() = this.Q

member this.Price
    with get() = this.P

member this.Tif
    with get() = this.tif

member this.Instrument
    with get() = this.I

member this.StopPrice
    with get() = this.Sp

member this.toggleOrderSide() =
    S <- this.toggleOrderSide(S)

member private this.toggleOrderSide(s: OrderSide) =
    match s with
    | Buy -> Sell
    | Sell -> Buy
    | Sellshort -> Sellshort

static member (~-) (o : Order) =
    Order(o.toggleOrderSide(o.Side), o.Type, o.Price, o.tif, o.Q,
    o.I, o.Sp)
```

The preceding method will toggle the side of the order, which is useful if we, for example, want to change the side of the order without creating a new one.

The preceding class can be used in the same way as other classes in F#:

```
// Limit buy order
let buyOrder = Order(OrderSide.Buy, OrderType.Limit, 54.50, Tif.
FillorKill, 100, "MSFT", 0.0)
```

The fields will be as follows, where the private fields are hidden and the property getter functions are the names of the fields:

```
val it : Order = FSI_0050+Order {Instrument = "MSFT";
                                Price = 54.5;
                                Qty = 100;
                                Side = Buy;
                                Status = Created;
                                StopPrice = 0.0;
                                Tif = FillorKill;
                                Timestamp = 2013-09-14 12:01:57;
                                Type = Limit;}
```

Suppose you want to validate an order for correctness and avoid simple mistakes. For example, if the order is a limit order, we can check whether the order has a price above zero. Furthermore, if the order is a stop order, the stop price has to be greater than zero, and so on. Let's write a function to execute this order:

```
/// Validates an order for illustrative purposes
let validateOrder (result:Result) : Result=
    match resultwith
    | Error s -> Error s
    | Valid order ->
        let orderType = order.Type
        let orderPrice = order.Price
        let stopPrice = order.StopPrice
        match orderType with
        | OrderType.Limit ->
            match orderPrice with
            | p when p > 0.0 -> Valid order
            | _ -> Error "Limit orders must have a price > 0"
        | OrderType.Market -> Valid order
        | OrderType.Stop ->
            match stopPrice with
            | p when p > 0.0 -> Valid order
            | _ -> Error "Stop orders must have price > 0"
        | OrderType.StopLimit ->
            match stopPrice with
            | p when p > 0.0 && orderPrice > 0.0 -> Valid order
            | _ -> Error "Stop limit orders must both price > 0 and
            stop price > 0"
```

The function will take an order object wrapped in the order monad. This will become clear when we look at how to use it in the following code:

1. We can create some orders of various types and see if our validation function works.

```
// Limit buy order
let buyOrder = Order(OrderSide.Buy, OrderType.Limit, 54.50, Tif.FillorKill, 100, "MSFT", 0.0)

// Limit buy order, no price
let buyOrderNoPrice = Order(OrderSide.Buy, OrderType.Limit, 0.0, Tif.FillorKill, 100, "MSFT", 0.0)

// Stop order that will be converted to limit order, no limit price
let stopLimitNoPrice = Order(OrderSide.Buy, OrderType.StopLimit, 0.0, Tif.FillorKill, 100, "MSFT", 45.50)

// Stop order that will be converted to market order
let stopNoPrice = Order(OrderSide.Buy, OrderType.Stop, 0.0, Tif.FillorKill, 100, "MSFT", 45.50)
```

2. Now, we will test our validation function for the preceding orders just created:

```
// Validate sample orders
validateOrder (Valid buyOrder) // Ok
validateOrder (Valid buyOrderNoPrice) // Failed
validateOrder (Valid stopLimitNoPrice) // Failed
validateOrder (Valid stopNoPrice) // Ok
```

3. Let's suppose we also want a validator to check whether the instrument is set for an order. We can then run these tests one by one. First, we need a function to do the validation:

```
let validateInstrument (result:Result) : Result =
    match result with
    | Error l -> Error l
    | Valid order ->
        let orderInstrument = order.Instrument
        match orderInstrument.Length with
        | 1 when l > 0 -> Valid order
        | _ -> Error "Must specify order Instrument"
```

4. Running this in F# Interactive will result in the following:

```
> validateInstrument (Valid stopNoPriceNoInstrument);;  
val it : Result = Error "Must specify order Instrument"
```

5. Now we want to combine the two validators into one validator. This can be done in F# using function composition. Function composition, with the `>>` operator, lets you take two or more functions and combine them into one new function. You can look at function composition as a way of chaining functions together. This is useful when smaller building blocks are used and reused and supports modularity.

```
/// Composite validator  
let validateOrderAndInstrument = validateOrder >>  
validateInstrument
```

6. This function can now be used in the same way as the one we used previously:

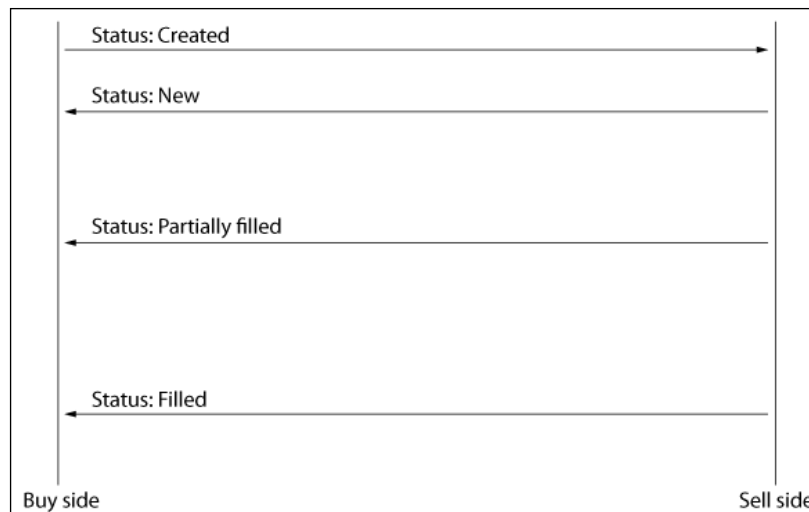
```
// Stop order that will be converted to market order  
let stopNoPriceNoInstrument = Order(OrderSide.Buy, OrderType.Stop,  
0.0, Tif.FillorKill, 100, "", 45.50)  
  
validateOrderAndInstrument (Valid stopNoPriceNoInstrument)
```

This is a rather powerful way of combining functionality and can be useful in many situations. We'll come back to function composition later on when we look at pretrade risks.

Understanding order execution

Let's look at order execution and order flow for a moment. Order execution is where the orders are executed, typically at an exchange. For algorithmic trading, the buy side (trader) will often have their own order management system or order execution engine at the exchange, close to the exchange's trading servers. The orders at the exchange are typically limit orders; there are other types too. All other order types are considered to be synthetic orders or non-native orders. If you use a broker, you will never see this. But for high frequency trading, limit orders are considered to be the only native type. Market orders are simply limit orders with the current market price (from the top of book).

The following illustration shows a simple order flow between an order execution engine and the exchange. The order execution engine resides on the buy side and keeps track of the orders that are currently in and their status.



Order execution flow and status updates for an order

The order status of an order is represented by the discriminated union we have seen before:

```
type OrderStatus =
  Created | New | Filled | PartiallyFilled | DoneForDay | Cancelled
  | Replaced | PendingCancel | Stopped | Rejected | Suspended |
  PendingNew | Calculated | Expired
```

The property will be updated for the order management system when an execution report is received for that particular order. First the order has the order status `Created` and then if it is accepted, it will have the order status `New`. Then, it will have any of the order statuses from the preceding `OrderStatus` object; for example `Filled`, `Expired`, `Rejected`, and so on.

Introducing market data

Market data is the data representing the current bid/ask for a financial instrument at an exchange. The market data can either be the top of book, best bid/ask for the particular instrument, or an aggregated book with several levels of depth. Normally, we just look at the best bid/ask prices, top of book called quotes. Market data can also be sent as OHLC bars or candles. Such bars are only useful for visualizing price information in charts or for trading at a longer horizon. It is a simple way of filtering the price information. The midpoint is defined as the average between the bid and ask.

```
Type Quote =
{
  bid : float
```

```
    ask : float
  } member this.midpoint() = (this.bid + this.ask) / 2.0
```

Let's see how to use this type:

```
let q = {bid = 1.40; ask = 1.45} : Quote
```

We can calculate the midpoint as follows:

```
> q.midpoint();;
val it : float = 1.425
```

We can extend the `Quote` type to have a built-in function to calculate the spread for that quote:

```
type Quote =
  {
    bid : float
    ask : float
  }
  member this.midpoint() = (this.bid + this.ask) / 2.0
  member this.spread() = abs(this.bid - this.ask)
```

```
let q2 = {bid = 1.42; ask = 1.48} : Quote
```

```
> q2.midpoint();;
val it : float = 1.45
> q2.spread();;
val it : float = 0.06
```

Sometimes, in real systems, the bid and ask are not sent in the same object because updates to them will not occur simultaneously. Then, it's preferable to separate them:

```
// Often data is just sent from the feed handler, as bid or ask
type LightQuote =
  | Bid of float | Ask of float

let lqb = Bid 1.31
let lqa = Ask 1.32
```

Implementing simple pretrade risk analysis

In this section, we'll briefly cover pretrade risks for a simple trading system. Pretrade risks are everything that are considered to be risks and analyzed before the order is sent to the exchange. Typically, this is done in the trading engine or order execution engine or order management system. Here we'll consider basic pretrade risk measurements, such as order size and maximum distance for limit prices (that is values that tend to be high/low from the current market price).

The following are the examples of pretrade risk rules:

- Limit the order price distance from the current market price
- The maximum order value
- Total exposure
- Number of orders sent per time unit

Validating orders

Let's dive into some code and look at how pretrade risk can be implemented using functional programming and function composition. We'll look at two simple pretrade risk rules. The first one is for the total order value and the other one is for checking if the limit price is set on the favorable side of the current market price. This can be useful for manual trading, for example.

```
let orderValueMax = 25000.0; // Order value max of $25,000

// A simple pre trade risk rule
let preTradeRiskRuleOne (result:Result) : Result =
  match result with
  | Error l -> Error l
  | Valid order ->
    let orderValue = (float order.Qty) * order.Price
    match orderValue with
    | v when orderValue > orderValueMax -> Error "Order value
exceeded limit"
    | _ -> Valid order
```


The next rule we will use is currying, which is a concept we talked about in *Chapter 2, Learning More About F#*. Currying is a way of calling a function where parts of the functions arguments are saved to be specified later. The following is the second rule for our pretrade risk analysis:

```
// Using currying
let preTradeRiskRuleTwo (marketPrice:float) (result:Result) : Result =
    match result with
    | Error l -> Error l
    | Valid order ->
        let orderLimit = (float order.Qty) * order.Price
        match orderLimit with
        | v when orderLimit < marketPrice && order.Side = OrderSide.
Buy -> Error "Order limit price below market price"
        | v when orderLimit > marketPrice && order.Side = OrderSide.
Sell -> Error "Order limit price above market price"
        | _ -> Valid order
```

We will now use function composition, as before, and create new rules from existing ones:

```
let validateOrderAndInstrumentAndPreTradeRisk =
    validateOrderAndInstrument >> preTradeRiskRuleOne

let validateOrderAndInstrumentAndPreTradeRisk2 marketPrice
= validateOrderAndInstrument >> preTradeRiskRuleOne >>
    (preTradeRiskRuleTwo marketPrice)

validateOrderAndInstrumentAndPreTradeRisk (Valid
    stopNoPriceNoInstrument)
validateOrderAndInstrumentAndPreTradeRisk (Valid
    buyOrderExceedsPreTradeRisk)
validateOrderAndInstrumentAndPreTradeRisk2 25.0 (Valid
    buyOrderBelowPricePreTradeRisk)
```

This pattern can quickly become quite impractical. However, there is an elegant technique to be used where we first specify the functions in a list and then use `List.reduce` and the composite operator to create a new composition:

```
/// Chain using List.reduce
let preTradeRiskRules marketPrice = [
    preTradeRiskRuleOne
    (preTradeRiskRuleTwo marketPrice)
]

/// Create function composition using reduce, >>, composite operator
```

```
let preTradeComposite = List.reduce (>>) (preTradeRiskRules 25.0)

preTradeComposite (Valid buyOrderExceedsPreTradeRisk)
preTradeComposite (Valid buyOrderBelowPricePreTradeRisk)
```

Introducing FIX and QuickFIX/N

In this section we'll learn about the FIX 4.2 standard and the QuickFIX/N library for .NET. FIX is the standard protocol of communicating with brokers and exchanges and stands for **Financial Information eXchange**. It has been around since the early 90s and uses an ASCII-based representation of messages. Other alternatives exist, such as proprietary APIs and protocols.

Using FIX 4.2

In this section we'll refactor the existing trading system in order to use FIX 4.2 messages.

1. Download FIXimulator from the following URL:
<http://fiximulator.org/>
2. Download QuickFIX/n from the following URL:
<http://www.quickfixn.org/download>.
3. Extract the files from the archives into folders.
4. Start FIXimulator by running `fiximulator.bat`.

When you have started FIXimulator, it will look like the following figure. At the bottom of the application, there is a status bar with indicators. The leftmost indicator is for the client connection status. The first step is to make a successful connection to the simulator.

Configuring QuickFIX to use the simulator

To be able to connect to the FIXimulator, you need a proper configuration file, `config.cfg`, in the classpath of the project as well as the correct path in the `SessionSettings` constructor in the following code:

The contents of `config.cfg` are:

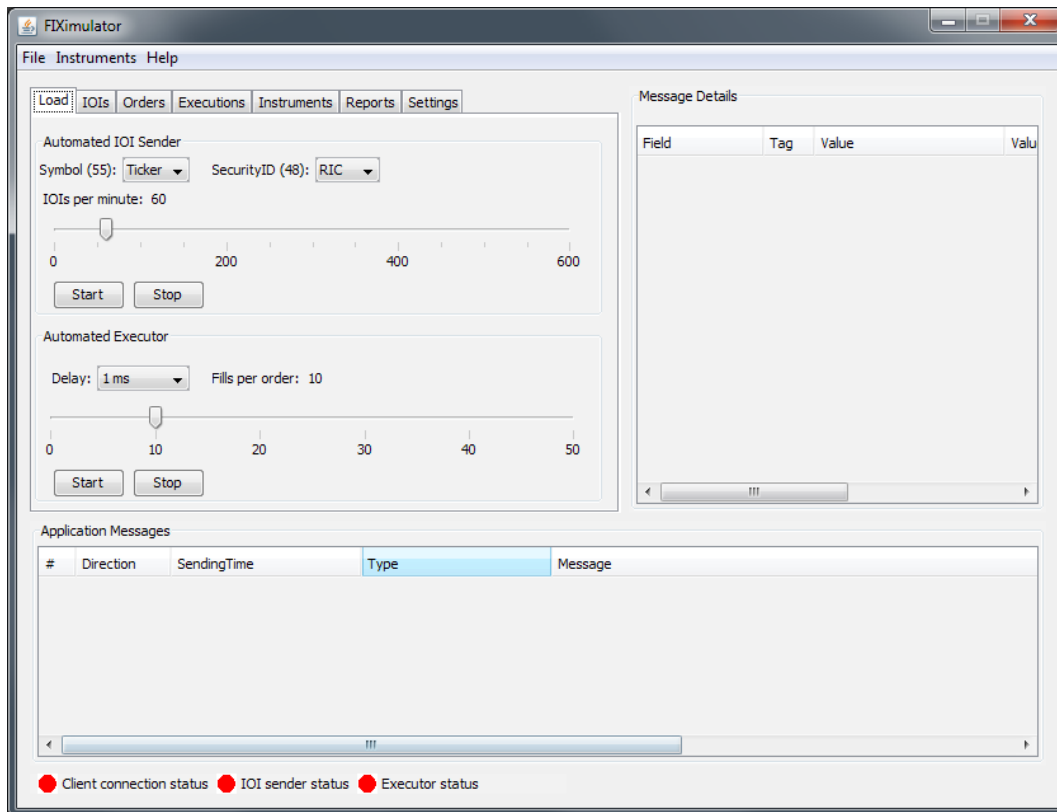
```
[DEFAULT]
ConnectionType=initiator
ReconnectInterval=60
SenderCompID=TRADINGSYSTEM

[SESSION]
BeginString=FIX.4.2
TargetCompID=FIXIMULATOR
StartTime=00:00:00
EndTime=00:00:00
HeartBtInt=30
ReconnectInterval=10
SocketConnectPort=9878
SocketConnectHost=192.168.0.25
FileStorePath=temp
ValidateUserDefinedFields=N

ResetOnLogon=Y
ResetOnLogout=Y
ResetOnDisconnect=Y
```

You have to change the value of the `SocketConnectHost` field to suit your setup, and this will be your local IP address if you run the simulator on the same machine.

There is also a configuration file for the simulator, namely `FIXimulator.cfg`, where the value of `TargetCompID` has to be changed to `TRADINGSYSTEM`.



No client is connected

On the client side, in our program, we'll use the QuickFIX library and implement the methods needed in order to connect to the simulator. Add the following code to a new file `FIX.fs`. We'll use this in the final project, so it's recommended to work from the TradingSystem project while doing this project, which will start in the next chapter.

```
namespace FIX
    open System
    open System.Globalization
    open QuickFix
    open QuickFix.Transport
    open QuickFix.FIX42
    open QuickFix.Fields
```

To use QuickFIX, a number of interfaces have to be implemented. The ClientInitiator function is where the messages will be handled:

```
module FIX =
    type ClientInitiator() =
        interface IApplication with
            member this.OnCreate(sessionID : SessionID) : unit =
                printfn "OnCreate"
            member this.ToAdmin(msg : QuickFix.Message, sessionID :
                SessionID) : unit = printf "ToAdmin"
            member this.FromAdmin(msg : QuickFix.Message, sessionID :
                SessionID) : unit = printf "FromAdmin"
            member this.ToApp(msg : QuickFix.Message, sessionID :
                SessionID) : unit = printf "ToApp"
            member this.FromApp(msg : QuickFix.Message, sessionID :
                QuickFix.SessionID) : unit = printfn "FromApp -- %A" msg
            member this.OnLogout(sessionID : SessionID) : unit =
                printf "OnLogout"
            member this.OnLogon(sessionID : SessionID) : unit = printf
                "OnLogon"
```

The ConsoleLog function is another interface needed to support logging.

```
type ConsoleLog() =
    interface ILog with
        member this.Clear() : unit = printf "hello"
        member this.OnEvent(str : string) : unit =
            printfn "%s" str
        member this.OnIncoming(str : string) : unit =
            printfn "%s" str
        member this.OnOutgoing(str : string) : unit =
            printfn "%s" str

type ConsoleLogFactory(settings : SessionSettings) =
    interface ILogFactory with
        member this.Create(sessionID : SessionID) : ILog =
            new NullLog() :> ILog
```

Finally, we have FIXEngine itself. This is the interface provided to the other parts of the system. It provides methods to start, stop, and send orders.

