August 26, 2009

Chapter 4: Value Types and Operators

Most financial programming involves making mathematical calculations. As in algebra, we often use variables in computer programs to hold different values we need for calculation. In this chapter, you will learn how to declare variables and perform calculations in .NET.

4.1 Declaring Variables

To a computer, primitive or simple value ***types***, called variables, are actual, physical spaces in memory that store data for use by our program. Before we can use a variable, we need to declare it using the Dim statement. That is, we have to tell the computer to set up a space in memory with a specific name. In programming, the variable names we use are usually descriptive of the contents they hold. For example, a program to analyze stock returns might contain variables like this:

VB

```
Dim m_MondayClose, m_StockPrice As Integer
Dim m_CallDelta as Double
```

C#

```
int m_Monday;
double m_CallDelta;
```

These lines of code set up variables, physical places in memory, that will be known by the names m_MondayClose, m_StockPrice, m_CallDelta and strTicker. Furthermore, the types of data that will go into each of these containers will be things called a single, a double and a string. Single, double and string are value types, which tell us what kind of data the variable can hold. Here is a list of the different value types supported by VB.NET, with descriptions:

Value Type with Identifier	**Range**	**Note**	**Example Using Naming Convention and Value Type Identifier**
Boolean	True or False	16 bits. Stored internally as an either 0 or 1.	Dim m_BuySell As Boolean
Char	Any Unicode character	Character codes 0 to 65,535	Dim m_ExpMonth as Char
DateTime	1/1/01000 to 12/31/9999 and 0:00:00 to 23:59:59	64 bits. Holds dates and times.	Dim m_ExpDate As DateTime
Decimal	1.0E-28 to 7.9E+28	128 bits. Large numbers.	Dim m_Covar As Decimal
Double	+/-5.0E-324 to +/-1.7E+308	64 bits. Double-precision floating-point variable.	Dim m_CallDelta As Double
Integer	-2,147,483,648 to 2,147,483,647	32 bits. Integers only. No decimal numbers.	Dim m_NumShares As Integer
Long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	64 bits. Big integers, but still no decimal numbers.	Dim m_NumTrades As Long
Short	-32,768 to 32,767	16 bits. Small integers only. No decimal numbers.	Dim m_NumContracts As Short

Single !	+/- 1.5E-45 to +/-3.4E+38	32 bits. Single-precision floating-point variable.	Dim m_StockPrice as Single
String $	Varies based upon the number of characters.	Character Data.	Dim strTicker as String

When a variable of any type is created, its default value is 0. We can define, or change the values of our variables this way:

```
m_MondayClose = 10.12
```

Alternatively, we could declare and define a variable in the same line:

```
Dim m_StockPrice as Integer = 4.92
```

In .NET all variables must be declared before they can be used. Later in the book, we will show you that this helps avoid common programming errors.

4.1.1 Constants

If the value of a variable is not going to change over the life of our program, we should declare it as a constant rather than a variable like this:

```
Const DIVISOR = 1.8
```

Declaring a value as a constant protects it against accidentally being changed down the road.

4.1.2 Variable Scope

Variables and constants can also be declared using an access modifier. Access modifiers serve to specify the scope and accessibility of the variable. The access modifiers are Friend, Private, Protected, Protected Friend and Public. Here is an example:

```
Dim m_Price as Double
```

In later chapters, we will discuss access and scope in more detail. For now, be aware that the scope of a variable refers to the parts of a program that can access a variable. Not all variables are accessible everywhere. Variables in Visual Basic.NET can have the following scope:

Scope	Accessibility or "Visibility"
Class	Accessible in what is known as the declaration space of the class.
Module	Accessible to all functions and procedures defined in the module.
Global or Namespace	Accessible anywhere in a project.
Block	Accessible only within the block of code in which they are declared.

Variables should always be defined with the smallest possible scope. Variables with global scope can make the logic of an application extremely difficult to understand and make the reuse and maintenance of your code more difficult. In a Visual Basic.NET application, global variables should be used only when there is no other convenient way to share data between parts of your program. When global variables must be used, it is good practice to declare them all in a single module, grouped by function. For now, just be aware that not all variables are accessible from everywhere in our applications. The access modifiers will limit the visibility of variables.

4.1.3 Representing Dates and Times

When making financial calculations, we also often need to represent dates and times in our programs for things like interest accrual and trade time stamps.

VB

```
Dim m_Date As DateTime
m_Date = DateTime.Now
```

C#

```
DateTime m_Date;
m_Date = DateTime.Now;
```

.NET is sensitive to the cultural differences in date representation. For example, if you are working in the United Kingdom and rerun the above example, the first four dates are interpreted as the 2nd of January, rather than the 1st of February. If you are writing programs for international use, make sure you read Appendix B, Developing World Ready Applications.

4.1.4 Option Strict

In Visual Basic.NET an Option Strict On statement should always appear in the declarations section of a module. Option Strict On prevents Visual Basic.NET from making implicit type conversions that may involve loss of data. For purposes of demonstration in this book, however, we will leave the default Option Strict Off. Just remember, in the real world you should always have the Option Strict On statement at the top in your programs. In C#, there is no such statement. Explicit type conversion is always required.

4.1.5 Structures

Generally, when a group of data fit together, but consist of different value types, we may prefer to create our own variable type, called a structure. Visual Basic.NET allows us to create our own user-defined value types using the Structure statement. Our structures will generally contain more than one element and each element must be declared with an access modifier. Here is an example of a user-defined data type called QuoteData:

VB

```
Structure QuoteData
      Public dtDate as Date
      Public m_Open as Double
      Public m_High as Double
      Public m_Low as Double
      Public m_Close as Double
      Public lngVolume as Long
End Structure
```

C#

```
struct QuoteData
{
      public DateTime m_Time;
      public double m_Open;
      public double m_High;
      public double m_Low;
      public double m_Close;
      public int m_Volume;
};
```

We can then declare a variable of the type QuoteData in the following way:

VB

```
Dim m_StockPrice as QuoteData
TextBox1.Text = m_StockPrice.dtDate
TextBox2.Text = m_StockPrice.m_Open
```

C#

```
QuoteData m_Data;
m_Data.m_High = 324.54;
TextBox1.Text = m_Data.m_High;
```

4.1.6 Enumerations

Enumerations are integer value types that have a limited set of acceptable values. VB.NET allows us to create enumerations using the Enum statement, the integer value type--byte, short, integer, long--and the acceptable values.

VB

```
Enum TradeStatus as Integer
      Filled
      Open
      Partial
      Canceled
      Rejected
End Enum
```

C#

```
enum TradeStatus
{
      Filled, Open, Partial, Canceled, Rejected
};
```

We can use this enumeration by calling on of its member names in code as follows:

VB

```
Dim myTrade as TradeStatus = TradeStatus.Partial
```

C#

```
TradeStatus m_Trade = TradeStatus.Partial;
```

Enumerations make it easier to understand the purpose of variables with a limited number of allowable values as opposed to the integer values.

4.2 Operators

.NET has a wealth of operators to handle mathematical calculations and other logical operations. As we go through the book, we will be making extensive use of operators as we write programs. Most of them are self-explanatory, but some may not be. You can use this section as a reference as they come up over the course of the book.

4.2.1 Arithmetic Operators

Math Operator	Name	Example	Description
^	Exponentiation	x^y	Raises x to the power of y (VB.NET only)
-	Negation	-y	Negates y
*	Multiplication	x*y	Multiplies x and y
/	Division	x/y	Divides x by y and returns a floating point result
\	Integer division	x\y	Divides x by y and returns an integer result
Mod	Modulos	x Mod y	Divides x by y and returns the remainder
+	Addition	x + y	Adds x and y
-	Subtraction	x – y	Subtracts y from x

4.2.2 Comparison Operators

Comparison Operator	Description	Example
=	Equal	m_StockPrice = 5.67
<>	Not Equal	intNumShares <> 500
>	Greater Than	m_CallDelta > .5
<	Less Than	intVolume < 10000
>=	Greater Than or Equal	m_ClosePrice >= 52.50
<=	Less Than or Equal	m_HighPrice <= m_PreviousClose

4.2.3 Assignment Operators

Assignment Operator	Example	Explanation	New Value
Assume that m_Price = 10.00 and strTicker = “PKR”			
+=	m_Price += 3	m_Price = m_Price +3	m_Price = 13.00
-=	m_Price -= 2.00	m_Price = m_Price – 2.00	m_Price = 8.00
*=	m_Price *= 1.15	m_Price = m_Price * 1.15	m_Price = 11.5
/=	m_Price /= 2	m_Price = m_Price / 2	m_Price = 5
\=	m_Price \= 3	m_Price = m_Price \ 3	m_Price = 3
^=	m_Price ^= .2	m_Price =m_Price ^ .2	m_Price = 1.5849
&=	strTicker &= “Q”	strTicker = strTicker & “Q”	strTicker = “PKRQ”