```
type FIXEngine() =
    let settings = new SessionSettings(@"conf\config.cfg")
    let application = new ClientInitiator()
    let storeFactory = FileStoreFactory(settings)
    let logFactory = new ConsoleLogFactory(settings)
    let messageFactory = new MessageFactory()
```

```

let initiator = new SocketInitiator(application, storeFactory,
settings)
let currentID = initiator.GetSessionIDs() |> Seq.head
member this.init() : unit =
    ()
member this.start() : unit =
    initiator.Start()
member this.stop() : unit =
    initiator.Stop()

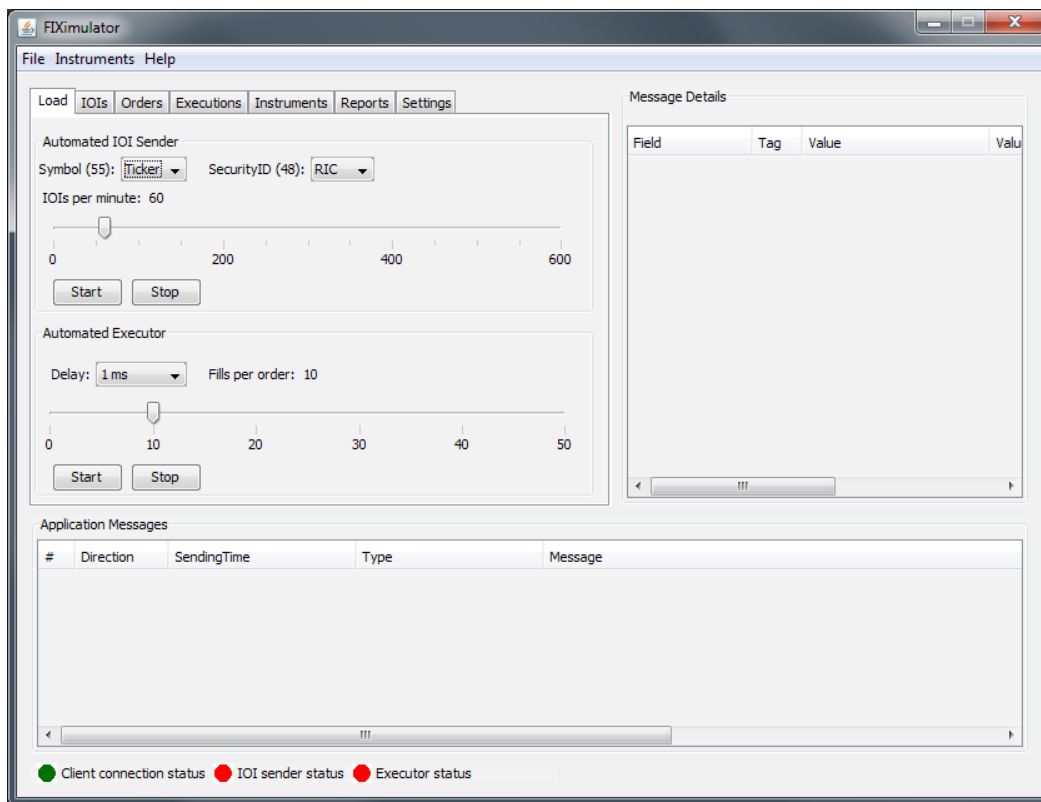
```

Now we can try the preceding code using F# Interactive, and we will be connected to the simulator. The client connection status in the **FIXimulator** window will now be green if everything worked out successfully.

```

let fixEngine = new FIX.FIXEngine()
fixEngine.init()
fixEngine.start()

```



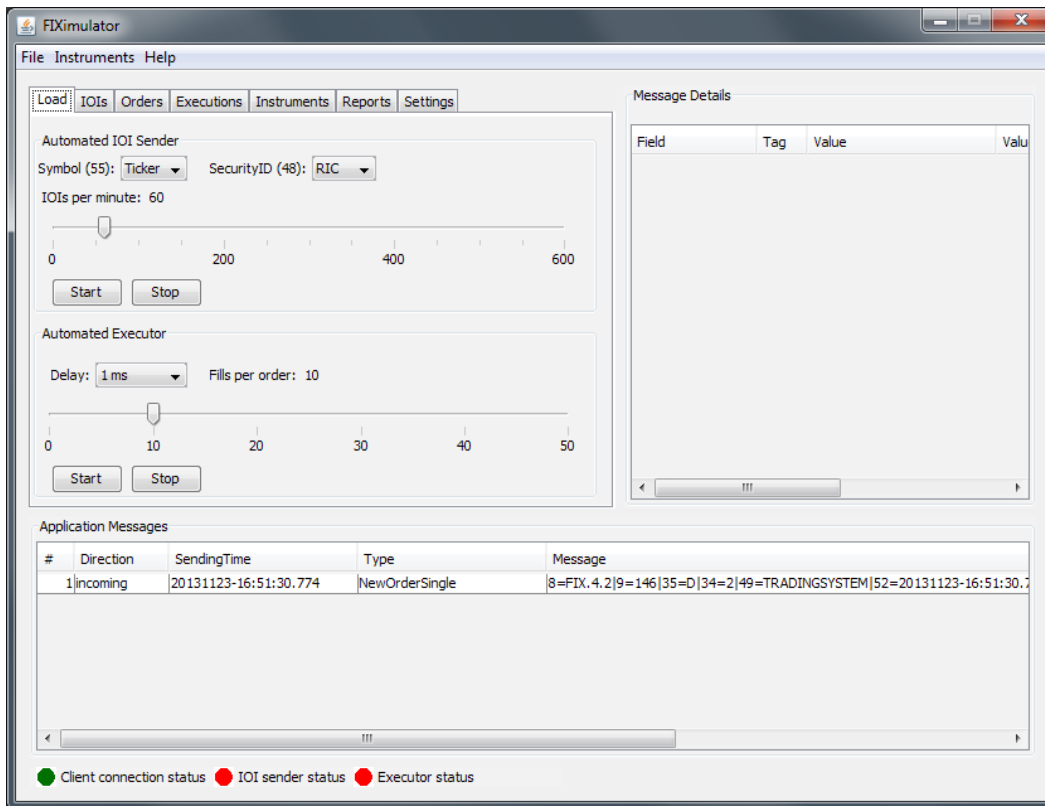
The client connection status is green when the client is connected

Let's make things more interesting and try out some code to send an order to the simulator. As previously mentioned, orders have various properties. The fields in our `Order` object are supposed to mimic the FIX 4.2 fields as closely as possible. We'll first try out the QuickFIX representation of orders, and send a limit order to the simulator to see if everything works as expected. To do this, we'll add a method to `FIXEngine`:

```
member this.sendTestLimitOrder() : unit =
    let fixOrder = new NewOrderSingle()
    fixOrder.Symbol <- new Symbol("ERICB4A115")
    fixOrder.ClOrdID <- new ClOrdID(DateTime.Now.Ticks.ToString())
    fixOrder.OrderQty <- new OrderQty(decimal 50)
    fixOrder.OrdType <- new OrdType('2'); // Limit order
    fixOrder.Side <- new Side('1');
    fixOrder.Price <- new Price(decimal 25.0);
    fixOrder.TransactTime <- new TransactTime();
    fixOrder.HandlInst <- new HandlInst('2');
    fixOrder.SecurityType <- new SecurityType("OPT"); // Option
    fixOrder.Currency <- new Currency("USD");
    // Send order to target
    Session.SendToTarget(fixOrder, currentID) |> ignore
```

This method will just be used for testing purposes and it is useful to have a working reference when we later add support to translate our internal `Order` object into the representation used by QuickFIX. You have to restart the F# Interactive session to see the changes in `FIXEngine` that we did earlier. Run the following code, and you'll hopefully have sent your first order to the simulator:

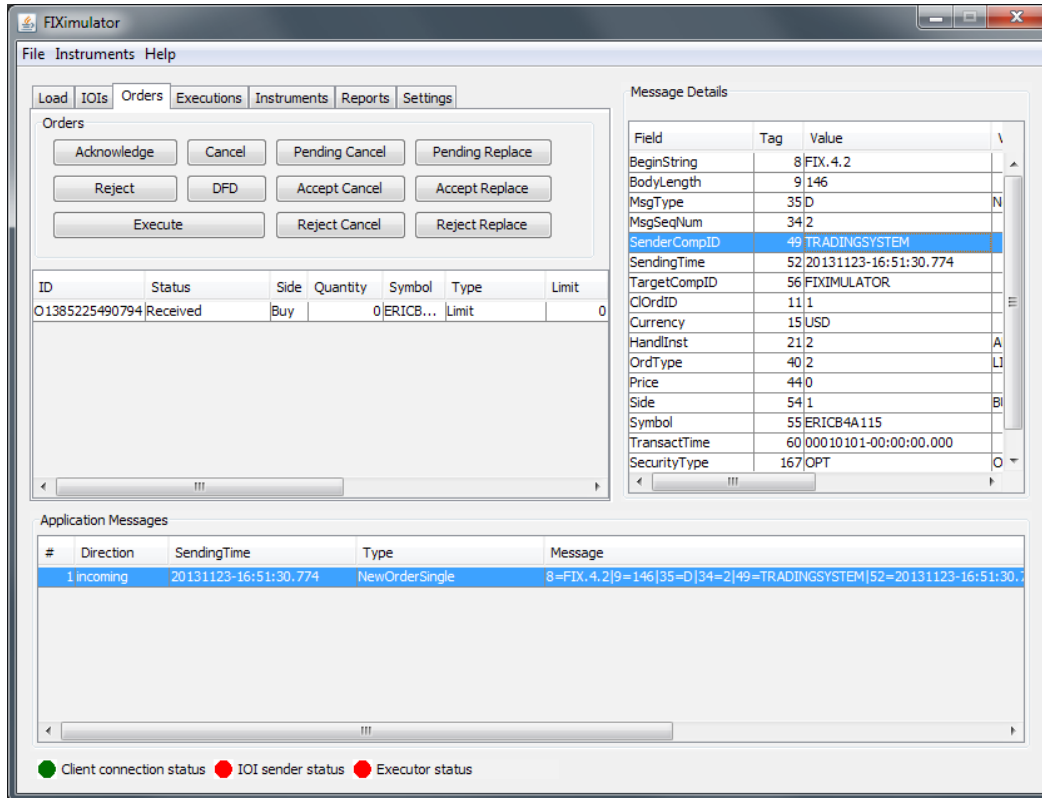
```
let fixEngine = new FIX.FIXEngine()
fixEngine.init()
fixEngine.start()
fixEngine.sendTestLimitOrder()
```



The client connection status is green when the client is connected

Let's stay in **FIXimulator** for a moment and see what useful functionality we got.

You can select the **Orders** tab in the simulator and you will have a new view, as shown in the following screenshot:



Investigating the order properties from incoming orders

The `SenderCompID` field corresponds to the actual sender of the order, which is the `TradingSystem`. We can also see the other properties of the order. Inside the simulator, you have several alternatives for each order, for example:

- Acknowledged
- Canceled
- Rejected
- Executed

We need some support for keeping track of the orders in our system and to change their state depending on the messages received from the counterpart (simulator); this is done by using the `ClientInitiator` function. Here we have the method called `FromApp`, which is a method to handle incoming messages; for example, modify the state of the orders in the system. The message to listen to, in the case of order execution management, is `ExecutionReport`. It is exactly what the name suggests, a report of execution. First we'll implement the code to just print out the status from the last order execution in the simulator. Then we'll use the simulator to test it out.

```
member this.FromApp(msg : QuickFix.Message, sessionID : QuickFix.
SessionID) : unit =
    match msg with
    | :? ExecutionReport as report ->
        match report.OrdStatus.getValue() with
        | OrdStatus.NEW ->
            printfn "ExecutionReport (NEW) %A" report
        | OrdStatus.FILLED ->
            printfn "ExecutionReport (FILLED) %A" report
        | OrdStatus.PARTIALLY_FILLED ->
            printfn "ExecutionReport (PARTIALLY_FILLED) %A" report
        | OrdStatus.CANCELED ->
            printfn "ExecutionReport (CANCELED) %A" report
        | OrdStatus.REJECTED ->
            printfn "ExecutionReport (REJECTED) %A" report
        | OrdStatus.EXPIRED ->
            printfn "ExecutionReport (EXPIRED) %A" report
        | _ -> printfn "ExecutionReport (other) %A" report
    | _ -> ()
```

In the preceding code, we try to cast `msg` to an `ExecutionReport` instance and if that succeeds, we pattern match on the order status. Let's try this together with the simulator.

1. Send a test order to the simulator using `fixEngine.sendTestLimitOrder()`.
2. In **FIXimulator**, under **Orders**, select the received order.
3. Press the **Acknowledge** button.
4. F# Interactive will output the execution report, which will look something like:

```
ExecutionReport (NEW) seq [[6, 0]; [11, 1]; [14, 0]; [17,
E1385238452777]; ...]
```

This means we have received an execution report, and the order status is now `New`.

5. Select the same order in the simulator and click on **Cancel**.
`ExecutionReport (CANCELED) seq [[6, 0]; [11, 1]; [14, 0]; [17, E1385238572409]; ...]`
6. We'll repeat the procedure, but this time we'll acknowledge and then execute the order in two steps, resulting in a `New`, `Partial fill`, and `Filled` execution report.
7. Send a test order again.
8. Acknowledge the order.
9. Execute the order with `LastShares=50` and `LastPx=25`.
10. Execute again with `LastShares=50` and `LastPx=24`.
11. Watch the output in F# Interactive.

Note the order in which the execution reports arrive. First, we execute half the order resulting in a partial fill. Then, we execute the remaining 50, which will fill the order.

```
ExecutionReport (PARTIALLY_FILLED) seq [[6, 25]; [11, 1]; [14, 50]; [17, E1385238734808]; ...]
ExecutionReport (FILLED) seq [[6, 24.5]; [11, 1]; [14, 100]; [17, E1385238882881]; ...]
```

In the simulator, we can see the value in the `AvgPx` column is 24.5. This corresponds to the average price for the whole order. Next, we'll implement a lightweight order manager to keep track of all this for us.

Let's look at the modified code for the `FromApp` callback function where we just altered the status of the order matching the order ID:

```
| OrdStatus.NEW ->
  printfn "ExecutionReport (NEW) %A" report
  orders |> Seq.find (fun order -> order.Timestamp.ToString() =
report.ClOrdID.GetValue()) |> (fun order -> order.Status <-
OrderStatus.New)
```

The `orders` list is a `BindingList`, which is just a declaration we use for experimentation outside of the MVC model where it will be located in the final system.

```
/// Use a list of NewOrderSingle as first step
let orders = new BindingList<Order>()
let fixEngine = new FIX.FIXEngine(orders)
fixEngine.init()
fixEngine.start()
let buyOrder1 = Order(OrderSide.Buy, OrderType.Limit, 24.50, Tif.
GoodForDay, 100, "ERICB4A115", 0.0)
```

```

let buyOrder2 = Order(OrderSide.Buy, OrderType.Limit, 34.50, Tif.
  GoodForDay, 100, "ERICB4A116", 0.0)
let buyOrder3 = Order(OrderSide.Buy, OrderType.Limit, 44.50, Tif.
  GoodForDay, 100, "ERICB4A117", 0.0)
fixEngine.sendOrder(buyOrder1)
fixEngine.sendOrder(buyOrder2)
fixEngine.sendOrder(buyOrder3)

```

The `sendOrder` method is also modified slightly so that it can handle our `Order` objects and convert them to the QuickFIX representation:

```

member this.sendOrder(order:Order) : unit =
  let fixOrder = new NewOrderSingle()
  /// Convert to Order to NewOrderSingle
  fixOrder.Symbol <- new Symbol(order.Instrument)
  fixOrder.ClOrdID <- new ClOrdID(order.Timestamp.ToString())
  fixOrder.OrderQty <- new OrderQty(decimal order.Qty)
  fixOrder.OrdType <- new OrdType('2'); /// TODO
  fixOrder.Side <- new Side('1');
  fixOrder.Price <- new Price(decimal order.Price);
  fixOrder.TransactTime <- new TransactTime();
  fixOrder.HandlInst <- new HandlInst('2');
  fixOrder.SecurityType <- new SecurityType("OPT"); /// TODO
  fixOrder.Currency <- new Currency("USD"); /// TODO
  /// Add to OMS
  orders.Add(order)
  /// Send order to target
  Session.SendToTarget(fixOrder, currentID) |> ignore

```

We can now use the code elaborated so far to test out the simple order manager using the simulator. When the three orders are sent to the simulator, the output will look like the following screenshot. Select the first one and click on **Acknowledge**. We can now compare the contents of the orders list to see if the execution report is handled and if the order manager has done its job.

Here is the first `Order` in the orders list:

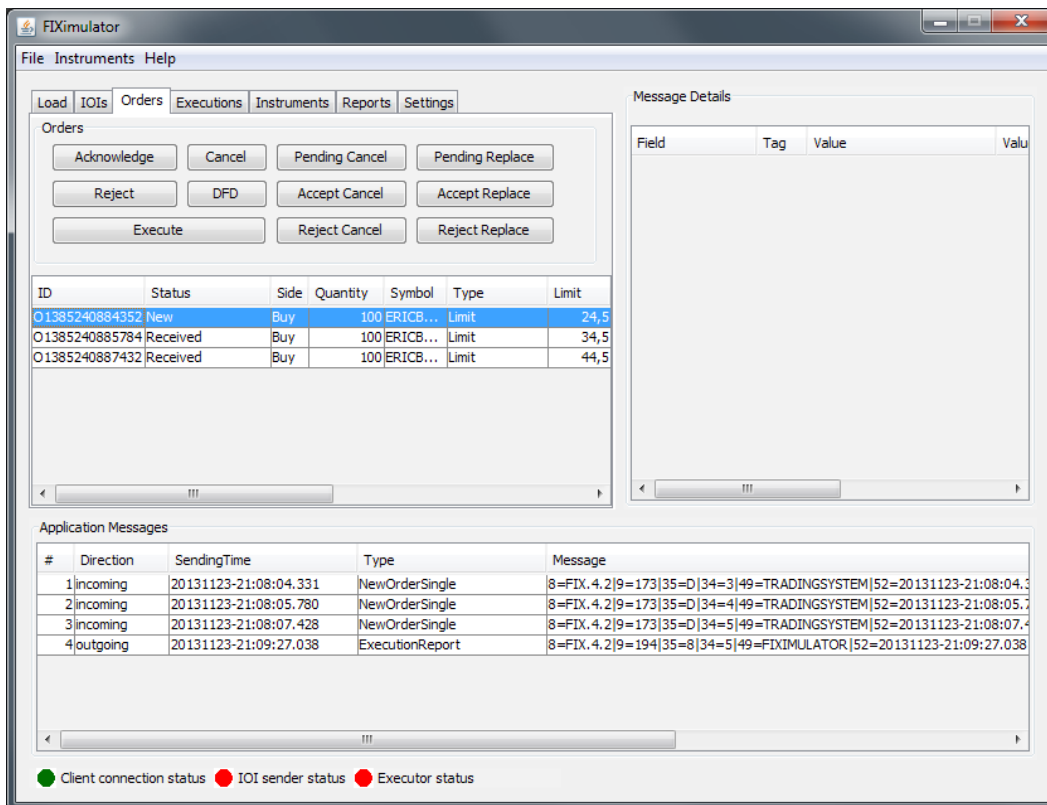
```

{Instrument = "ERICB4A115";
  OrderId = "635208412525809991";
  Price = 24.5;
  Qty = 100;
  Side = Buy;
  Status = Created;
  StopPrice = 0.0;
  Tif = GoodForDay;
  Timestamp = 2013-11-23 22:09:31;
  Type = Limit;}
...

```

This will be changed to the following if everything works out:

```
{Instrument = "ERICB4A115";  
OrderId = "635208412525809991";  
Price = 24.5;  
Qty = 100;  
Side = Buy;  
Status = New;  
StopPrice = 0.0;  
Tif = GoodForDay;  
Timestamp = 2013-11-23 22:09:31;  
Type = Limit;}
```



Testing out the first version of our order manager

The remaining part is to include support for partial fills, where we have open and executed quantities and the average price. This is the final part of the order manager to be elaborated. We'll also enhance the outputs from the `ClientInitiator` function to include these fields from the execution reports.

The following are the modifications done to the code to handle the order management part:

```

member this.findOrder str =
    try
        Some (orders |> Seq.find (fun o -> o.Timestamp = str))
    with | _ as ex -> printfn "Exception: %A" ex.Message; None

member this.FromApp(msg : QuickFix.Message, sessionID : QuickFix.
SessionID) : unit =
match msg with
| :? ExecutionReport as report ->
    let qty = report.CumQty
    let avg = report.AvgPx
    let sta = report.OrdStatus
    let oid = report.ClOrdID
    let lqty = report.LeavesQty
    let eqty = report.CumQty

let debug = fun str -> printfn "ExecutionReport (%s) # avg price:
%s | qty: %s | status: %s | orderId: %s" str (avg.ToString()) (qty.
ToString()) (sta.ToString()) (oid.ToString())

match sta.getValue() with
| OrdStatus.NEW ->
    match this.findOrder(oid.ToString()) with
    | Some(o) ->
        o.Status <- OrderStatus.New
    | _ -> printfn "ERROR: The order was not found in OMS"
        debug "NEW"

```

To handle the status of Filled, we set the average price, the open quantity, and the executed quantity:

```

| OrdStatus.FILLED ->
    /// Update avg price, open price, ex price
    match this.findOrder(oid.ToString()) with
    | Some(o) ->
        o.Status <- OrderStatus.Filled
        o.AveragePrice <- double (avg.getValue())
        o.OpenQty <- int (lqty.getValue())
        o.ExecutedQty <- int (eqty.getValue())
    | _ -> printfn "ERROR: The order was not found in OMS"
        debug "FILLED"

```

To handle the status of `PartiallyFilled`, we set the average price, the open quantity, and the executed quantity:

```
| OrdStatus.PARTIALLY_FILLED ->
  /// Update avg price, open price, ex price
  match this.findOrder(oid.ToString()) with
  | Some(o) ->
    o.Status <- OrderStatus.PartiallyFilled
    o.AveragePrice <- double (avg.getValue())
    o.OpenQty <- int (lqty.getValue())
    o.ExecutedQty <- int (eqty.getValue())
  | _ -> printfn "ERROR: The order was not found in OMS"
  debug "PARTIALLY_FILLED"
```

The remaining status updates are straightforward updates:

```
| OrdStatus.CANCELED ->
  match this.findOrder(oid.ToString()) with
  | Some(o) ->
    o.Status <- OrderStatus.Cancelled
  | _ -> printfn "ERROR: The order was not found in OMS"
  debug "CANCELED"
| OrdStatus.REJECTED ->
  match this.findOrder(oid.ToString()) with
  | Some(o) ->
    o.Status <- OrderStatus.Rejected
  | _ -> printfn "ERROR: The order was not found in OMS"
  debug "REJECTED"
| OrdStatus.REPLACED ->
  match this.findOrder(oid.ToString()) with
  | Some(o) ->
    o.Status <- OrderStatus.Replaced
  | _ -> printfn "ERROR: The order was not found in OMS"
  debug "REPLACED"
| OrdStatus.EXPIRED ->
  printfn "ExecutionReport (EXPIRED) %A" report
| _ -> printfn "ExecutionReport (other) %A" report
| _ -> ()
```

We can test this out using the simulator, and a test run with partial fills, as before, results in the following content in the orders list:

```
{AveragePrice = 0.0;
ExecutedQty = 0;
Instrument = "ERICB4A115";
OpenQty = 0;
Price = 24.5;
Qty = 100;
Side = Buy;
Status = Cancelled;
```

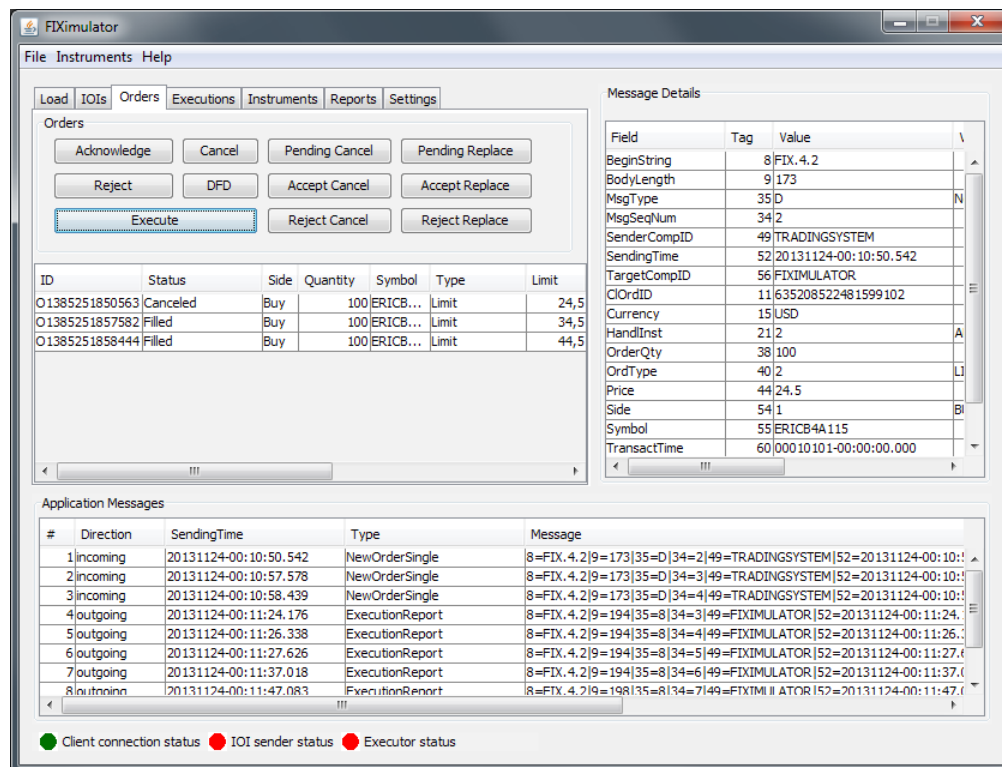
```

...

{AveragePrice = 23.0;
ExecutedQty = 100;
Instrument = "ERICB4A116";
OpenQty = 0;
Price = 34.5;
Qty = 100;
Side = Buy;
Status = Filled;
...

{AveragePrice = 24.5;
ExecutedQty = 100;
Instrument = "ERICB4A117";
OpenQty = 0;
Price = 44.5;
Qty = 100;
Side = Buy;
Status = Filled;
...

```



The final iteration and testing of the order manager

The values will be presented in a data grid in the GUI for the trading system, which we'll start preparing in the next chapter.

Summary

In this chapter we introduced several concepts regarding orders and market data. We also looked at how to model orders and market data in F# using some of the information available from the FIX 4.2 standard to guide us. It's good to have an overview of these facts while working with quantitative finance, especially when dealing with algorithmic trading.

The results from this chapter will be used when we start to set up the trading system, which we'll start doing in the next chapter. We'll also look at how to implement test cases for the validation code provided in this chapter.

8

Setting Up the Trading System Project

In this chapter we'll set up the trading system, which we'll develop throughout the rest of this book. The trading system will summarize the things we've learned so far. It's also a good way of illustrating the power of F# in combination with existing tools and libraries. We'll start out by setting up the project in Visual Studio, then adding the references needed for testing and connecting to Microsoft SQL Server. Type providers and **Language-INtegrated-Query (LINQ)** will be briefly introduced here, and more details will be covered in the next chapter.

In this chapter, we will learn:

- More about automated trading
- Test-driven development
- Requirements for the trading system
- Setting up the project
- Connecting to a database
- Type providers in F#

Explaining automated trading

Automated trading is becoming increasingly popular these days. Most trading strategies can be implemented to be traded by a computer. There are many benefits to automating a trading strategy. The trading strategy can be backtested using historical data. This means the strategy is run on historical data and the performance of the strategy can be studied. We'll not cover backtesting in this book, but the trading system developed here can be modified to support backtesting.

Automated trading systems are, as the name suggests, automated systems for trading that are run on a computer. They often consist of several parts such as feed handlers, order management systems, and trading strategies. Typically, automated trading systems will represent a pipeline from market data to orders to be executed, and keep track of state and history. Rules are written to be executed in near real time for the market data entering the system. It's much like a regular control system, with an input and an output. In the following chapters we'll look at how to implement a rather simple, yet powerful, trading system in F#, wrapping up what we have learnt in this book.

The following are the parts of an automated trading system:

- Feed handlers and market data adapters
- Trading strategies
- Order execution and order management
- Persistence layer (DBs)
- GUI for monitoring and human interaction

Here is the block diagram showing the parts of an automated trading system:

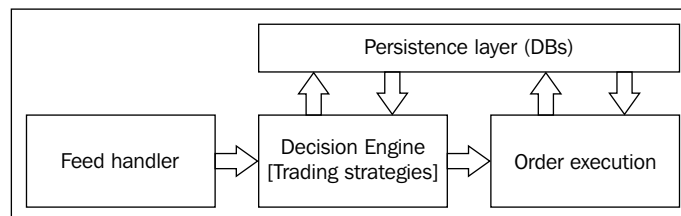


Figure 1: Typical block diagram of a trading system

Understanding software testing and test-driven development

When writing software, it's crucial to be able to test the functionality of the system being written. In software development, there is a popular and effective way of writing code in a more agile fashion, namely test-driven development. This method is driven by tests, and the tests written before the main logic are implemented. In other words, when you are about to write a test case for a system, you will certainly have a couple of requirements already at hand, or an idea about the software. In test-driven development, the tests will reflect the requirements. This is a way of writing the requirements in code that will test the piece of software for a given set of functionality. The tests are implemented as test cases, and test cases are collected

into test suites. The tests will preferably be automated with a tool. Having automated tests will enable the developers to rerun the tests every time a change is made to the code.

We'll focus on unit testing in this chapter using **NUnit**.

Understanding NUnit and FsUnit

NUnit is an open source unit testing framework for all .NET languages, same as JUnit is for Java. NUnit enables the programmer to write unit tests, and execute the test to be able to see which tests are successful and which failed. In our project we'll use NUnit and its external tool for running tests. A typical line for testing in F# using FsUnit can look something like the following:

```
> 1 |> should equal 1;;
```

Requirements for the system

In this section we'll discuss some of the main requirements for the trading system. We'll not specify all the details, because some of them are needed to be divided into parts. The trading system will make use of some libraries and tools, specified in the following section.

Table some of the most important requirements of the trading system we'll develop. It will be a simple system with trading options for volatility arbitrage opportunities using S&P 500 index options and the CBOE Volatility Index (VIX). The S&P 500 index consists of the largest 500 companies listed on NYSE or NASDAQ. It's considered to be a general indicator of the U.S. stock market. The VIX is an index of the implied volatility of S&P 500 index options.

The system should be able to do the following:

- Store log entries in a Microsoft SQL Server database
- Store trade history in a Microsoft SQL Server database
- Download quotes from Yahoo! Finance
- Manage orders
- Send an order using FIX 4.2
- Connect using FIX 4.2 to a FIX simulator
- Execute a trading strategy written in F#
- Be controlled from a basic GUI with start/stop buttons
- Display the current position(s)

- Display the current profit and loss
- Display the latest quote(s)
- Use the MVC pattern

The following are the libraries and tools used:

- QuickFIX/N: The FIX protocol for .NET
- QuantLib: This is a library for quantitative finance
- LINQ
- Microsoft SQL Server
- Windows Forms
- FSharpChart: This is an F# friendly wrapper of Microsoft Chart Controls

Setting up the project

In this section we'll set up the solution in Visual Studio. It will consist of two projects; one project for the trading system and one project for the tests. Separating the two has some advantages and will produce two binaries. The tests will be run from the NUnit program, which is a standalone program for running unit tests.

The following steps will create two projects in the same solution:

1. Create a new F# application, name it **TradingSystem** as shown in the following screenshot:

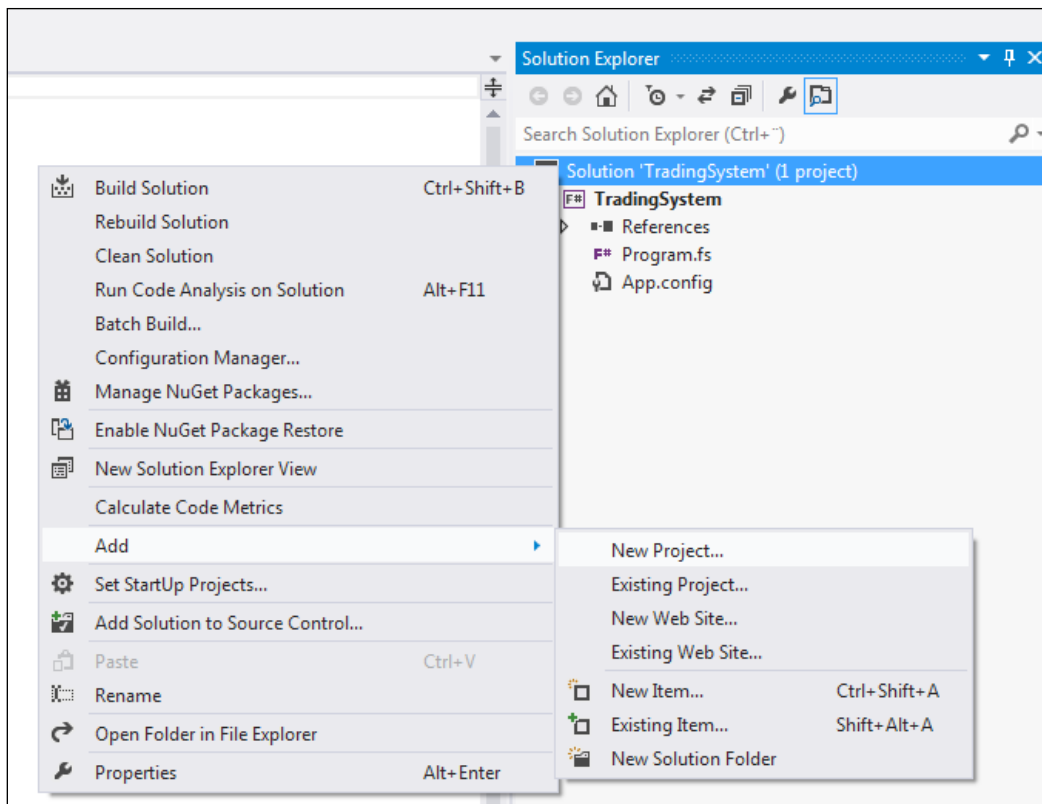


Figure 2: Adding a new project to a solution

2. Add a new project to the existing **TradingSystem** solution. Right-click on the solution as shown in *Figure 2*, and navigate to **Add | New Project...** Create another F# application and name it **TradingSystem.Tests**.

3. Next, we have to add the testing frameworks to the **TradingSystem.Tests** project as shown in the following screenshot:

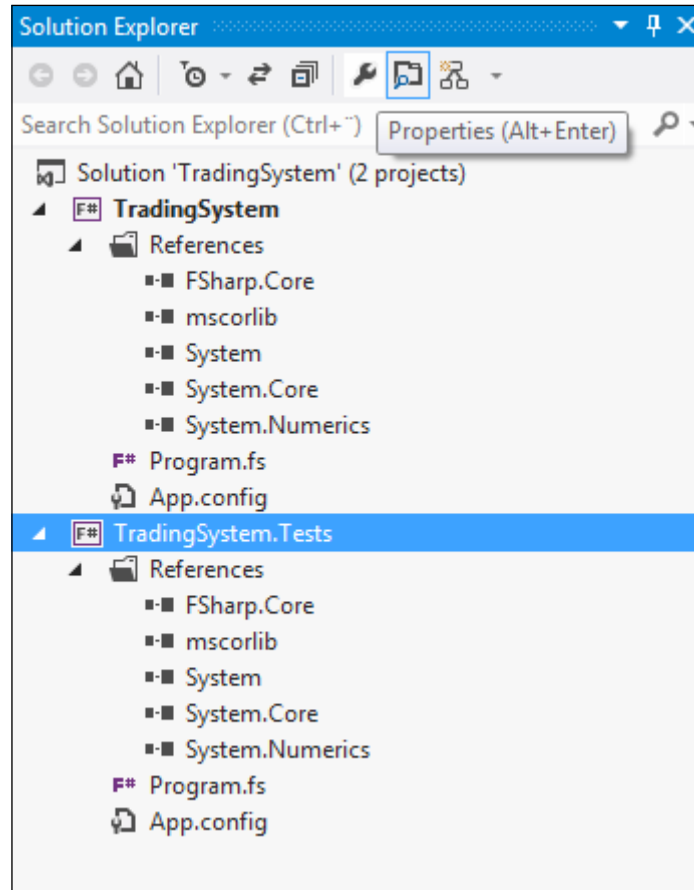


Figure 3: The solution with the two projects

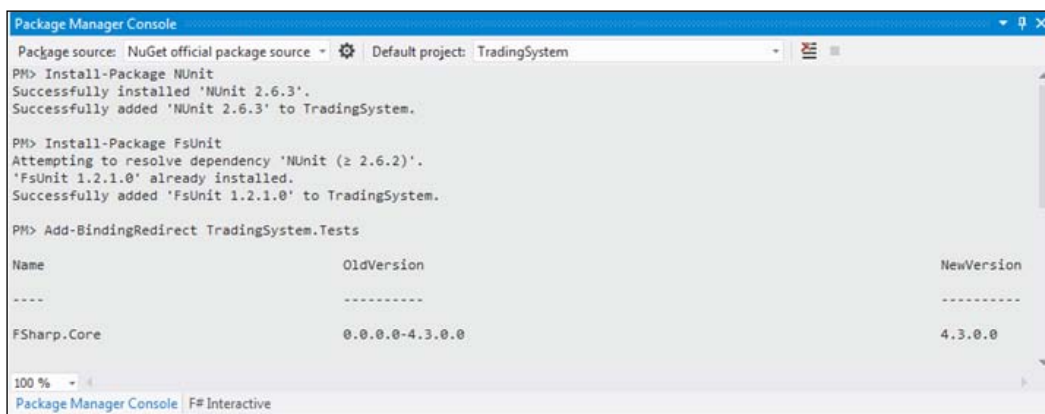
4. Now we have the solution set up with two projects. You can see the references in each of the projects in *Figure 3*. We'll add some more references to both projects in this chapter. Next we'll install the testing frameworks, first NUnit, then FsCheck.

Installing the NUnit and FsUnit frameworks

In this section we'll look at how to install NUnit and FsUnit and how to use it with F# Interactive to verify that everything works as expected.

To install NUnit and FsUnit, follow these steps in Visual Studio 2012:

1. Go to the Package Manager console by navigating to **VIEW | Other Windows | Package Manager Console**.
2. Select **TradingSystem.Tests** as the default project in the drop-down menu.
3. Type in `Install-Package NUnit`.
4. Type in `Install-Package FsUnit`.
5. Type in `Add-BindingRedirect TradingSystem.Tests`. The following screenshot shows the result from the preceding steps:



The last command is to ensure that `Fsharp.Core` is up-to-date, and it will also update `App.config` if necessary.

6. Let's play around with the NUnit framework in F# Interactive to get a feeling of what it is and to see if everything is set up correctly. Add the following code to an F# script file:

```

#r @"[...]\TradingSystem\packages\FsUnit.1.2.1.0\Lib\Net40\FsUnit.
NUnit.dll";;
#r @"[...]\TradingSystem\packages\NUnit.2.6.2\lib\nunit.framework.
dll";;
  
```

```

open NUnit.Framework
open FsUnit
  
```

```

1 |> should equal 1
  
```


- Note that you have to find the path of the two DLLs, because it will differ from machine to machine. Simply go to **References** in your project (**TradingSystem.Tests**), click on the particular framework in **Solution Explorer**, and **Full Path** will be updated in the **Properties** window. Use this **Full Path** in the preceding code, do it for both the DLLs.

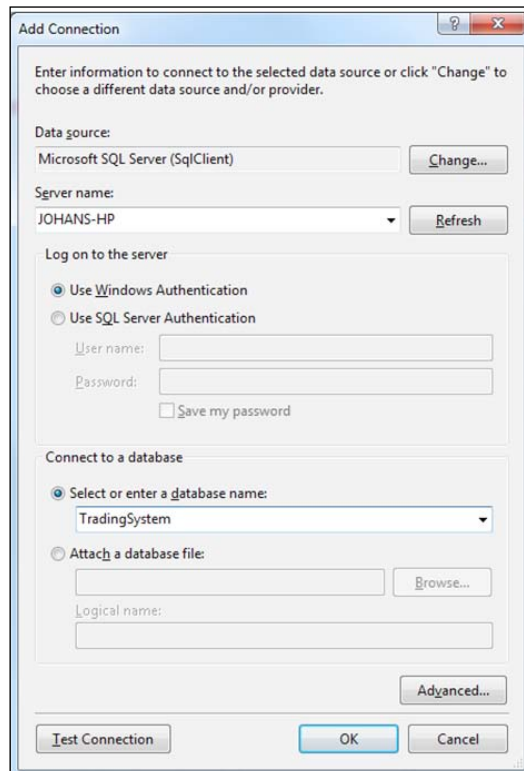
Finally you can test the preceding code:

```
> 1 |> should equal 1;;  
val it : unit = ()
```

This means our first testing framework is set up successfully. Next we'll look at adding some tests in the code and run them from NUnit.

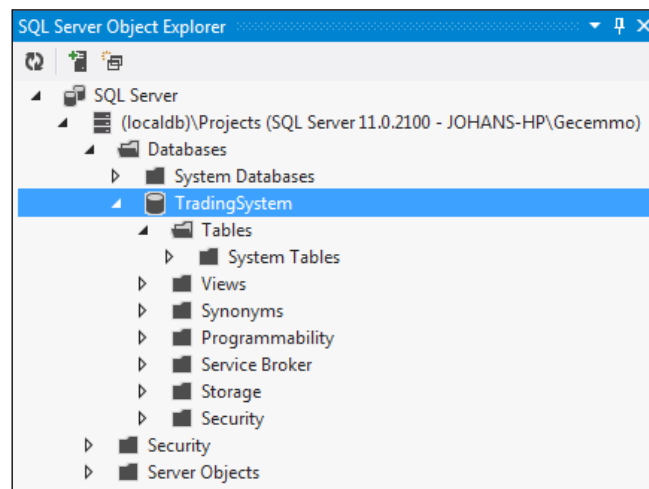
Connecting to Microsoft SQL Server

This chapter will assume you have a running instance of Microsoft SQL Server either on your local machine, as part of Visual Studio 2012, or on a remote machine with access permissions as shown in the following screenshot:



The steps for connecting to Microsoft SQL Server are as follows:

1. Navigate to **VIEW | Server Explorer**.
2. Right-click on **Data Connections**, and choose **Add connection**.
3. Select **Microsoft SQL Server (SqlClient)** as the **Data Source**.
4. Choose the local machine if you have Microsoft SQL Server installed locally.
5. Select **Use Windows Authentication**.
6. Specify the name of the database. From now on, we'll refer to our database as **TradingSystem**.
7. To test whether the setup is successful, press **Test Connection** in the lower-left corner.
8. Press **OK**.
9. Then you will encounter a dialog asking if you want to create it, press **Yes**.
10. Now we have the database for our project, and next we'll add the tables needed for the project. To do this, open **VIEW | SQL Server Object Explorer**. It will look like the following screenshot:



11. To add our tables, we'll use the SQL snippets provided in the next step. Right-click on the **TradingSystem** database, and click on **New Query...**. Paste the following code to create the table `Log`.
12. The following is the SQL code to create the `Log` table:

```
DROP TABLE LOG
CREATE TABLE LOG
(
```

```
    log_id int IDENTITY PRIMARY KEY,  
    log_datetime datetime DEFAULT CURRENT_TIMESTAMP,  
    log_level nvarchar(12) DEFAULT 'info',  
    log_msg ntext  
)
```

13. This will output the following in the SQL terminal below the editor:

```
Command(s) completed successfully.
```

We'll use this table in the next section about type providers and LINQ.

Introducing type providers

Type providers are powerful ways of dealing with structured data from XML documents, SQL databases, and CSV files. They combine the type system of F# with structured data, which can be a tedious task in statically typed languages in many cases. Using type providers, the type of the data source is automatically converted to native types; this means the data is parsed and stored using the same field names as used in the data source. This enables Visual Studio and IntelliSense to support you in your coding without looking in the data source for the field names all the time.

Using LINQ and F#

LINQ is a feature in the .NET framework and has been supported in F# since Version 3.0. It's used to provide powerful query syntax and can be used together with databases, XML documents, .NET collections, and so on. In this section we'll briefly introduce LINQ and see how we can use it together with a SQL database. But first we'll look at LINQ together with collections:

1. First we need a list of values:

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

2. Then we can construct our first LINQ query, which will select all the values in the list:

```
let query1 = query { for value in values do select value }  
query1 |> Seq.iter (fun value -> printfn "%d" value)
```

3. This is not that useful, so let's add a condition. We'll now select all values greater than 5, this is done using a where clause:

```
let query2 = query { for value in values do  
                    where (value > 5)  
                    select value }  
query2 |> Seq.iter (fun value -> printfn "%d" value)
```

4. We can add another condition, selecting values greater than 5 and less than 8:

```
let query3 = query { for value in values do
                    where (value > 5 && value < 8)
                    select value }
query3 |> Seq.iter (fun value -> printfn "%d" value)
```

This is quite powerful, and it's more useful when dealing with data from, for example, SQL databases. In the next section we'll look at a sample application using both type providers and LINQ to insert and query a database.

Explaining sample code using type providers and LINQ

In this section we'll look at a sample application using type providers and LINQ. We'll use the `Log` table that we created earlier. This is a good way to test if everything works with the SQL database and the permissions to read and write to and from it:

```
open System
open System.Data
open System.Data.Linq
open Microsoft.FSharp.Data.TypeProviders
open Microsoft.FSharp.Linq

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

The connection string has to be changed in some cases; if so, simply check the Connection string value in the properties for the database:

```
/// Copied from properties of database
type EntityConnection = SqlEntityConnection<ConnectionString="Data
Source=(localdb)\Projects;Initial Catalog=TradingSystem;
Integrated Security=True;
Connect Timeout=30;Encrypt=False;TrustServerCertificate=False",
Pluralize = true>
let context = EntityConnection.GetDataContext ()

/// Format date according to ISO 8601
let iso_8601 (date:Nullable<DateTime>) =
    if date.HasValue then
        date.Value.ToString("s")
    else "2000-00-00T00:00:00"
```

We'll use LINQ to query our table. The code will be modified and used in the final application. Here we have one query to list all entries from the `Log` table:

```
/// Query all LOG entries
let listAll() =
    query { for logentry in context.LOGs do
            select logentry }
    |> Seq.iter (fun logentry -> printfn "%s -- %d -- %s -- %s"
    (iso_8601 logentry.log_datetime) logentry.log_id
    logentry.log_level logentry.log_msg)

/// Insert a LOG entry
let addEntry(logLevel, logMsg) =
    let fullContext = context.DataContext
    let logTimestamp = DateTime.Now
    let newEntry = new EntityConnection.ServiceTypes.LOG(log_level
    = logLevel,
    log_msg = logMsg,
    log_datetime = Nullable(logTimestamp))
    fullContext.AddObject("LOGs", newEntry)
    fullContext.SaveChanges() |> printfn "Saved changes: %d
    object(s) modified."
```

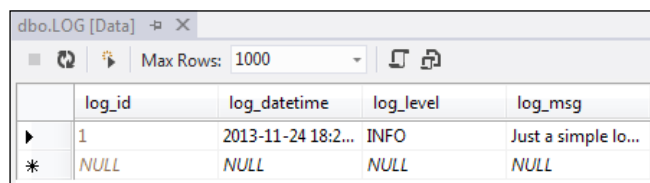
We can use F# Interactive to see if the code works by adding a log entry and then trying to obtain it:

```
> addLogEntry("INFO", "Just a simple log entry");;
Saved changes: 1 object(s) modified.
val it : unit = ()

> listAll();;
2013-09-26T14:08:02 -- 1 -- INFO -- Just a simple log entry
val it : unit = ()
```

This seems to work fine, and we can consider the SQL Server setup done. We'll add tables and functionality in the next two chapters. But this is fine for now, the setup is complete.

It's also convenient to browse the data in Visual Studio for a table in the database. To do this, right-click on the `Log` table in the **SQL Server Object Explorer** and **View Data**. You will see a view similar to the following screenshot:



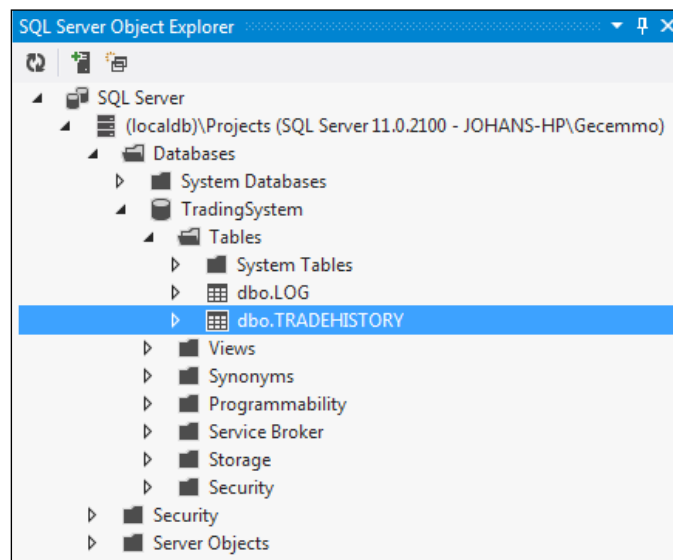
	log_id	log_datetime	log_level	log_msg
▶	1	2013-11-24 18:2...	INFO	Just a simple lo...
*	NULL	NULL	NULL	NULL

Creating the remaining table for our project

The SQL database will be used to store the state and history of our system. It's mainly for illustrative purposes. We'll store log information and trade history. For this, we have to add a new table, and this will be done in the same way as before.

```
CREATE TABLE TRADEHISTORY
(
    tradehistory_id int IDENTITY PRIMARY KEY,
    tradehistory_datetime datetime DEFAULT CURRENT_TIMESTAMP,
    tradehistory_instrument nvarchar(12),
    tradehistory_qty int,
    tradehistory_type nvarchar(12),
    tradehistory_price float
)
```

If you execute the preceding SQL code, you will now see the table added to the **Tables** view in the **SQL Server Object Explorer** as shown in the following screenshot:



Following in our previous footsteps, we will examine sample code to query and insert a trade history entry:

```
/// Query trading history
let listAllTrades() =
    query { for trade in context.TRADEHISTORIES do select trade }
    |> Seq.iter (fun trade -> printfn "%A" (iso_8601
    trade.tradehistory_datetime, trade.tradehistory_id,
```

```
trade.tradehistory_instrument, trade.tradehistory_type,
trade.tradehistory_price, trade.tradehistory_qty))

/// Insert a trade
let addTradeEntry(instrument, price, qty, otype) =
    let fullContext = context.DataContext
    let timestamp = DateTime.Now
    let newEntry = new
        EntityConnection.ServiceTypes.TRADEHISTORY
        (tradehistory_instrument = instrument,
         tradehistory_qty = Nullable(qty),
         tradehistory_type = otype,
         tradehistory_price =
             Nullable(price),
         tradehistory_datetime =
             Nullable(timestamp))
    fullContext.AddObject("TRADEHISTORIES", newEntry)
    fullContext.SaveChanges() |> printfn "Saved changes: %d
object(s) modified."
```

We can use F# Interactive to see if the code works by adding a trade entry and then trying to obtain it:

```
> addTradeEntry("MSFT", 45.50, 100, "limit")
Saved changes: 1 object(s) modified.
val it : unit = ()

> listAllTrades()
("2013-11-24T20:40:57", 1, "MSFT", "limit", 45.5, 100)
val it : unit = ()
```

It seems to work fine!

Writing test cases

In this section we'll look at some of the test cases that can be written for the trading system. We'll use NUnit together with the graphical user interface provided by NUnit to accomplish this. The following screenshot displays the main GUI from NUnit:

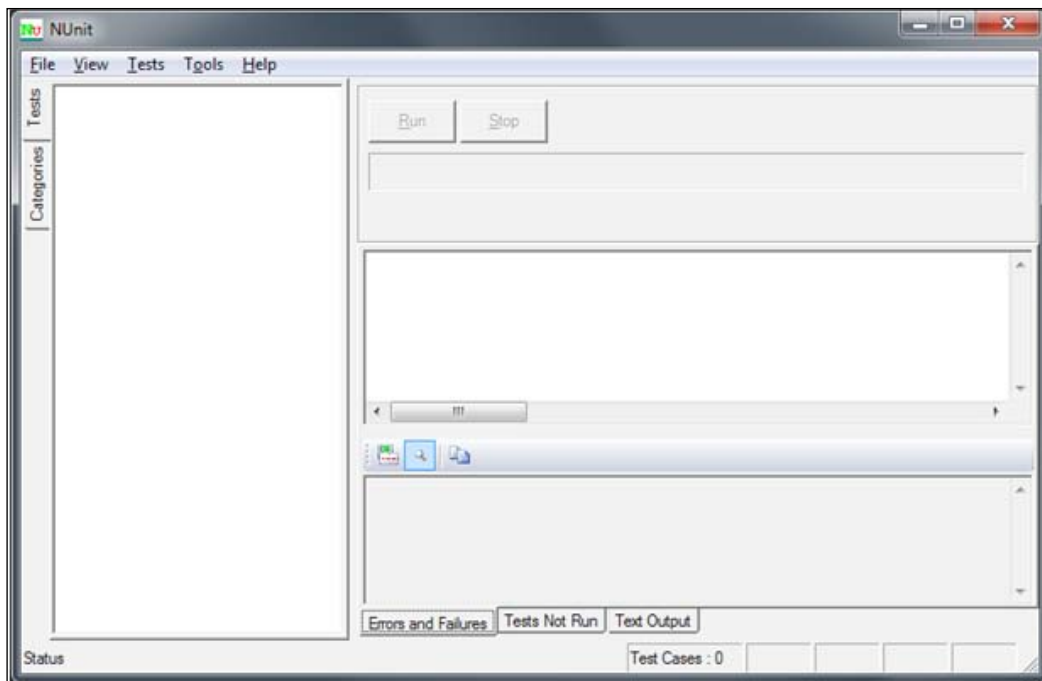


Figure 4: The NUnit user interface

Before we start to write real tests for our system, we'll write a simple test to verify that our setup is correct. NUnit will automatically rerun the executable every time it's built. We start by writing a simple test inside the `TestOrderValidation` file, before we write the real ones:

```
[<Test>]
let OneIsEqualToOne() =
    1 |> should equal 1
```

This is quite silly, but we'll be able to see if NUnit detects changes and if NUnit will detect the test cases inside the `.exe` file. The steps for writing a simple test case are as follows:

1. Open up **NUnit** and navigate to **File | Open Project...**
2. Select the `.exe` file corresponding to the `.exe?` file in **TradingSystem.Tests**, located in `..\visual studio 2012\Projects\TradingSystem\TradingSystem.Tests\bin\Debug`.

3. Press the **Run** button and it should look like the following figure:

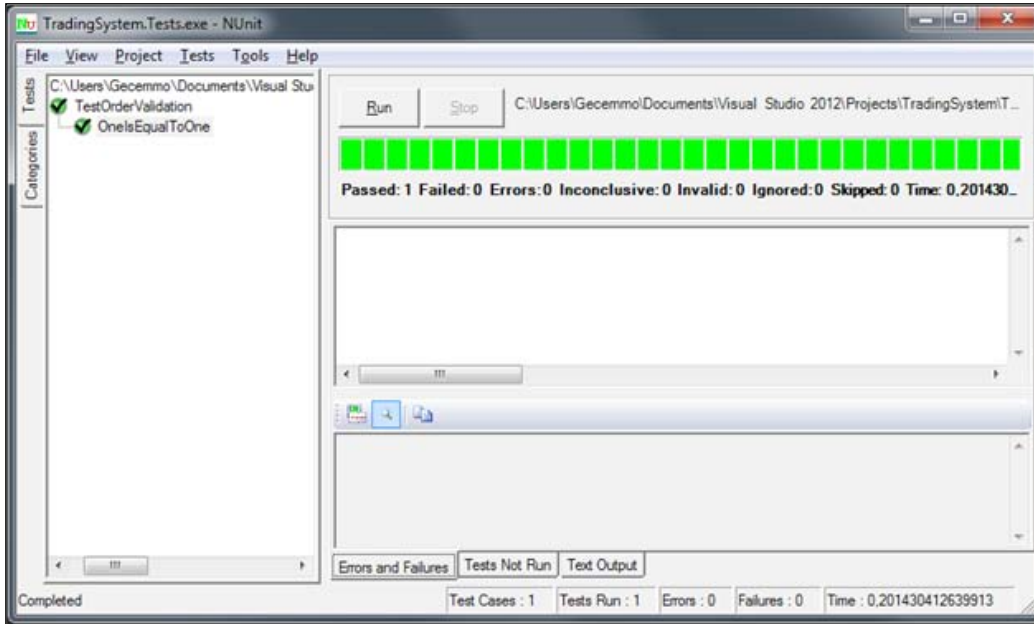


Figure 5: NUnit when the sample unit test is run successfully

Now that we know the setup is correct, let's start to write some real unit tests for our order validation code:

```
namespace TradingSystem.Tests

open System
open NUnit.Framework
open FsUnit

open TradingSystem.Orders

module OrderTests =
    [<Test>]
    let ValidateBuyOrder() =
        let buyOrder = Order(OrderSide.Buy, OrderType.Limit,
            54.50, Tif.FillorKill, 100, "MSFT", 0.0)
        validateOrder (Right buyOrder) |> should equal (Right
            buyOrder)
```

Now we need to build the project and then NUnit should be able to detect the changes. If everything works correctly, the NUnit GUI should look like the following screenshot:

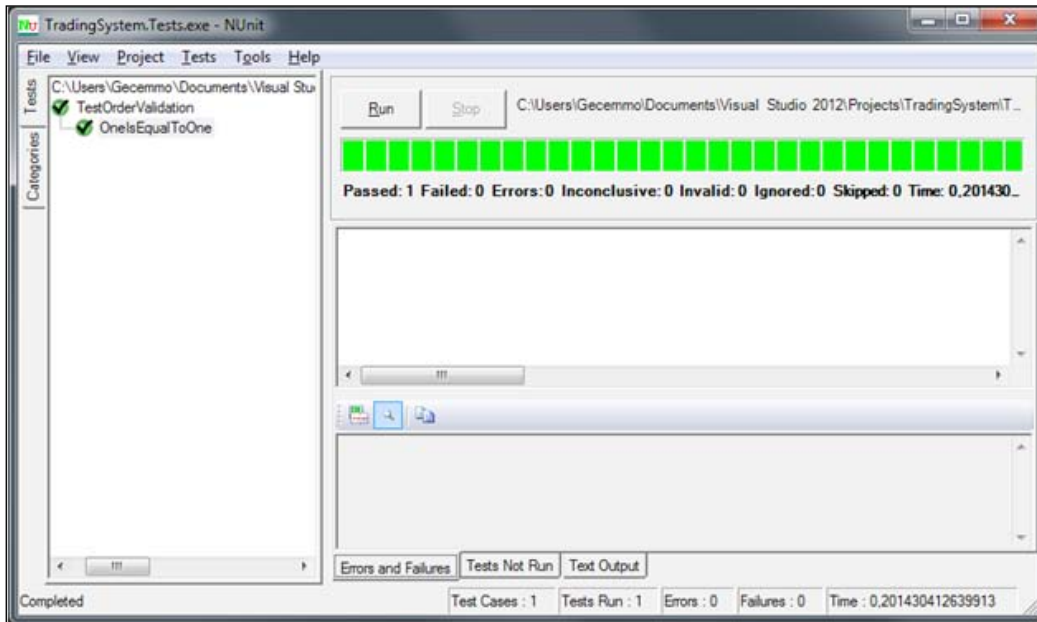


Figure 6: NUnit when the ValidateBuyOrder test is run

Let's add some more tests for the order validation code:

```

module ValidateOrderTests =
    [<Test>]
    let ValidateBuyOrder() =
        let buyOrder = Order(OrderSide.Buy, OrderType.Limit,
            54.50, Tif.FillorKill, 100, "MSFT", 0.0)
        validateOrder (Right buyOrder) |> should equal (Right
            buyOrder)

    [<Test>]
    let ValidateOrderNoPrice() =
        let buyOrderNoPrice = Order(OrderSide.Buy,
            OrderType.Limit, 0.0, Tif.FillorKill, 100, "MSFT", 0.0)
        validateOrder (Right buyOrderNoPrice) |> should equal
            (Left "Limit orders must have a price > 0")

    [<Test>]
    let ValidateStopLimitNoPrice() =
        let stopLimitNoPrice = Order(OrderSide.Buy,

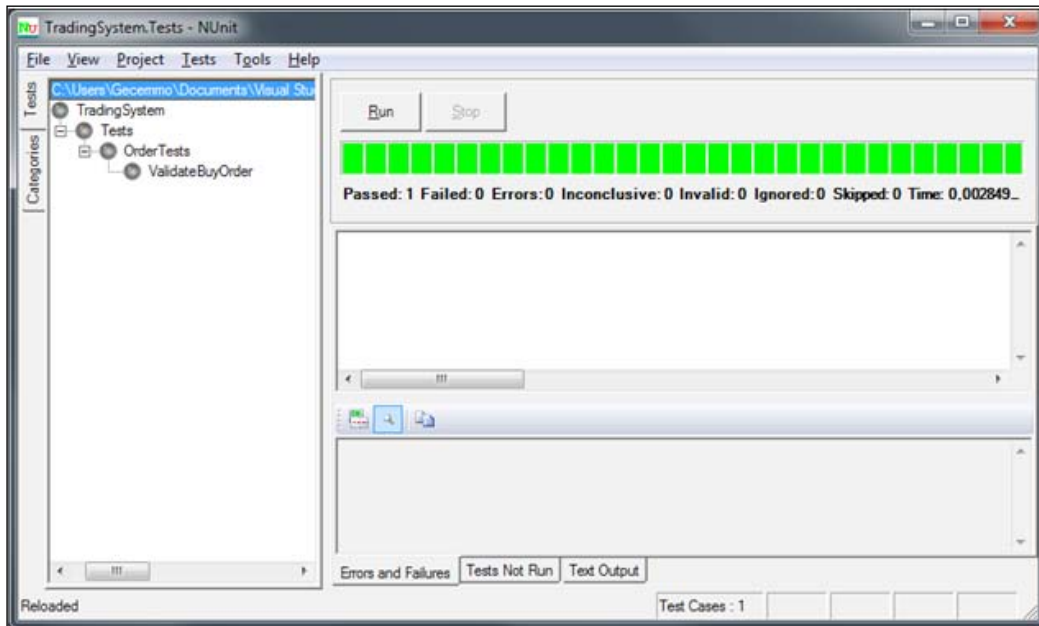
```