4.2.4 Logical Operators

Logical Operator	Description	Example
And (VB.NET) && (C#)	Evaulates to True only if both conditions are true.	m_Price > 55 And m_Price < 56
Not (VB.NET) ! (C#)	Reverses or negates the meaning of an operand.	
Or (VB.NET) || (C#)	Evaluates to True if one or both conditions are true.	m_Price > 55 Or m_Price < 40

4.2.5 Concatenation Operators

Concatenation Operator	Description	Example
& (VB)	Concatenates or binds a number of strings together. (Preferred)	m_Tick = m_Symbol & “Q”
+ (C#)	Concatenates or binds a number of strings together.	m_Tick = m_Symbol + “Q”

4.3 Control Structures

The code we have written thus far has all been linear, or sequential, in nature. That is, lines of code were executed in order, one after the other, till the end of the program. Although this is fine for very short tasks, to tackle more complex situations, we will need to employ control structures, which involve the use of program flow statements. Program flow statements fall into one of two categories:

- Decision Structures. Conditional statements in which code is executed based on whether or not a condition is met.

• Repetition Structures. Looping statements, in which code is executed repeatedly either a number of times or until a condition is met.

4.3.1 If…Then…Else Statement

The If...Then...Else statement lets us say, in effect, "If this is true, then do this; otherwise, do that." The logic couldn't be more intuitive. The following example illustrates the use of the If…Then…Else structure.

VB

```
If m_StockPrice > 55 OrElse m_StockPrice < 40 Then
      Console.WriteLine("SELL!!!")
Else
      Console.WriteLine("HOLD")
End If
```

C#

```
if ( m_StockPrice > 55 || m_StockPrice < 40 )
{
      Console.WriteLine( "SELL!" );
}
else
{
      Console.WriteLine( "HOLD!" );
}
```

In the example, the statements following the If are executed only if the expression evaluates to True, that is if the stock price is greater than 55 or less than 40. The Else block of code executes if the expression evaluates to False. So, in this case, if the stock price is between 40 and 55, we will hold. The expression used in If...Then is a Boolean expression, true or false. The use of the Else block in an If statement is optional.

4.3.2 The Select…Case Statement

The Select…Case structure is very similar to the If…Then…Else structure, but it is much more efficient and makes our code much more readable if there are several branches to the decision structure. In the Select...Case structure we can include an unlimited number of clauses. Let's look at an example that not only illustrates the logic statements within a Select...Case framework, but also demonstrates how to build a histogram of log returns:

VB

```
Dim m_Bin1, m_Bin2, m_Bin3, m_Bin4, m_Bin5, m_Bin6 as Integer
Dim m_Return As Double = Math.Log( 51 / 50 )
Select Case m_DailyReturns
      Case Is < -.02
           m_Bin1 += 1
      Case –.02 To –.01
           m_Bin2 += 1
      Case -.01 To 0
           m_Bin3 += 1
      Case 0 To .01
           m_Bin4 += 1
      Case 0.01 To .02
           m_Bin5 += 1
      Case Is > .02
           m_Bin6 += 1
      Case Else
           MessageBox.Show("Error")
End Select
```

C# Cannot to ranges in C#. Also can only switch on ints and chars.

```
switch ( r )
{
      case 1:
            m_Bin1 += 1;
            break;
      case 2:
            m_Bin2 += 1;
            break;
      case 3:
            m_Bin3 += 1;
            break;
      case 4:
            m_Bin4 += 1;
            break;
      case 5:
            m_Bin5 += 1;
            break;
      default:
            MessageBox.Show("Error");
 };
```

Since the natural log of (51 / 50) is .0198, the value of m_Bin5 will be incremented by 1. The Case Else clause at the end of the structure is optional. Also, multiple conditions are evaluated separately with a logical OR as opposed to an AND, so it's best to keep Select…Case logic as simple as possible. Let's look at another example evaluating strings.