Setting Up the Trading System Project

```
OrderType.StopLimit, 0.0, Tif.FillorKill, 100, "MSFT",
45.50)
validateOrder (Right stopLimitNoPrice) |> should equal
(Left "Stop limit orders must both price > 0 and stop
price > 0")

[<Test>]
let ValidateStopNoPrice() =
    let stopNoPrice = Order(OrderSide.Buy, OrderType.Stop,
0.0, Tif.FillorKill, 100, "MSFT", 45.50)
    validateOrder (Right stopNoPrice) |> should equal (Right
stopNoPrice)
```

Note that the name of the module is renamed to `ValidateOrderTests`. We'll add more tests for validating instruments and using pre-trade risks in the same file. The following screenshot shows the four tests for validating orders:



NUnit when the four tests are run

Details about the setup

The following is a listing of the references used in the project until now. Use this to check if your project is up-to-date:

`TradingSystemFSharp.Core`

- `FSharp.Data.TypeProviders`
- `mscorlib`
- `System`
- `System.Core`
- `System.Data`
- `System.Data.Linq`
- `System.Numerics`

`TradingSystem.Tests`

- `Fsharp.Core`
- `FsUnit.NUnit`
- `TradingSystem.Core`
- `FSharp.Data.TypeProviders`
- `mscorlib`
- `nunit.framework`
- `System`
- `System.Core`
- `System.Numerics`

Summary

In this chapter we set up the project, consisting of the trading system itself and the tests. We looked at how to use Visual Studio to handle our solution and integrate it with NUnit and FsUnit. We also looked at how to connect to Microsoft SQL Server and how to write queries in LINQ and retrieve data throughout the type providers.

In the next two chapters, we'll continue to develop the trading system from the base built in this chapter.

9

Trading Volatility for Profit

In this chapter, we'll look at various trading strategies for volatility. We'll cover directional volatility trading and relative value volatility arbitrage. Options and payoff diagrams, where we use F# and FSharpChart to visualize them, are also briefly covered. In this chapter, you will learn:

- Trading volatility
- Volatility arbitrage opportunities
- Obtaining and calculating the data needed for the strategy
- Deriving the mathematics behind volatility arbitrage

Trading the volatility

Trading volatility is like trading most other assets, except that volatility can't be traded explicitly. Volatility is traded implicitly using, for example, options, futures, and the VIX index. Because volatility is an intrinsic value of the assets, it can't be traded directly. To be able to trade volatility, either a hedge position using a derivative and its underlying asset or an option position is initiated.

One often divides volatility trading into two categories: directional trading and relative value. Directional trading in volatility means we trade in the direction of the volatility. If the volatility is high, we may initiate a short trade in volatility. Relative value means we initiate two trades, where, for example, we go long in a call and short in another call. The first call may be under-valued in terms of volatility and the other may be slightly over-priced. The two related assets are then supposed to mean revert, and the profit is to be monetized.

In this chapter, we'll briefly cover volatility trading and the ways of earning profit from this activity.

Plotting payoff diagrams with FSharpCharts

In this section, we'll construct basic payoff diagrams for European call and put options. Payoff diagrams are useful to visualize the theoretical payoff given the price of the stock and the strike of the option.

The payoff function for a **call** option is defined as follows:

$$\max \{S_t - K, 0\}$$

And the payoff function for a **put** option is defined as follows:

$$\max \{K - S_t, 0\}$$

Let's look at how to do this in F#. We start by defining the payoff functions for calls and puts:

```
/// Payoff for European call option
// s: stock price
// k: strike price of option
let payoffCall k s = max (s-k) 0.0

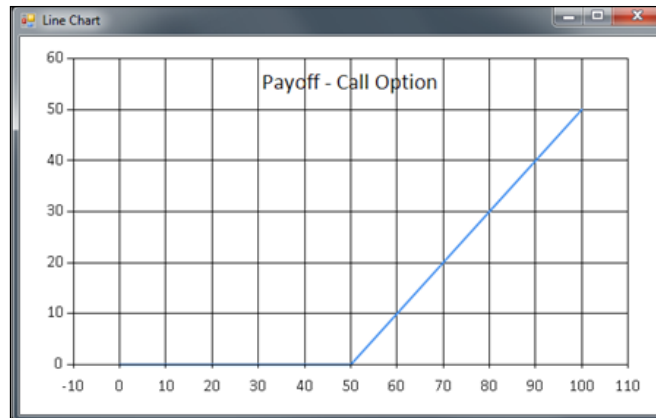
/// Payoff for European Put option
// s: stock price
// k: strike price of option
let payoffPut k s = max (k-s) 0.0
```

We can use these functions to produce numbers to be fed into FSharpChart and visualize the data:

```
/// Calculate the payoffs given payoff function
let payoff payoffFunction = [ for s in 0.0 .. 10.0 .. 100.0 -> s,
    payoffFunction s ]
```

We start by generating the payoff diagram for a call option with a strike value of 50.0:

```
let callPayoff = payoff (payoffCall 50.0)
Chart.Line(callPayoff).WithTitle("Payoff - Call Option")
```

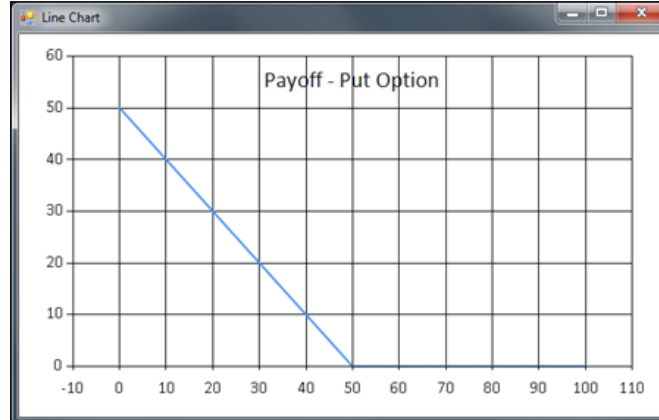


The payoff diagram showing the call option

In the preceding chart, we can see the payoff of the call option being **0** until the stock price reaches **50** (the strike of the option). From there, the payoff goes up. We only plot values from **0** to **100**, so the maximum payoff in this diagram is **50**.

The procedure is the same for a put option with a strike value of 50.0:

```
let putPayoff = payoff (payoffPut 50.0)
Chart.Line(putPayoff).WithTitle("Payoff - Put Option")
```



The payoff diagram showing the put option

The payoff diagram for the put option is the opposite of what we had seen earlier. In the preceding chart, the payoff will decline until it is zero at the strike of the option. This means the put option will be profitable for stock prices below the strike of the option.

Finally, we create a combined chart:

```
/// Combined chart of payoff for call and put option  
let chart = Chart.Combine [Chart.Line(callPayoff);  
    Chart.Line(putPayoff).WithTitle("Payoff diagram")]
```

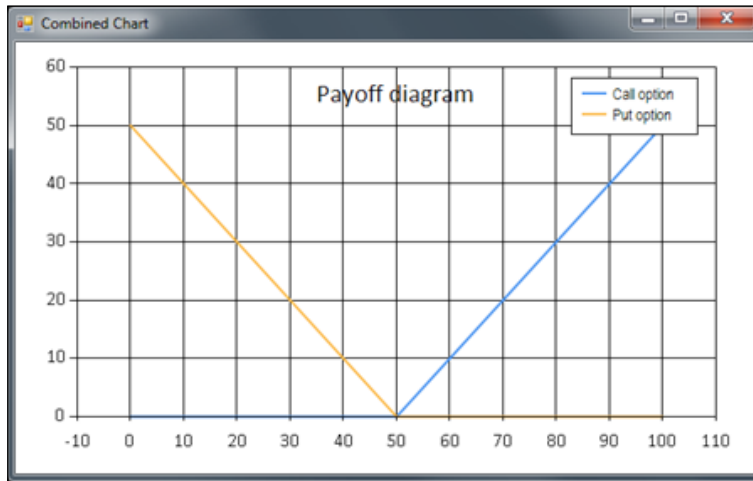


Figure 3: The combined payoff diagram

Learning directional trading strategies

Directional trading in volatility means trading in the direction of the volatility. If the volatility is high, we may initiate a short trade in volatility. In this section, we'll first look at how to trade volatility using option strategies. Then, using an option and the underlying price to trade volatility, we'll look at the VIX index and the delta neutral position.

Trading volatility using options

One way of trading volatility is to use options. We'll look at two option strategies for trading volatility or price movement in the underlying option.

Trading the straddle

The straddle position consists of two options: one put and one call. Straddles are useful when the viewpoint of the underlying market is neutral, which means there is no speculation in the long-term movement of the market. It also means the straddle position is useful when one wants to trade volatility, regardless of the movement of the market.

Long straddle

A long straddle trade is created by taking a long position on both a call and a put option with the same strike and expiration. The long straddle is useful if you think the volatility is low and you want to monetize the potential increase in volatility.

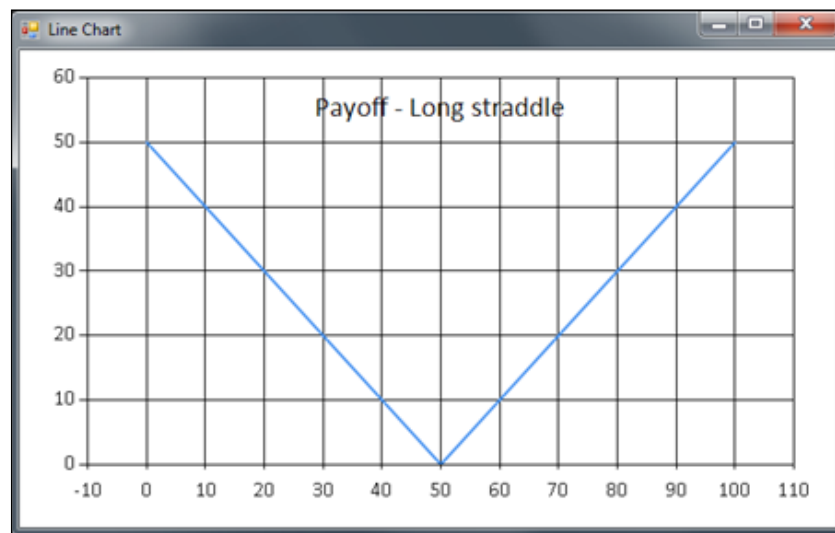
The idea is that the two options (call and put) will cancel out the exposure to the underlying market, except for the volatility in the underlying asset of the options. This means the straddle is very sensitive to changes in volatility. In more technical terms, their respective deltas will be close to 0.5 and -0.5, which means they cancel out. This is because the delta for the money options is around 0.5 for calls and -0.5 for puts.

Let's look at some code to implement the payoff function for the long straddle:

```
/// Payoff for long straddle
// s: stock price
// k: strike price of option
let longStraddle k s =
    (payoffCall k s) +
    (payoffPut k s)
```

```
Chart.Line(payoff (longStraddle 50.0)).WithTitle("Payoff - Long
straddle")
```

The following screenshot depicts the line chart for the payoff function for the long straddle:



The combined payoff diagram

Short straddle

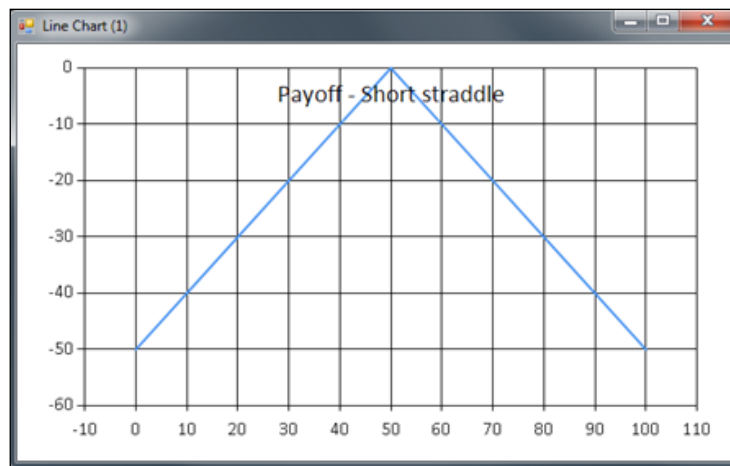
A short straddle is the opposite of a long straddle. Here, we create the position by going short on one call and one put option that have the same strike and expiration. A short straddle is used for trading a decrease in the volatility of an asset, without exposure to the market in other aspects.

The code for the payoff for our short straddle is obtained by adding two short positions (note the minus preceding (payoffCall k s)):

```
/// Payoff for Short straddle
// s: stock price
// k: strike price of option
let shortStraddle k s =
  -(payoffCall k s) +
  -(payoffPut k s)
```

```
Chart.Line(payoff (shortStraddle 50.0)).WithTitle("Payoff - Short
straddle")
```

The following screenshot depicts the line chart for the payoff function for the short straddle:



The combined payoff diagram

Trading the butterfly spread

The butterfly spread consists of three legs and comes in two flavors: the long butterfly spread and the short butterfly spread.

The long butterfly spread

The long butterfly position is created by selling two at-the-money call options and buying two calls: an in-the-money call option and an out-of-the-money call option. The two calls will serve as insurances for the short position.

In summary, you need to:

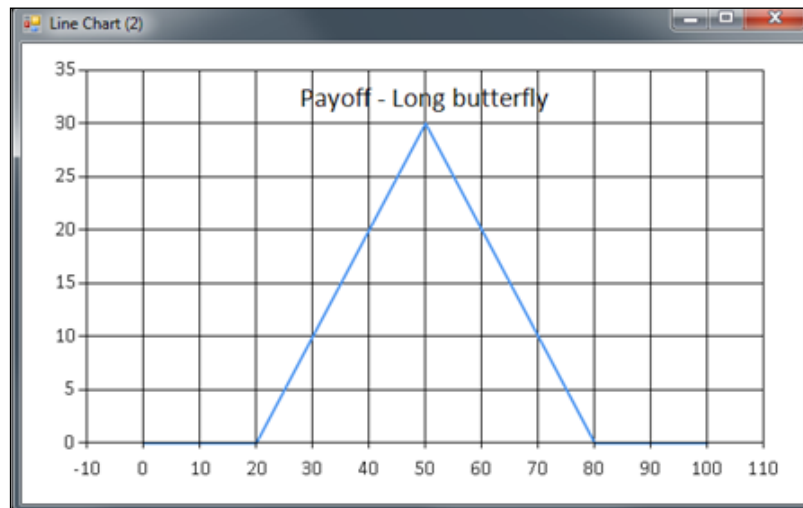
- Short sell two at-the-money call options
- Buy an in-the-money call
- Buy an out-of-the-money call

We can represent the preceding rules as code in F# to generate payoffs:

```
/// Payoff for long butterfly
// s: stock price
// h: high price
// l: low price
let longButterfly l h s =
    (payoffCall l s) +
    (payoffCall h s) -
    2.0 * (payoffCall ((l + h) / 2.0) s)
```

```
Chart.Line(payoff (longButterfly 20.0 80.0)).WithTitle("Payoff - Long butterfly")
```

This code will generate a diagram showing the payoff for the long butterfly:



The payoff diagram for the long butterfly spread

The short butterfly spread

The short butterfly position is created by buying two at-the-money call options and selling an in-the-money and an out-of-the-money call. The two calls will serve as insurances for the short position.