Call and put option symbols include a strike price and expiration month. The second to last letter in the symbol denotes the month of expiration and the last term the price. So, for example, GEKD would be the symbol for the General Electric November 20.00 calls. GERT would be the June 17.50 puts. We will have more examples using option symbols later in the book, but here is a Select…Case structure using the char data type to determine the month of expiration:

VB

```
Dim m_Month as Char = "D"
Dim m_Month as String
Select Case m_Month
            Case "A", "a", "M", "m"
                m_Month = "January"
            Case "B", "b", "N", "n"
                m_Month = "February"
            Case "C", "c", "O", "o"
                m_Month = "March"
            Case "D", "d", "P", "p"
                m_Month = "April"
            Case "E", "e", "Q", "q"
                m_Month = "May"
            Case "F", "f", "R", "r"
                m_Month = "June"
            Case "G", "g", "S", "s"
                m_Month = "July"
            Case "H", "h", "T", "t"
                m_Month = "August"
            Case "I", "i", "U", "u"
                m_Month = "September"
            Case "J", "j", "V", "v"
                m_Month = "October"
            Case "K", "k", "W", "w"
                m_Month = "November"
            Case "L", "l", "X", "x"
                m_Month = "December"
        End Select
```
Since the value of chrMonth is “D,” the value of strMonth will be set to “April.”

4.4 Repetition Structures

.NET provides a number of different types of loops that you can use to implement repetitive operations.

4.4.1 The For...Next Loop

The `For...Next` loop executes a series of statements a specific number of times. The basic syntax is:
VB
```
For x = 0 to 10 Step 2
      Console.Writeline(“Your stock is down “ & x & “ points.”)
Next x
```
C#
```
for (int x = 0; x < 10; x++)
{
      Console.WriteLine("Your stock is down " + x + " points.");
}
```
Here, the program will loop through this code five times, starting with x = 0. Each time it loops x will incremented by two until the maximum value of x, in this case 10, is reached. In the example above, the printout will show our stock fall by 2 points with each successive loop. If the Step phrase is left out, your program will automatically increment the loop counter variable by +1. Let’s take a look at this code:
```
For x = 1 to 5
      m_Sum += x
Next x
```
After completing the loop, the value of m_Sum = 1 + 2 + 3 + 4 + 5 = 15

4.4.2 The Do While Loop

Here is an example of a Do…While Loop.
VB
```
Dim m_StockPrice As Double = 35
Do While m_StockPrice < 100
      m_StockPrice += 1
Loop
Console.WriteLine(“The stock price is “ & m_StockPrice)
```
C#
```
double m_StockPrice = 35;
while ( m_StockPrice < 100 )
{
   m_StockPrice += 1;
}
Console.WriteLine("The stock price is " + m_StockPrice);
```
When this loop is finished, it prints out the price as 100. This routine evaluates m_StockPrice < 100 each time through the loop. When m_StockPrice = 99, the loop increments m_StockPrice to 100. The next evaluation of m_StockPrice = 100 is False so program execution exits the loop and continues with the line after the Loop statement, printing m_StockPrice as 100.

4.4.3 The Do…Until Loop

Here is an example of a Do…Until Loop.
VB

```
Sub Main()
     Dim m_SellPrice as Double = 95
     Dim m_StockPrice as Double = 45
     Do Until m_StockPrice >= m_SellPrice
           m_StockPrice *= Math.Exp(0.1)
           Console.WriteLine("We are still holding the stock.”)
     Loop
     Console.WriteLine("We have sold the stock at " & m_StockPrice)
End Sub
```

As with the Do...While loop, the Do...Until is not necessarily executed at all since the program evaluates the exit condition before entering the loop. In this example, we sold the stock at 100.149.

4.4.4 The Do…Loop While Loop

To make sure that a loop executes at least once, place the exit condition at the Loop statement, rather than at the Do statement, as in the following:

VB

```
Sub Main()
      Dim m_StockPrice# = 35
      Do
            Console.WriteLine("Incrementing the stock price.”)
            m_StockPrice -= 1
      Loop While m_StockPrice > 30
      Console.WriteLine(“Sold the stock at “ & m_StockPrice)
End Sub
```

In this program the stock is sold at 30.

4.4.5 The Do… Loop Until Loop

You can similarly put the Until condition at the end of a loop. In the previous example you knew you wanted to go through the loop at least once. By putting the Until statement at the end, you don't need to worry about the initial value of the variable.

VB

```
Sub Main()
      Dim m_StockPrice# = 35
      Do
            m_StockPrice -= 1
      Loop Until m_StockPrice = 25
      Console.WriteLine(“We sold the stock at “ & m_StockPrice)
End Sub
```

In this program, the stock is sold at 25.

4.4.6 The While...End While Loop

VB.NET also provides another general-purpose loop statement called the While ...End While loop. The While...End While loop has the following syntax:

VB

```
Sub Main()
      Dim m_StockPrice# = 35
      While m_StockPrice <= 50
            m_StockPrice += 1
            Console.WriteLine("Holding the stock.")
      End While
      Console.WriteLine("We sold the stock at " & m_StockPrice)
End Sub
```

In this program the stock is sold at 51.

4.5 The Exit Commands

There are occasions where you need to break out of a loop. In such a case we can insert an Exit command. Depending on which type of loop structure you are using, you will use the Exit For command, or the Exit Do command. We might generally do this inside an If…Then statement inside a loop. Here is an example of an infinite loop. The Do While 1 statement will never evaluate to False, so this program will loop forever until some event causes an exit from the loop. As you can imagine, its best to be very careful with infinite loops.

VB

```
Sub Main()
      Dim m_StockPrice# = 35
      Do While 1
            m_StockPrice += 1
            If m_StockPrice > 100 Then
                  Exit Do
            End If
      Loop
      Console.WriteLine("We sold the stock at " & m_StockPrice)
End Sub
```

C#

```
break;
```

In this program, we sold the stock at 101.

4.6 Nested Loops

You can put a `For...Next` loop inside of another `For...Next` loop. Consider the following example showing nested For...Next loops to transpose a matrix. Again, we haven't looked at arrays yet, so don't worry about the variable references. For now, just note the structure of embedded loops.

VB

```
For x = 0 To intRows
      For y = 0 To intCols
            outArray(y, x) = inArray(x, y)
      Next y
Next x
```

Although For...Next loops are useful when we know in advance how many times we want to execute the loop, there are occasions when we do not have this information in advance.

4.7 Estimating and Forecasting Volatility

Often when analyzing financial data, we often estimate volatility over a period of time in the past. This is easily done if we have a time series of price data, as was the case in Project 4.1 where we used four log returns to calculate the standard deviation of returns. If we have several years of historical data, we can estimate the daily volatility by simply calculating the standard deviation of daily log returns.

How though do we estimate volatility given only one day of data? Usually, we would use the same method. We estimate one day standard deviation using close-to-close data as follows:

$$\sigma_{CC} = \sqrt{\left[\ln\left(\frac{C_i}{C_{i-1}}\right)\right]^2}$$

However, this method certainly does not capture all of the information of intra-day volatility. A stock could close at 50 one day, gap open to 53 the following day, trade down to 44 and close back at 50. In this case, using this close-to-close calculation would not be a very good indicator of volatility since "0" is not a good description of what happened.

To better account for one-period volatility, several other, more efficient methods have been proposed which use intra-period highs and lows to estimate volatility. These methods are often grouped under the term "extreme value estimators." Since several models that we use in financial markets are based on the assumption of continuous time, it is more intuitive to examine the entire time period rather than simply the ends. The most well-known of the extreme value estimators have been proposed by Parkinson (1980), and Garman and Klass (1980) (Nelken, 1997). The Parkinson's equation uses the intra-period high and low thusly:

$$\sigma_P = .601\sqrt{\left(\ln\frac{H_i}{L_i}\right)^2}$$

The Garman Klass estimator, which the intra-period high and low as well as the open and close data, has the form:

$$\sigma_{GK} = \sqrt{\left[\frac{1}{2}\left(\ln\frac{H_i}{L_i}\right)^2 - \left(2\ln(2)-1\right)\left(\ln\frac{C_i}{O_i}\right)^2\right]}$$

Notice that these equations represent and estimate of the one period historical volatility of the underlying. You may notice, however, that neither of these models take into account gaps, either up or down, from the previous days close. Volatility that happens over night will in neither of these models be accounted for. For this and other reasons there are dozens of derivatives of these two extreme value estimators currently in use. We will not examine any of them beyond the two standard models presented.

These Parkinson and Garman Klass models estimate past volatility. They do not forecast future volatility. Forecasting volatility is its own subject and is the topic of literally hundreds on research papers and books. The most popular models for forecasting volatility are the GARCH family.