In summary, you have to:

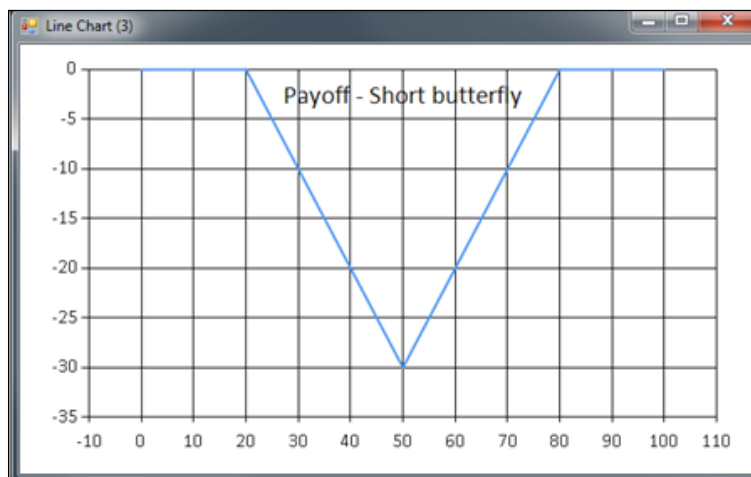
- Buy two at-the-money (or in the middle of the two other's strike prices) call options
- Sell an in-the-money call
- Sell an out-of-the-money call

We can represent the preceding rules as code in F# to generate payoffs:

```
/// Payoff for short butterfly
// s: stock price
// h: high price
// l: low price
let shortButterfly l h s =
    -(payoffCall l s) +
    -(payoffCall h s) -
    2.0 * -(payoffCall ((l + h) / 2.0) s)
```

```
Chart.Line(payoff (shortButterfly 20.0 80.0)).WithTitle("Payoff - Short butterfly")
```

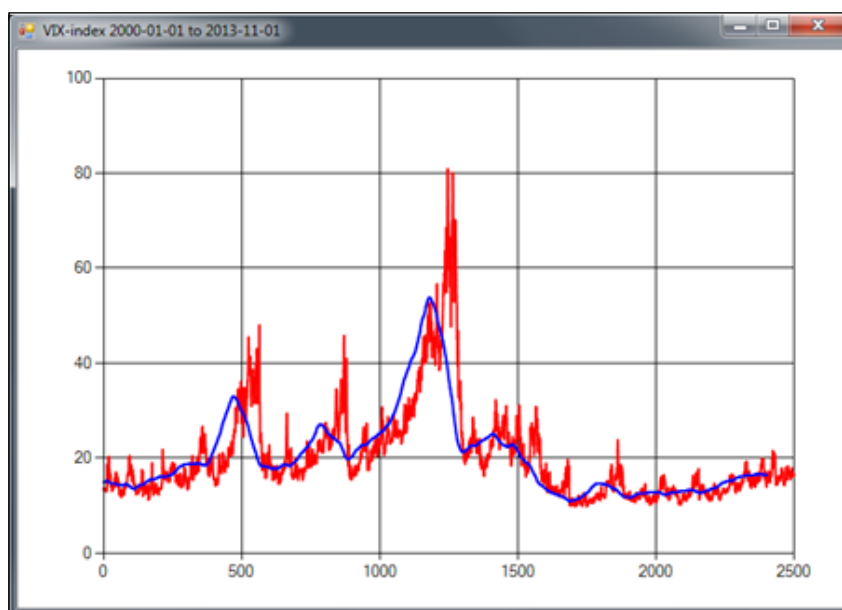
This code will generate the following diagram showing the payoff for the short butterfly:



The payoff diagram for the short butterfly spread

Trading the VIX

Another alternative if you are interested in trading volatility using a directional trading strategy is to trade the VIX index. VIX is an index that combines the implied volatility of the S&P 500 index options. This can be interpreted as an indication of future volatility for the next 30 days to come. The prediction power of the VIX is in parity with historical returns of the S&P 500 index itself. This means that the information provided from the VIX is not a silver bullet when it comes to volatility forecasting; it is better used for directional trading in volatility. The following is a screenshot of a plot over the VIX index and a moving average indicator:



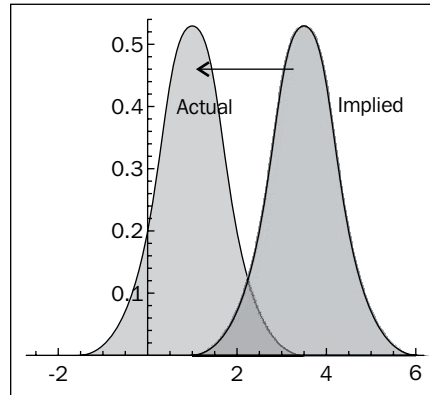
The VIX index from 2000-01-01 to 2013-11-01

Trading the delta neutral portfolio

A delta neutral portfolio is constructed by an option and the underlying instrument. The portfolio will, in theory, be resistant to small changes in the underlying price (delta neutral). On the other hand, other things tend to change the value of the portfolio; this means we have to rebalance it on a regular basis.

In this section, we'll mainly extend our analysis of the delta neutral portfolio and study how it can be used to trade volatility.

The following diagram shows the actual volatility and the implied volatility for an asset:



The actual and implied volatilities for an asset

Which volatility should we use in the delta hedge position? We have two choices, either the **actual** volatility or the **implied** volatility.

This turns out to be a rather tricky question to answer without studying the profit and loss on a **mark to market (MTM)** basis for the two volatilities. To put it simply, we can use the actual volatility and then take a random walk until the locked-in profit is realized, that is, the two probability distributions intersect. The other choice is to use the implied volatility. Using the implied volatility will result in a more sane profit and loss curve, which is often preferred from the perspective of risk management because the stochastic term is somewhat reduced, and the profit will come in small increments over time until it's fully realized.

Deriving the mathematics

In this section, we'll look at the mathematics needed to trade a delta neutral portfolio.

The following table presents values of a market neutral portfolio:

Component	Value
Option	V^i
Stock	$-\Delta^i S$
Cash	$-V^i + \Delta^i S$

The following table shows the values of the market neutral portfolio for the next day:

Component	Value
Option	$V^i + dV^i$
Stock	$-\Delta^i S - \Delta^i dS$
Cash	$(-V^i + \Delta^i S)(1 + rdt) - \Delta^i DSdt$

Hedging with implied volatility

In this section, we are going to derive the mathematical tools for hedging with implied volatility to be able to watch the mark to market profit and loss.

The following is the mark to market profit from the current day to the next day:

$$dV^i - \Delta^i dS - r(V^i - \Delta^i S)dt - \Delta^i DSdt = \frac{1}{2}(\sigma^2 - \tilde{\sigma}^2)S^2 \Gamma^i dt$$

Here, S is the stock price and Γ is the Black-Scholes gamma function.

The following is the theoretical profit until the end of the arbitrage trade:

$$\frac{1}{2}(\sigma^2 - \tilde{\sigma}^2) \int_{t_0}^T e^{-r(t-t_0)} S^2 \Gamma^i dt$$

We integrate the discounted value of each profit made until the end of the trade to get the total theoretical profit.

Implementing the mathematics

Using Math.NET, let's implement the mathematics derived in the previous section to get a feel of the close connection between the formulas and F#:

```

/// Mark-to-market profit

/// Normal distribution
open MathNet.Numerics.Distributions;

```



```
let normd = new Normal(0.0, 1.0)

/// Black-Scholes Gamma
// s: stock price
// x: strike price of option
// t: time to expiration in years
// r: risk free interest rate
// v: volatility
let black_scholes_gamma s x t r v =
let d1=(log(s / x) + (r+v*v*0.5)*t)/(v*sqrt(t))
normd.Density(d1) / (s*v*sqrt(t))

let mark_to_market_profit s,x,t,r,v,vh = 0.5*(v*v -
vh*vh)*S*S*gamma(s,x,t,r,v)
```

Learning relative value trading strategies

Relative value volatility trading refers to trading volatility using opposite legs of some financial instruments, such as options, to take advantage of the movement in volatility. Usually, one would initiate a trade with a long call and a short call, forming a two-legged trade. There are a lot of variations to these types of trades, and we will mainly look at trading the slope of the volatility smile using options. This will form the basis of the trading strategy used in this book.

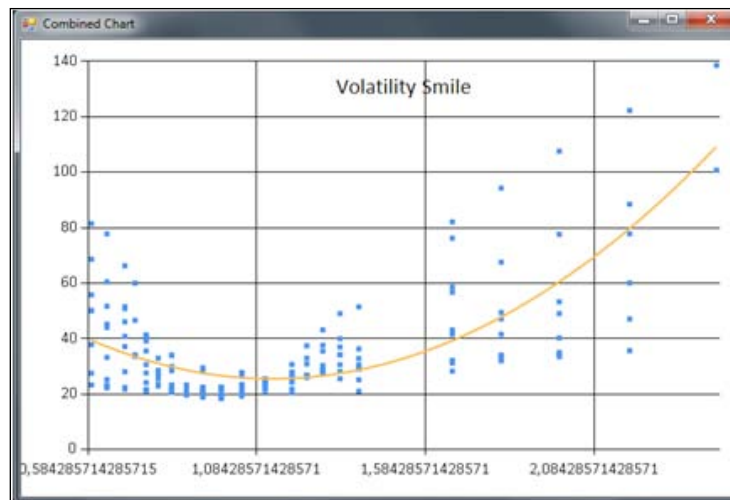
Trading the slope of the smile

First, we'll recap *Chapter 6, Exploring Volatility*, where we looked at the smile effect for options on the OMX exchange in Sweden. The volatility smile is a phenomenon observed in stock markets. The smile is obtained by plotting the implied volatility from the options on the y axis and moneyness on the x axis.

Moneyness is the ratio between the spot price of the underlying asset, S , and the strike price of the option, K :

$$M = \frac{S}{K}$$

In the following screenshot, you will see the moneyness, M , on the x axis and the implied volatility on the y axis:



The volatility smile

One shortcoming of the preceding chart is that we plot multiple expiration dates in the same chart. We need to refine this approach to be able to study the volatility smile in more detail. As the first step, let's modify the code for this:

```
open MathNet.Numerics
open MathNet.Numerics.LinearAlgebra
open MathNet.Numerics.LinearAlgebra.Double
open MathNet.Numerics.Distributions

let filePath = @"C:\Users\Gecemmo\Desktop\smile_data.csv"

/// Split row on commas
let splitCommas (l:string) =
    l.Split(',')

/// Read a file into a string array
let openFile (name : string) =
    try
        let content = File.ReadAllLines(name)
        content |> Array.toList
    with
        | :? System.IO.FileNotFoundException as e -> printfn
            "Exception! %s " e.Message; ["empty"]
```

The first modification is made in the `readVolatilityData` function where a date parameter is added. This is used to filter out rows that match the date from the CSV file:

```
/// Read the data from a CSV file and returns
/// a tuple of strike price and implied volatility%
// Filter for just one expiration date
let readVolatilityData date =
    openFile filePath
    |> List.map splitCommas
    |> List.filter (fun cols -> cols.[1] = date)
    |> List.map (fun cols -> (cols.[2], cols.[3]))
```

The following code is the same as we used before, but in the next step, we need to make a minor change:

```
/// 83.2
/// Calculates moneyness and parses strings into numbers
let calcMoneyness spot list =
    list
    |> List.map (fun (strike, imp) -> (spot / (float strike), (float
imp)))

// Filter out one expiration date -- 2013-12-20
let list = readVolatilityData "2013-12-20"
let mlist = calcMoneyness 83.2 list

/// Plot values using FSharpChart
fsi.AddPrinter(fun (ch:FSharp.Charting.ChartTypes.GenericChart) ->
ch.ShowChart(); "FSharpChartingSmile")
Chart.Point(mlist)
```

The following is the code to plot the data points together with the regression line obtained. As we will see in the chart, the regression is not satisfactory:

```
let xdata = mlist |> Seq.map (fun (x, _) -> x) |> Seq.toList
let ydata = mlist |> Seq.map (fun (_, y) -> y) |> Seq.toList

let N = xdata.Length
let order = 2

/// Generating a Vandermonde row given input v
let vandermondeRow v = [for x in [0..order] do yield v ** (float
x)]

/// Creating Vandermonde rows for each element in the list
let vandermonde = xdata |> Seq.map vandermondeRow |> Seq.toList

/// Create the A Matrix
let A = vandermonde |> DenseMatrix.ofRowsList N (order + 1)
```

```
A.Transpose()
```

```
/// Create the Y Matrix
let createYVector order l = [for x in [0..order] do yield l]
let Y = (createYVector order ydata |> DenseMatrix.ofRowsList
  (order + 1) N).Transpose()
```

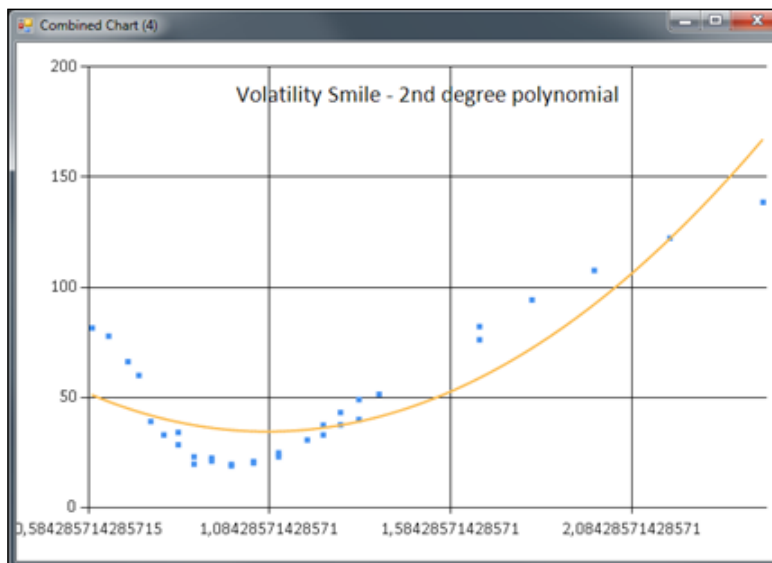
The final step is to calculate the regression coefficients and to use these to calculate the points for our curve. Then, we will use a combined plot with the points and the fitted curve as we did in *Chapter 6, Exploring Volatility*:

```
/// Calculate coefficients using least squares
let coeffs = (A.Transpose() * A).LU().Solve(A.Transpose() *
  Y).Column(0)

let calculate x = (vandermondeRow(x) |> DenseVector.ofList) *
  coeffs

let fitxs = [(Seq.min xdata).. 0.01 ..(Seq.max xdata)]
let fitys = fitxs |> List.map calculate
let fits = [for x in [(Seq.min xdata).. 0.01 ..(Seq.max xdata)] do
  yield (x, calculate x)]

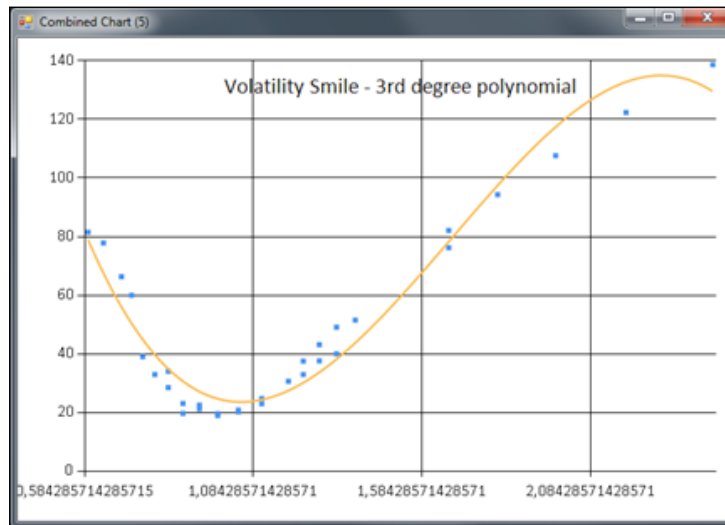
let chart = Chart.Combine [Chart.Point(mlist); Chart.Line(fits)].
  WithTitle("Volatility Smile - 2nd degree
  polynomial")]
```



The volatility smile for the expiration on January 20, 2013

Instead, we can try to fit a third degree polynomial and evaluate the graph. We just change the order value to 2:

```
let order = 2
```



The volatility smile using a third degree polynomial with the same expiration date

The result is more compelling this time. As we can see, there is some inconsistency between the options. This isn't necessarily the same as there exist some arbitrage opportunities in this case.

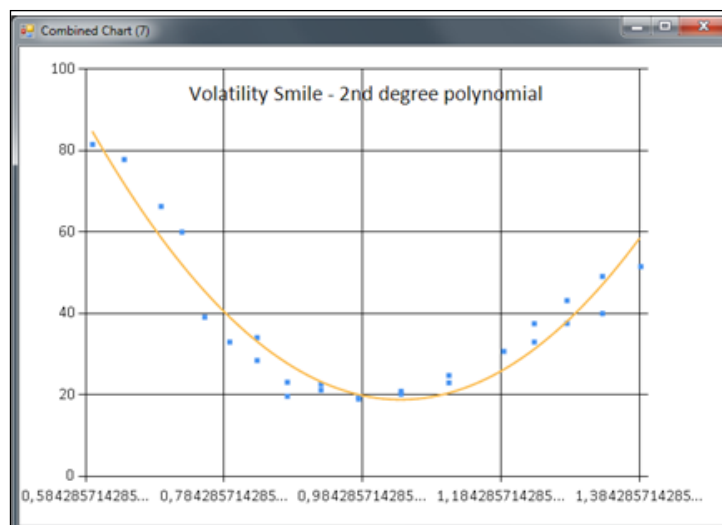
How can we take advantage of the inconsistency in the volatility for options with the same maturity date? One way could be to study the movements of the implied volatility as time proceeds. What if the volatilities are mean reverting the regression in some way?

First, we'll zoom in on the problem and on the set of options. The x axis is the moneyness of the options. We'll look at the moneyness between 0.50 and 1.5 in the first experiment.

We write some F# code to help us with this:

```
// Filter on moneyness, 0.5 to 1.5
let mlist = calcMoneyness 83.2 list |> List.filter (fun (x, y) ->
    x > 0.5 && x < 1.5)
```

This is just a modification to the assignment to `mlist`, filtering on the `x` value. The filter and a change to the second degree polynomial will render the following chart:



The volatility smile using a second degree polynomial for moneyness between 0.5 and 1.5

We'll assume that the slope will change and that the slope is somewhat mean reverting, meaning we can initiate a position with one long and one short position in options affected by the movement.

Defining the trading strategy

The trading strategy for our system will be based on relative value volatility arbitrage as described earlier. This will enable us to trade exclusively with options, to be more precise, in-the-money call options.

First, we define the slope between the two "edges" of the moneyness: the upper and lower bounds of the moneyness. We have to look at a graph for doing this. For the preceding graph, that would typically be [0.5, 1.0].

To get a more mathematical expression for the slope, we look at two points and calculate the slope from these:

$$\beta(t) = \frac{\sigma_1(t) - \sigma_2(t)}{m_1 - m_2}$$

Here, m is the moneyness and σ (sigma) is the implied volatility from the option prices. The slope can either rise or fall, which means β will increase, decrease, or of course, neither will happen. Let's look at the two cases more closely.

Case 1 – increasing the slope

In the case of a slope that is lower than the regression (average), we can assume that the slope will eventually revert. In the case of a rising slope, the slope is as follows:

$$\beta(t_1) - \beta(t_0) > 0$$

This leads to the following inequality, where the combined volatility is lower at time 0 than in the upcoming point in time:

$$\sigma_1(t_0) - \sigma_2(t_0) < \sigma_1(t_1) - \sigma_2(t_1)$$

We can trade this increasing slope by creating a trade with one long call and one short call. The difference between the rise in volatility will result in a potential profit. This means we need to consider the Vega of the two options. In case the Vega is higher for the option corresponding to σ_2 than it is for σ_1 , the position may lose money.

Case 2 – decreasing the slope

As in the case of an increasing slope, the same holds for the case of a decreasing slope. We can assume that the slope is reverting at some later point in time. This means the slope at time one (t1) minus the slope at point zero (t0) is less than zero:

$$\beta(t_1) - \beta(t_0) < 0$$

This leads to the following inequality, where the combined volatility is greater at time zero than at the upcoming point in time:

$$\sigma_1(t_0) - \sigma_2(t_0) > \sigma_1(t_1) - \sigma_2(t_1)$$

The trade is initiated with a short call and a long call.

Defining the entry rules

The entry rules for the system will be:

- Every time there is a situation where the slope for β is lower than the slope of the regression, we initiate a trade as per case 1
- Every time there is a situation where the slope for β is greater than the slope of the regression, we initiate a trade as per case 2

Defining the exit rules

The trade will be closed as soon as the inequality in either case 1 or 2 no longer holds. This means the slope has reverted and we may lose money. We'll also add a time constraint, which tells us to limit the duration of the trade to two days. This can be adjusted of course, but this kind of behavior is typically for intraday behavior.

We will implement the rules defined here in the next chapter, where we put the pieces together to build a fully working trading system by using options.

Summary

In this chapter, we have looked into the details about the theory needed for our trading strategy. We derived some of the mathematical tools used in volatility trading and discussed how these can be used in the trading system. Some of the concepts have been introduced in earlier chapters and only slightly modified versions were introduced here. In the next chapter, we'll put the pieces together and look at how to present the data from the trading system in a GUI.

10

Putting the Pieces Together

This chapter covers the final step of building an automated trading system. We will look at how to refactor the system and change it to reflect new requirements.

In this chapter you will learn about:

- Executing a trading strategy
- Presenting information in the GUI
- Possible additions to the trading system

Understanding the requirements

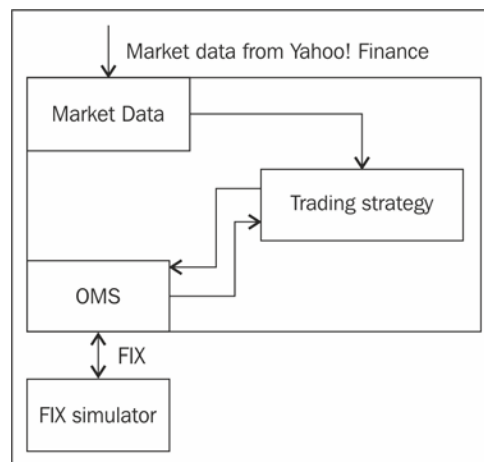
We've covered some of the requirements in *Chapter 8, Setting Up the Trading System Project*, but let's review them again and see how the system will be defined. The main thing about the automated trading system is that it needs to be able to process market data and make decisions based on the data. The decisions will then be converted to FIX 4.2 messages and sent to a FIX simulator, a real broker, or the stock exchange. In this rather simple setup, the market data will be the daily data from Yahoo! Finance that will be downloaded and parsed daily.