Dozens of variations of GARCH (Generalized Autoregressive Conditional Heteroscedasticity) models have been proposed for forecasting volatility based on the assumption that returns are generated by a random process with time-varying and mean reverting volatility (Alexander, p. 65). That is, in financial markets, periods of low volatility tend to be followed by periods of low volatility, but are interspersed with periods of high volatility. The most commonly referenced GARCH model for forecasting variance is GARCH(1,1):

(5.1) $$\hat{\sigma}_{t+1}^2 = (1-\alpha-\beta)\cdot V + \alpha r_t^2 + \beta\hat{\sigma}_t^2$$

and

(5.2) $$\hat{\sigma}_{t+j}^2 = V + \left(\alpha+\beta\right)^{j-1}\cdot\left(\hat{\sigma}_{t+1}^2 - V\right)$$

where α and β are optimized coefficients, r is the log return and V is the sample variance over the entire data set. Determining the values of these coefficients, α and β, is in itself an art and a science called optimization. In a later chapter we will discuss how to employ an optimization engine to calculate the values of these coefficients using maximum likelihood methods. For now, lets get familiar with forecasting variance, and therefore the volatility, of an underlying stock for use in option pricing.

Since the variance forecasts are additive, we can estimate the volatility between now, time t, and expiration h days in the future in the following way:

(5.3) $$\hat{\sigma}_{t,t+h}^2 = \sum_{j=1}^{h}\hat{\sigma}_{t+j}^2$$

So, if 10 days remain to expiration, we first calculate the forecast of variance for t+1, or tomorrow, using equation 5.1. Then we can calculate the individual forecasts for the remaining 9 days using equation 5.2. Summing them up, we get a forecast of variance from today until expiration 10 days from now. From there, we can easily calculate an annualized volatility, which may or may not differ from a market implied volatility in an option.

Let's create a Windows application that uses a For…Next Loop to forecast volatility for a user-defined number of days ahead.

STEP 1 Open VB.NET, select New Project. In the New Project Window, select Windows Application and give your project the name GARCH and a location of C:\ModelingFM.

STEP 2 Now, that the GUI designer is on your screen, from the Toolbox, add to your form a button, named Button1, a text box, named TextBox1 and a label, named Label1. In the Properties Window for Button1, change the text property to "Calculate." You should also clear the text property for the TextBox1 and Label1.

STEP 3 In the Solution Explorer Window, click on the View Code icon to view the Form1 code window.

In this project, we will demonstrate the use of a user-defined value type, called QuoteData, as well as other data types. You may remember the discussion of a QuoteData type in the previous chapter. In any case, we need a construct to hold price data and the QuoteData type works nicely. Before we can use the QuoteData type, we need to define it for the compiler. Then we can declare some variables, known as m_Monday and m_Tuesday, as QuoteDatas.

STEP 4 In the code window, change the code to the following:

```
Public Class Form1
      Inherits System.Windows.Forms.Form
Windows Form Designer generated code
      Structure QuoteData
            Public m_Open As Double
            Public m_High As Double
            Public m_Low As Double
            Public m_Close As Double
      End Structure
      Dim m_Monday As QuoteData
      Dim m_Tuesday As QuoteData
End Class
```

STEP 5 In the Class Name combo box at the top left of your code window, select Form1. In the Method Name combo box at the top right of your code window, select Form1_Load. A code stub for the Form1_Load event handler will appear. Within this subroutine add the following code to define the contents of m_Monday and m_Tuesday.

VB

```
Private Sub Form1_Load(ByVal sender …) Handles MyBase.Load
        m_Monday.m_Open = 50
        m_Monday.m_High = 51.25
        m_Monday.m_Low = 49.75
        m_Monday.m_Close = 50.5
        m_Tuesday.m_Open = 50.5
        m_Tuesday.m_High = 51.0
        m_Tuesday.m_Low = 48.5
        m_Tuesday.m_Close = 49.5
End Sub
```

STEP 6 We have now defined two daily bars for a stock. From here we can add code to forecast volatility. In the same way as in Step 5, select the Button1_Click event. Within this subroutine add the following code to declare and define some variables and calculate the volatility forecast according to the GARCH(1,1) formula.