The automated trading system should be able to:

- Store log entries in a Microsoft SQL Server database
- Store trading history in a Microsoft SQL Server database
- Download quotes from Yahoo! Finance
- Manage orders with an **Order Management System (OMS)**
- Send orders using FIX 4.2
- Connect the trading system to a FIX simulator via FIX 4.2

- Execute a trading strategy written in F#
- Control itself using a basic GUI with start/stop buttons
- Display the current position(s)
- Display the current **profit and loss (P&L)**
- Display the latest quote(s)
- Use the MVC pattern and the `INotifyPropertyChanged` interface

The following is a diagram illustrating the data flow in the trading system:



Data flow in a trading system

Revisiting the structure of the system

We'll revisit the project structure and make sure all the dependencies are added. The following are the parts of an automated trading system:

- Feed handlers and market data adapters
- Trading strategies
- Order execution and order management
- Persistence layer (DBs)
- GUI for monitoring the system

We need two new dependencies. They are as follows:

- `System.Windows.Forms`
- `System.Drawing`

We need the `System.Windows.Forms` dependency to create our GUI. It provides support for Windows itself and the controls that are used. The `System.Drawing` dependency is also needed to provide the basic graphics functionality. The following is a list of the references needed in the project. You can verify your own project against the list to make sure you have all the dependencies needed.

The trading system is split into two projects: `TradingSystem` and `TradingSystem.Tests`.

The following is the list of dependencies required for `TradingSystem`:

- `FSharp.Core`
- `FSharp.Data.TypeProviders`
- `microsoft`
- `NQuantLib`
- `System`
- `System.Core`
- `System.Data`
- `System.Data.Linq`
- `System.Drawing`
- `System.Numerics`
- `System.Windows.Forms`

The following is the list of dependencies required for `TradingSystem.Tests`:

- `FSharp.Core`
- `FsUnit.NUnit`
- `TradingSystem.Core`
- `FSharp.Data.TypeProviders`
- `microsoft`
- `nunit.framework`
- `System`
- `System.Core`
- `System.Numerics`

Understanding the Model-View-Controller pattern

In this section, we'll look at the concept of the MVC design pattern. The MVC pattern is a concept that was introduced at Xerox PARC and has been around since the early days of Smalltalk. It is a high-level design pattern often used in GUI programming. We'll use it later in more detail, but a gentle introduction here will make the concept familiar to you when needed later.

The main idea behind MVC is to separate the model from the view. The view is simply the GUI, which interacts with the user of the program. The GUI will take care of the buttons clicked on and the data displayed on the screen. The model is the data to be used in the program. It can be, for example, financial data. It's often desirable to separate the code for the model (data) and the view (GUI).

The MVC pattern described in the preceding figure is a modified version of the traditional MVC pattern. The main difference is that there is no direct communication between the view and the model in this variant. This is a more refined way of using the MVC pattern, where the view doesn't have to know anything about the model.

The model

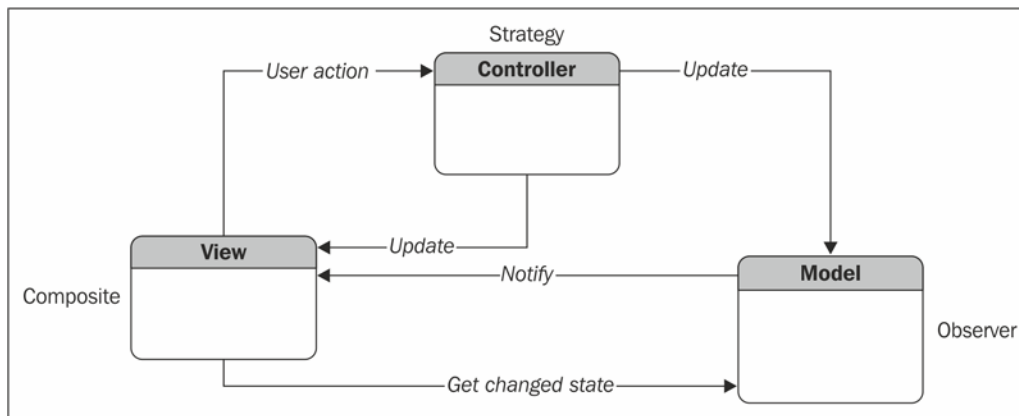
The model is typically the data and the state of the application. In this case, the model will consist of orders, the market data, and the state of the system.

The view

The view is the `TradingSystemForm` class and will be the only GUI form used apart from dialog boxes, which are standard Windows Forms components. The view is the GUI.

The controller

The controller is responsible for connecting the view to the model. The controller is initialized with an instance of the model, and the view is added to the controller during the execution of the program (`Program.fs`). The following diagram is a schematic representation of the relationship between the parts in the MVC pattern:



The MVC pattern, where the controller knows both the model and the view

In our case, the controller will be responsible for updating the view when an action is taken. This means the model will notify the controller, and the controller will update the view. Strictly speaking, this is a slightly modified version of the classic MVC pattern where the model knows about the view and notifies the view instead of the controller.

The main problem with this classic approach is that of tight coupling. Using the controller as the mediator, a compound pattern is formed. This is the same strategy used in popular libraries such as *Cocoa* by Apple.

Executing the trading strategy using a framework

The trading strategy is executed through `onMarketData` when the downloading of data is completed/successful (it sends a message to the agent). If any error occurs, notify the agent. Everything is logged to a SQL backend (SQL Server).

The trading strategy will have six callable functions:

- `onInit`: This function is called when a strategy is initialized
- `onStart`: This function is called when the strategy starts
- `onStop`: This function is called when the strategy stops
- `onMarketData`: This function is called whenever new market data arrives
- `onTradeExecution`: This function is called whenever a trade is executed
- `onError`: This function is called with every error that occurs

The trading strategy will be implemented as a type of its own where callbacks are member functions that are called from the **strategy executor**. The strategy executor consists of an agent receiving messages from the system. The start and stop commands are sent from the two event handlers connected to the buttons in the GUI.

Let's look at the main structure of the framework used to execute the trading strategy:

```
/// Agent.fs
namespace Agents

open System

type TradingStrategy() =
    member this.onInit : unit = printfn "onInit"
    member this.onStart : unit = printfn "onStart"
    member this.onStop : unit = printfn "onStop"
    member this.onTradeIndication : unit = printfn
        "onTradeIndication"

// Type for our agent
type Agent<'T> = MailboxProcessor<'T>
```

We need control messages to be used to communicate with the agent; these are modeled as discriminated unions. The messages are used to change the state and communicate the change between the parts in the system. This is needed because the agent is running in another thread, and the passing of the messages is the way we communicate with them. An example of this is as follows:


```
// Control messages to be sent to agent
type SystemMessage =
    | Init
    | Start
    | Stop

type SystemState =
    | Created
    | Initialized
    | Started
    | Stopped
```

The `TradingAgent` module will receive the control messages and take the appropriate action. The following is the code to implement the functionality to call the corresponding method in the trading strategy using pattern matching:

```
module TradingAgent =
let tradingAgent (strategy:TradingStrategy) = Agent.Start(fun
  inbox ->
  let rec loop state = async {
    let! msg = inbox.Receive()
    match msg with
    | Init ->
    if state = Started then
      printfn "ERROR"
    else
      printfn "Init"
      strategy.onInit
      return! loop Initialized
    | Start ->
    if state = Started then
      printfn "ERROR"
    else
      printfn "Start"
      strategy.onStart
      return! loop Started
    | Stop ->
    if state = Stopped then
      printfn "ERROR"
    else
      printfn "Stop"
      strategy.onStop
      return! loop Stopped
  }
loop Created)
```


The following are parts of the GUI code that are used to control the trading system. Most of the code uses .NET classes mainly from Windows Forms libraries.

[ There are many good resources available at MSDN about Windows Forms at [http://msdn.microsoft.com/en-us/library/ms229601\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms229601(v=vs.110).aspx).]

```
// GUI.fs
namespace GUI

open System
open System.Drawing
open System.Windows.Forms
open Agents

// User interface form
type public TradingSystemForm() as form =
    inherit Form()

    let valueLabel = new Label(Location=new Point(25,15))
    let startButton = new Button(Location=new Point(25,50))
    let stopButton = new Button(Location=new Point(25,80))
    let sendButton = new Button(Location=new Point(25,75))
```

The trading strategy will be initiated and passed to the agent. During the initiation, the parameters and other constant values will be passed:

```
let ts = TradingStrategy()
let agent = TradingAgent.tradingagent ts

let initControls =
    valueLabel.Text <- "Ready"
    startButton.Text <- "Start"
    stopButton.Text <- "Stop"
    sendButton.Text <- "Send value to agent"
do
    initControls

form.Controls.Add(valueLabel)
form.Controls.Add(startButton)
form.Controls.Add(stopButton)

form.Text <- "F# TradingSystem"
```

```

startButton.Click.AddHandler(new System.EventHandler
    (fun sender e -> form.eventStartButton_Click(sender, e)))

stopButton.Click.AddHandler(new System.EventHandler
    (fun sender e -> form.eventStopButton_Click(sender, e)))

// Event handlers
member form.eventStartButton_Click(sender:obj, e:EventArgs) =

```

When the **Start** button is pressed, this event handler is called and two messages are sent to the agent:

```

    agent.Post(Init)
    agent.Post(Start)
    ()

member form.eventStopButton_Click(sender:obj, e:EventArgs) =
    agent.Post(Stop)
    ()

```

The following is the code used in `Program.fs` to start the application and view the GUI:

```

/// Program.fs
namespace Program

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

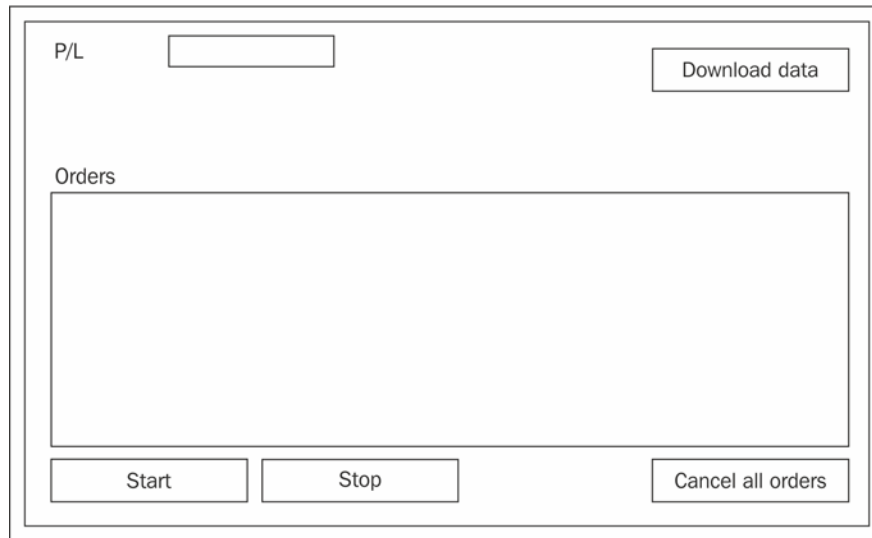
open GUI

module Main =
    [<STAThread>]
    do
        Application.EnableVisualStyles()
        Application.SetCompatibleTextRenderingDefault(false)
        let view = new TradingSystemForm()
        Application.Run(view)

```

Building the GUI

The GUI we used in the previous section is not sufficient for our trading application, but it illustrated the basics of how to put a GUI together using F#. Next, we'll add the controls needed and prepare it to present the information from the model. The following is a mock representation showing where the controls will be placed and the overall idea about the GUI:



A mock representation of the trading system's GUI

Let's look at the required code. Most of the code is straightforward, following the same rules used in the GUI in the last section. The `DataGridView` control has some properties set for the width to be adjusted automatically. The same is true for labels, where the property `AutoSize` is set to true. The final GUI will look like the one in the screenshot displayed after the following code:

```
/// GUI code according to mock
namespace GUI

open System
open System.Drawing
open System.Windows.Forms
open Agents
open Model

open System.Net
open System.ComponentModel
```

```
// User interface form
type public TradingSystemForm() as form =
  inherit Form()

  let plLabel = new Label(Location=new Point(15,15))
  let plTextBox = new TextBox(Location=new Point(75,15))

  let startButton = new Button(Location=new Point(15,350))
  let stopButton = new Button(Location=new Point(95,350))
  let cancelButton = new Button(Location=new Point(780,350))
  let downloadButton = new Button(Location=new Point(780,15))

  let ordersLabel = new Label(Location=new Point(15,120))
  let dataGridView = new DataGridView(Location=new
    Point(0,140));

  let initControls =
    plLabel.Text <- "P/L"
    plLabel.AutoSize <- true

    startButton.Text <- "Start"
    stopButton.Text <- "Stop"
    cancelButton.Text <- "Cancel all orders"
    cancelButton.AutoSize <- true
    downloadButton.Text <- "Download data"
    downloadButton.AutoSize <- true

  do
    initControls

  form.Size <- new Size(900,480)

  form.Controls.Add(plLabel)
  form.Controls.Add(plTextBox)
  form.Controls.Add(ordersLabel)

  form.Controls.Add(startButton)
  form.Controls.Add(stopButton)

  form.Controls.Add(cancelButton)
  form.Controls.Add(downloadButton)
```

```
dataGridView.Size <- new Size(900,200)
dataGridView.RowHeadersWidthSizeMode <- DataGridViewRow
    HeadersWidthSizeMode.EnableResizing
dataGridView.AutoSizeColumnsMode <- DataGridViewAutoSize
    ColumnsMode.AllCells

form.Controls.Add(dataGridView)

form.Text <- "F# TradingSystem"
```



The final GUI build from the code according to the mock

Presenting information in the GUI

In this section, we'll look at ways of presenting information in a GUI which is updated on a regular basis. We'll use the MVC pattern to update the data.

In .NET, in general, the interface `INotifyPropertyChanged` is used when the notification of an update is needed in the model. In this example, we'll use a `DataGridView` control and a `DataSource` that consists of a list with items of a custom type implementing the `INotifyPropertyChanged` interface.

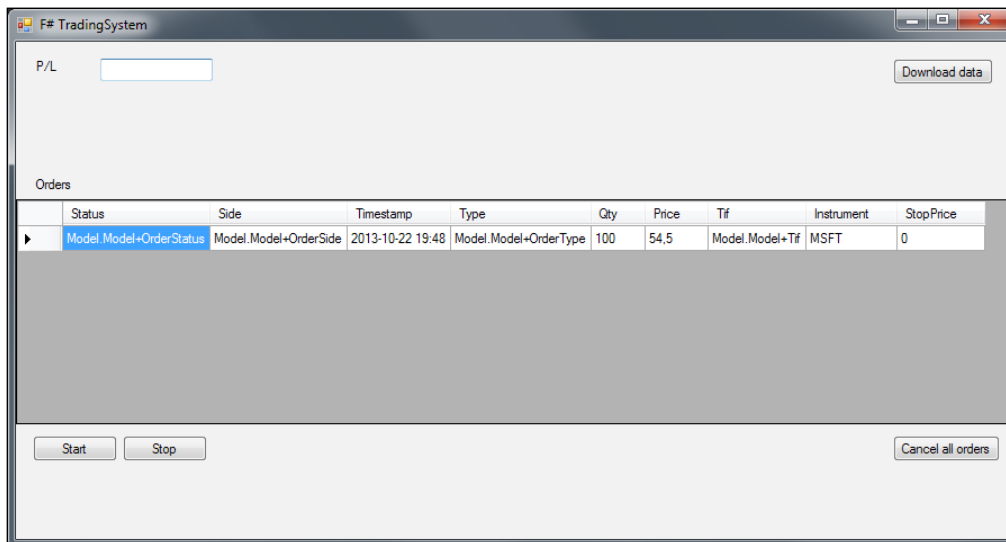
The updates to the model are handled by the controller and then the GUI is updated from the `DataSource` itself. Let's start by looking at the list of orders and how to present that list of orders in the `DataGridView` control. Add the following code to the GUI `.fs` file:

```
let initOrderList() =
    let modelList = new BindingList<Model.Order>()
    let buyOrder = Model.Order(Model.OrderSide.Buy,
        Model.OrderType.Limit, 54.50, Model.Tif.FillorKill,
        100, "MSFT", 0.0)
    modelList.Add(buyOrder)

    dataGridView.DataSource <- modelList
    dataGridView.Size <- new Size(900,200)
    dataGridView.RowHeadersWidthSizeMode <- DataGridViewRow
        HeadersWidthSizeMode.EnableResizing
    dataGridView.AutoSizeColumnsMode <- DataGridViewAutoSize
        ColumnsMode.AllCells
```

Also, add the following function call just under `initControls`:

```
initOrderList()
```



The GUI with `DataGridView` populated with an order item

As you may notice, the content in some cells is not displayed as we would like. We need to add a custom cell formatter for them where we specify in which way the value is presented in the GUI.

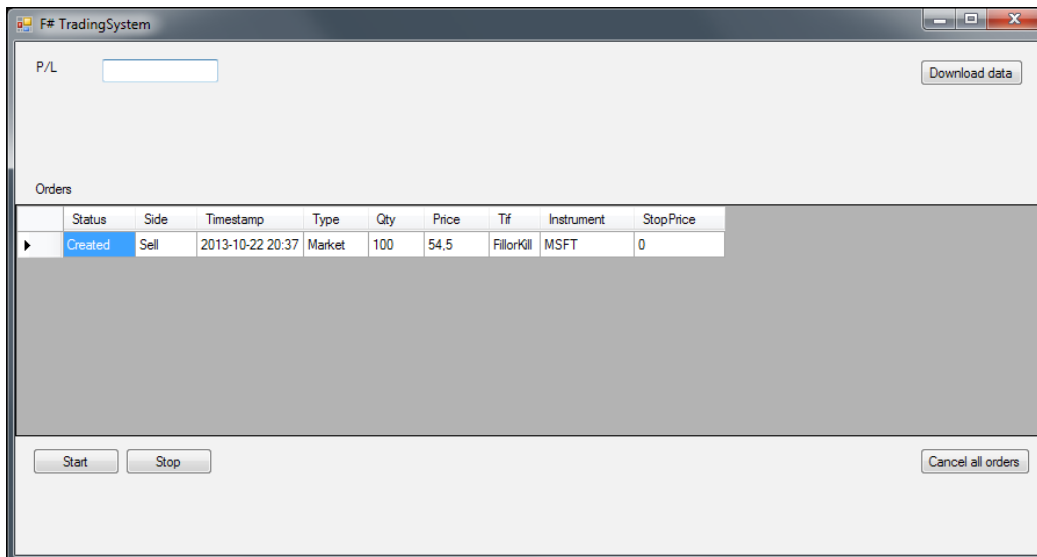
Add the following line of code at the end in the `initOrderList` function:

```
dataGridView.CellFormatting.AddHandler(new  
    System.Windows.Forms.DataGridViewCellFormattingEventHandler(fun  
        sender e -> form.eventOrdersGrid_CellFromatting(sender, e)))
```

Then, we need to implement the `eventOrdersGrid_CellFromatting` function as follows:

```
member form.eventOrdersGrid_CellFromatting(sender:obj,  
    e:DataGridViewCellFormattingEventArgs) =  
    match (sender :?> DataGridView).  
        Columns.[e.ColumnIndex].DataPropertyName with  
    | "Status" -> e.Value <- sprintf "%A" model  
        List.[e.RowIndex].Status  
    | "Side" -> e.Value <- sprintf "%A" modelList.[e.RowIndex].Side  
    | "Type" -> e.Value <- sprintf "%A" modelList.[e.RowIndex].Type  
    | "Tif" -> e.Value <- sprintf "%A" modelList.[e.RowIndex].Tif  
    | _ -> ()
```

Now when we run the program, the `DataGridView` control for the order items will format the cells correctly as shown in the following screenshot:



The GUI with `DataGridView` using a custom cell formatter

To make the GUI complete, we need to add functionalities to update text fields and to handle button clicks. We need callback functions that will be called from the controller to update the text fields in the GUI:

```
// Functions to update GUI from controller

let updatePlTextBox(str:string) =
    plTextBox.Text <- str
```

Next, we'll need event handlers for buttons. Each button will have its own event handler as follows:

```
startButton.Click.AddHandler(new System.EventHandler
    (fun sender e -> form.eventStartButton_Click(sender, e)))

stopButton.Click.AddHandler(new System.EventHandler
    (fun sender e -> form.eventStopButton_Click(sender, e)))

cancelButton.Click.AddHandler(new System.EventHandler
    (fun sender e -> form.eventCancelButton_Click(sender, e)))

downloadButton.Click.AddHandler(new System.EventHandler
    (fun sender e -> form.eventDownloadButton_Click(sender, e)))

// Event handlers
member form.eventStartButton_Click(sender:obj, e:EventArgs) =
    Controller.startButtonPressed()
    Controller.testUpdateGUI(updateSP500TextBoxPrice)

member form.eventStopButton_Click(sender:obj, e:EventArgs) =
    Controller.stopButtonPressed()
    Controller.testUpdateGUI(updateSP500TextBoxPrice)

member form.eventCancelButton_Click(sender:obj, e:EventArgs) =
    Controller.cancelButtonPressed()
    Controller.testUpdateGUI(updateSP500TextBoxPrice)

member form.eventDownloadButton_Click(sender:obj, e:EventArgs) =
    Controller.downloadButtonPressed(updatePlTextBox,
    updateSP500TextBoxPrice, updateSP500TextBoxVol,
    updateVixTextBoxPrice, updateVixTextBoxVol)
```


Adding support for downloading the data

The market data will be pulled from Yahoo! Finance on a daily basis; we'll use closing prices and from them calculate the data needed. The data will be downloaded once the **Download data** button in the GUI is clicked on. The following is the code to illustrate how downloading can be handled by a background thread:

```
let fetchOne(url:string) =
    let uri = new System.Uri(url)
    let client = new WebClient()
    let html = client.DownloadString(uri)
    html

let downloadNewData(url1:string, url2:string) =
    let worker = new BackgroundWorker()
    worker.DoWork.Add(fun args ->
        printfn("starting background thread")
        let data = fetchOne(url)
        printfn "%A" data)
    worker.RunWorkerAsync()
```

The trading system will follow these steps from the downloading process until the data is parsed:

1. Download the data from Yahoo! Finance.
2. Parse the data and perform the calculations.
3. Store the data in the model.

Looking at possible additions to the system

In this section, we'll look at possible additions to the trading system we have developed. The ideas here can work as inspiration for an interested reader. Trading systems involve topics from many areas in finance and computer science. The trading system developed here is rather elementary and is mainly for illustrative purposes.

Improving the data feed

The data feed used here isn't an actual feed; it's more of a data service. A data feed is as the name suggests: a feed of data. The data feed will provide a continuous stream of the market data to the application and follows a publisher-subscriber pattern. It's not easy to find a data feed provider that delivers data for free. The following is a list of some alternatives worth looking into:

- **Bloomberg's Open Market Data Initiative:**
<http://www.openbloomberg.com/>
- **Interactive Brokers:** <https://www.interactivebrokers.com/en/main.php>
- **eSignal Feed API:** <http://www.esignal.com/>

Support for backtesting

Backtesting is useful in many cases, the least being to verify the correctness of the trading logic. Backtesting can also provide some valuable insights into the historical performance of a trading strategy. When developing a backtesting engine, you need a feed adapter to use historical data and a broker adapter to keep track of executed orders and the profit and loss.

This data is used to calculate statistics such as:

- Total number of trades
- The ratio between winners and losers
- The average size of a trade
- Total return on an account
- Volatility and sharp ratio; sharp ratio is the volatility-adjusted return

Extending the GUI

The GUI provided for this trading system is quite limited. The GUI can easily be extended to support more features and provide charts of the market data using FSharpChart among others. One alternative is to develop the GUI using C# or another language that has a visual designer built into Visual Studio. This makes things much easier.

The main reason that the GUI is developed in F# in this book is to illustrate the flexibility of F#. When there is a visual designer for F#, there is no reason not to use F# for most parts of the program. Writing the GUI code by hand is cumbersome regardless of the language that is used.

Converting to the client-server architecture

The current architecture is better modelled as part of a client-server solution. In a client-server solution, the data feed, strategy execution, and order management will reside on the server, whereas the user interface will either be a native application or a browser implementation. There are two ways to go here. The first is to communicate with the server using a message queue such as Microsoft Message Queuing. The other is to use a browser-based GUI, communicating with the server using WebSockets and RESTful techniques.

Some useful technologies to look deeper into are:

- **Microsoft Message Queuing (MSMQ)**
- ZeroMQ
- WebSocket
- RESTful

Summary

In this chapter, we have put the pieces together we learned throughout this book, and this has resulted in a trading system for volatility arbitrage. Many aspects of F# programming and the .NET framework together with external libraries have been illustrated and covered up to now.

Index

Symbols

@ operator 20

A

active patterns 59-62

actual volatility 158, 232

aggregate statistics

about 93

average of sequence, calculating 94

example application 97, 98

maximum of sequence, calculating 95

minimum of sequence, calculating 94

standard deviation of sequence, calculating 96, 97

sum of sequence, calculating 93

variance of sequence, calculating 96

American options 136

anonymous function

creating 16

application

extending, Bollinger bands used 127-130

arithmetic comparisons 90, 91

arithmetic operators 90

ark to market (MTM) basis 232

array module

functions 40

arrays 34, 38, 39, 54

asynchronous binding 71

asynchronous programming

about 65, 70

asynchronous binding 71

F# asynchronous workflows 71

Asynchronous Programming Model (APM)

70

async workflow

example 71-74

automated trading

about 203, 204

parts 204

B

background workers 66, 67

bar chart

creating 132, 133

bisection method

about 107

example 107, 108

Black-Scholes

delta hedging 168

Black-Scholes formula

about 135, 139

assumptions 139, 140

implementing, in F# 141, 142

using, with charts 142-144

black_scholes function 142

Bollinger bands

used, for extending application 127-130

C

call option 135

CancelAsync() function 68

candlestick chart

creating, from stock prices 130, 131

classes 77

code

documenting 22

Comma-Separated Values (CSV) 25

Common Language Infrastructure (CLI) 5

conditional orders 176

conversion functions 92, 93
currying 17, 18

D

data structures

arrays 34, 38, 39
discriminated unions 33, 36, 37
enumerations 34, 38
lists 34, 41-43
maps 34, 51
options 34, 52, 53
record types 33-35
selecting 54
sequences 34, 45, 47
sets 34, 48, 49
strings 34, 53

data visualization

about 119, 120
application code 125, 126
financial data, displaying from Yahoo
 finance 124
form, extending 122, 124

Delta 146

delta hedging

Black-Scholes used 168, 169

delta-neutral portfolio

trading 231, 232

directional trading

about 226
butterfly spread, trading 228
delta-neutral portfolio, trading 231, 232
long butterfly spread 229
long straddle 227
options, used 226
short butterfly spread 230
short straddle 228
straddle, trading 226
strategies 226
VIX, trading 231

discriminated unions 33, 36, 37

E

enumerations 34, 38

European options 136

event handling 119

events 65, 66

Exotic options 136

F

F#

about 5, 6, 77, 113
Black-Scholes formula, implementing 141, 142
code documentation 22
currying 17, 18
data visualization 119
functions 15, 16
greeks, implementing 146
GUI programming 113
Hello World of finance application 22, 23
immutable variable 13
interfaces, composing 114, 115
lists 18, 19
mutable variable 13
numerical types 88
orders 175
primitive types 14, 15
programming with 12
tuples 21
URL 6
volatility, exploring 159-162

F# asynchronous workflows 71

filter function 20

Financial Information eXchange. *See* **FIX**

F# Interactive 10-12

first-order greeks 145

FIX 187

FIX 4.2

messages, using 187
using 187

FIXimulator 187

floating-point numbers

about 87
IEEE 754 floating-point standard 87

F# program

structuring 30

F# project

creating, in Visual Studio 6-8
F# script file, adding 9, 10
program template, understanding 8, 9

FsChart

about 130

- candlestick chart, creating from stock prices 130, 131
- F# script file**
 - adding 9, 10
- FSharpCharts**
 - used, for plotting payoff diagrams 224, 225
- FsUnit 205**
- functional programming**
 - about 55
 - lazy evaluation 63
 - pattern matching 57
 - recursive functions 55
 - tail recursion 56
 - units of measure 63-65
- functions**
 - about 15, 16
 - anonymous function 16
 - higher-order function 17
 - pipe operator 22
 - using, in modules 31, 32
- functions, array module**
 - Array.average a 40
 - Array.empty 40
 - Array.exists f a 40
 - Array.filter f a 40
 - Array.find f a 40
 - Array.isEmpty a 40
 - Array.length a 40
 - Array.map f a 40
 - Array.max a 40
 - Array.min a 40
 - Array.sort a 40
 - Array.zip a b 40
- functions, list module**
 - List.append a b 45
 - List.average a 44
 - List.empty 45
 - List.exists f a 45
 - List.filter f a 45
 - List.find f a 45
 - List.fold f s a 45
 - List.head a 45
 - List.length a 45
 - List.map f a 45
 - List.max a 44
 - List.min a 45
 - List.nth a 44
 - List.sort a 45
 - List.tail a 45
 - List.zip a b 45
- functions, map module**
 - Map.add(k,v) 52
 - Map.containsKey k a 52
 - Map.empty 52
 - Map.exists f a 52
 - Map.filter f a 52
 - Map.find f a 52
 - Map.fold f s a 52
 - Map.partition f a 52
- functions, sequence module**
 - Seq.append a b 48
 - Seq.average a 48
 - Seq.empty 48
 - Seq.exists f a 48
 - Seq.filter f a 48
 - Seq.find f a 48
 - Seq.fold f s a 48
 - Seq.head 48
 - Seq.length a 48
 - Seq.map f a 48
 - Seq.max a 48
 - Seq.min a 48
 - Seq.nth a 48
 - Seq.sort a 48
 - Seq.zip a b 48
- functions, Set module**
 - Set.count a 50
 - Set.empty 50
 - Set.exists f a 50
 - Set.filter f a 50
 - Set.map 50
 - Set.partition f a 50
- functions, String module**
 - String.Compare s1 s1 54
 - String.Concat s1 s2 54
 - String.Copy s 54
 - String.Empty 54
 - String.exists f s 54
 - String.IsNullOrEmpty 54
 - String.IsNullOrWhiteSpace 54
 - String.length s 54
 - String.map f s 54
- fun keyword 16**

G

Gamma 147

generics 62

greeks

about 145

code listing, for visualizing 152, 153

Delta 146

first-order greeks 145

Gamma 147

implementing, in F# 146

Rho 148

second-order greeks 145

sensitivity, investigating 149-151

Theta 148

Vega 147

guards

using 58

GUI programming

in F# 113

GUI, trading system

building 252

H

Head function 20

Hello World of finance application

about 22, 23

code listing 24

coding 26

extending 25

imperative code 27

interoperability 27

understanding 24

higher-order function 17

I

IEEE 754 floating-point standard 87, 88

immutable variable 13

imperative code 27, 28

imperative programming 77

implied volatility 232

about 158, 164

extracting 166, 167

solving method 165

incomplete pattern matching 58

integers

two's complement 86

interfaces

agents 116

composing 114, 115

event handling 119

main application 118

user interface 117

L

Language-INtegrated-Query (LINQ) 203

lazy evaluation 63

least squares method 103

Length function 20

let keyword 12

let statement 33

limit orders 176

linear regression, Math.NET library

about 103

least squares method, using 103

polynomial regression, using 104-106

list module

functions 44, 45

lists

about 18, 19, 34, 41-43, 54

concatenating 20

pattern matching 44

M

MailboxProcessor 74-76

map function 20

map module

functions 52

maps 34, 51, 55

market data 183, 184

market orders 176

mathematics

deriving 232, 233

implementing 233

implied volatility, hedging 233

Math.NET library

about 98

implementing 98

installing 98, 99

linear regression 103

probability distributions 100

- random number generation 99
- statistics 102
- Math.NET linq algebra** 98
- Math.NET neodym** 98
- Math.NET numerics** 98
- Math.NET ytrium** 98
- math operators** 91
- Microsoft SQL Server**
 - connecting to 210-212
- midpoint() function** 35
- modules**
 - about 30
 - functions, using in 31, 32
- Monte Carlo method** 135, 154, 155
- MouseDown event** 66
- mutable variable** 13
- MVC pattern, trading system**
 - controller 246, 247
 - model 246
 - view 246

N

- namespace** 32, 33
- Newton-Raphson method**
 - about 109
 - example 109
 - used, for finding roots 109
- normal distribution** 100
- number representation**
 - about 86
 - floating-point numbers 87
 - integers 86
- numerical types, F#** 88
- NUnit** 205

O

- object-oriented programming**
 - about 77
 - classes 77
 - members 78, 79
 - method 79, 80
 - objects 78, 79
 - overloaded operators 80, 81
 - properties 79, 80
- openFile function** 25

- open-high-low-close (OHLC)** 34
- open keyword** 33
- operator (:?)** 25
- options**
 - about 34, 52, 53, 135
 - American options 136
 - call option 135
 - contract specifications 136
 - European options 136
 - Exotic options 136
 - put option 135
- order execution** 182, 183
- order properties**
 - about 176, 177
 - examples 177-182
- orders**
 - about 175
 - order execution process 182
 - properties 176
- order types**
 - about 175
 - conditional orders 176
 - limit orders 176
 - market orders 176
 - stop-orders 176
- overloaded operators** 80, 81

P

- parallel programming**
 - about 65
 - background workers 66, 67
 - events 65, 66
 - thread pools 69, 70
 - threads 68, 69
 - with TPL 74
- pattern matching**
 - about 44, 57
 - active patterns 59-62
 - guards, using 58
 - in assignments 59
 - incomplete pattern matching 58
 - in input parameters 59
- payoff diagrams**
 - plotting, FSharpCharts used 224, 225
- pipe-forward operator (|>)** 22
- pipe operator** 22

polynomial regression
using 104

possible additions, trading system
backtesting 259
client-server architecture, converting to 260
data feed, improving 259
GUI, extending 259

pretrade risk analysis
implementing 185
orders, validating 185, 186
rules 185

primitive types 14, 15

probability distributions, Math.NET library 100

program template
understanding 8, 9

prototyping 27

put option 135

Q

QuickFIX
configuring 187-202

QuickFix/n library 187

R

random number generation, Math.NET library
about 99
Mersenne twister 100
pseudo-random numbers 99, 100

Read Eval Print Loop. See REPL

rec keyword 55

record types 33-35

recursive functions 55

relative value volatility trading
about 234
slope of smile, trading 234-239
strategies 234

REPL 5

requisites, trading system
libraries 206
listing 205
tools 206

Rho 148

root-finding algorithms
about 106
bisection method 107
roots, finding Newton-Raphson method used 109
roots, finding secant method used 110

S

sample application
type providers, using with LINQ 213, 214

secant method
example 110
used, for finding roots 110

second-order greeks 145

sequence module
functions 48

sequences 34, 45, 47

series.Points.Add method 126

Set module
functions 50

sets 34, 48, 49, 55

software testing, trading system 204

statistics
about 93
aggregate statistics 93

stop-orders 176

String module
functions 54

strings 34, 53

sum function 16

T

Tail function 20

tail recursion 56

Task Parallel Library. See TPL

test cases, trading system
writing 216-220

test-driven development, trading system 204

testParse function 62

Theta 148

thread pools 69

threads 68, 69

TPL
about 74
parallel programming 74

trading strategy

- defining 239
- entry rules, defining 240
- exit rules, defining 241
- slope, decreasing 240
- slope, increasing 240

trading system

- automated trading 203
- FsUnit, installing 209, 210
- functionalities 243
- GUI, building 252
- history, saving 215, 216
- information, presenting in GUI 254-257
- Microsoft SQL Server, connecting to 210, 211
- MVC pattern 246
- NUnit, installing 209, 210
- possible additions 258
- project structure 244, 245
- requisites 205, 243
- setting up 203-208
- setup details 221
- software testing 204
- state, saving 215, 216
- support, adding for downloading data 258
- test cases 216
- test-driven development 204
- trading strategy, executing 247-250
- TradingSystem 245
- TradingSystem.Tests 245
- type providers 212

tuples 21

type inference 13, 15

type providers

- about 212
- F#, using 212, 213
- LINQ, using 212, 213

U

- units of measure 63, 64, 65**

V

values

- functions, using in 31, 32
- using, in modules 31, 32

Vega 147, 164

Visual Studio

- F# project, creating in 6-8
- getting started 6

VIX

- trading 231

volatility

- about 157
- actual volatility 158
- exploring, in F# 159-162
- implied volatility 158
- trading 223

volatility smile

- about 169
- exploring 169, 171

W

when keyword 58

Wiener processes 136-139

X

XML documentation 81, 82

XML tags

- about 82
- exception 82
- remark 82
- returns 82
- see also 82
- summary 82



Thank you for buying **F# for Quantitative Finance**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

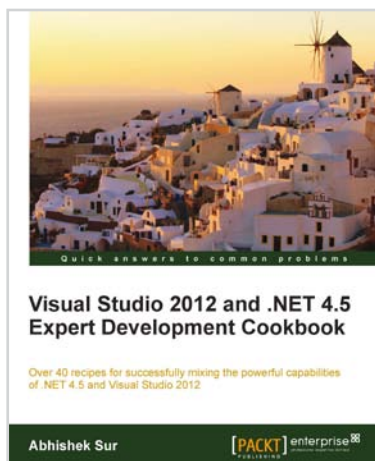


Windows Phone 7.5 Application Development with F#

ISBN: 978-1-84968-784-3 Paperback: 138 pages

Develop amazing applications for Windows Phone using F#

1. Understand the Windows Phone application development environment and F# as a language
2. Discover how to work with Windows Phone controls using F#
3. Learn how to work with gestures, navigation, and data access



Visual Studio 2012 and .NET 4.5 Expert Development Cookbook

ISBN: 978-1-84968-670-9 Paperback: 380 pages

Over 40 recipes for successfully mixing the powerful capabilities of .NET 4.5 and Visual Studio 2012

1. Step-by-step instructions to learn the power of .NET development with Visual Studio 2012
2. Filled with examples that clearly illustrate how to integrate with the technologies and frameworks of your choice
3. Each sample demonstrates key concepts to build your knowledge of the architecture in a practical and incremental way

Please check www.PacktPub.com for information on our titles



Microsoft .NET Framework 4.5 Quickstart Cookbook

ISBN: 978-1-84968-698-3 Paperback: 226 pages

Get up to date with the exciting new features in .NET 4.5 Framework with these simple but incredibly effective recipes

1. Designed for the fastest jump into .NET 4.5, with a clearly designed roadmap of progressive chapters and detailed examples
2. A great and efficient way to get into .NET 4.5 and not only understand its features but clearly know how to use them, when, how, and why
3. Covers Windows 8 XAML development, .NET Core (with Async/Await and reflection improvements), EF Code First and Migrations, ASP.NET, WF, and WPF



C# 5 First Look

ISBN: 978-1-84968-676-1 Paperback: 138 pages

Write ultra responsive applications using the new asynchronous features of C#

1. Learn about all the latest features of C#, including the asynchronous programming capabilities that promise to make apps ultra-responsive
2. Examine how C# evolved over the years to be more expressive, easier to write, and how those early design decisions enabled future innovations
3. Explore the language's bright future building applications for other platforms using the Mono Framework

Please check www.PacktPub.com for information on our titles