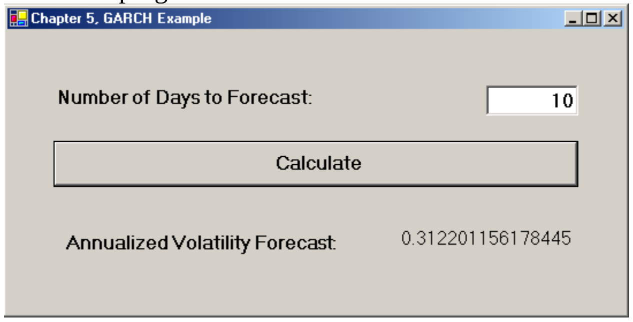
VB

```
Private Sub Button1_Click(ByVal sender …) Handles Button1.Click
     Dim m_SampleVariance as Double = 0.0002441  ' V is the equation
     Dim m_Alpha as Double = 0.0607       ' Optimized coefficient
```

```
    Dim m_Beta as Double = 0.899          ' Optimized coefficient
    Dim m_PrevForecast as Double = 0.0004152
    Dim m_TotalForecast, x As Double
    Dim m_OneDayAheadForecast as Double = (1 – m_Alpha – m_Beta) * _
        m_SampleVariance + m_Alpha * Math.Log(m_Tuesday.m_Close _
        / m_Monday.m_Close) ^ 2 + m_Beta * m_PrevForecast
   For x = 1 To TextBox1.Text
        m_TotalForecast += (m_SampleVariance + (m_Alpha + _
        m_Beta)  ^ (x - 1) * (m_OneDayAheadForecast - _
        m_SampleVariance))
    Next x
    ' Calculate the annualized volatility forecast.
    Label1.Text = m_TotalForecast ^ 0.5 * (256 / 10) ^ 0.5
End Sub
```

The GARCH(1,1) equation forecasts variance. The square root of this 10 day variance forecast will give us a 10 day volatility forecast. Multiplying this by the square root of 256 trading days divided by 10 gives us an annualized volatility number.

STEP 7 Run the program.



Number of Days to Forecast: 10

Annualized Volatility Forecast: 0.312201156178445

4.8 Stock Index Futures

The most widely traded equity index futures contract in the U.S. is the S&P 500. The futures contracts on the S&P 500 index are traded at the Chicago Mercantile Exchange (CME). The value of the contract is $250 times the futures price. The CME's "e-Mini" contract is a smaller, electronically-traded version of the original pit-traded contract and has a value of $50 times the futures price. So, if the futures contract was valued at 1000, it would have a notional value of $250,000 and the "e-Mini" a notional value of $50,000. The CME also trades options on these futures contracts. The Chicago Board Options Exchange (CBOE) trades options on the cash S&P 500 index. The S&P 500 Index consists of 500 stocks, each selected for their market size, liquidity, and industry group. Also, the S&P 500 is a market value weighted index where the market value of an individual stock is the stock price times the number of shares outstanding. Each stock's weight in the Index then is proportionate to its market value. The weights for the individual stocks change as their respective prices rise and fall relative to other stocks in the index (Kolb, 1997). Alternatively, an index could be price weighted, where the index weights are proportional to the stock prices. The Dow Jones Industrial Average is an example of a price weighted index.

Here is an example of a formula for the calculation of the *cash* value of a market value weighted index:

$$S\&P\ 500 = \left(\frac{\sum_{i=1}^{500} N_i P_i}{O.V.} \right) \times 10$$

where:

O.V. = The original valuation.
N_i = Number of shares outstanding for the ith firm.
P_i = Price per share of the ith firm.

Let’s build a simple program that will calculate the price of a market value weighted stock index. In this example, we will demonstrate the simplest type of computer program, one that uses procedural programming techniques. Procedural programs are those written as lists of instructions divided into sections or units of code called the main block, plus subroutines and functions which we will look at in Chapter 6. Procedural programming works well for small projects because it is very intuitive. Moreover, machine code is procedural, so compiling procedural code is very efficient.

STEP 1 Open the Visual Basic.Net IDE. For this exercise we are going to create a new Console application, so click on the icon named “Console Application” and name the project “IndexFutures.” A console application is the simplest type of VB.NET program and contains only text input and output as you will see. The interface will be a command, or console, window.

STEP 2 When the project IDE opens up, you will be presented only with a window in which to write code. Within the “Sub main()” procedure, we need to create the necessary variables and algorithms to make our calculations.

For simplicity, we will assume that there are two stocks in this index, known as stock A and stock B, and that it is a market value weighted index like the S&P 500. Also, to keep things simple, we will not add Option Strict to our code.

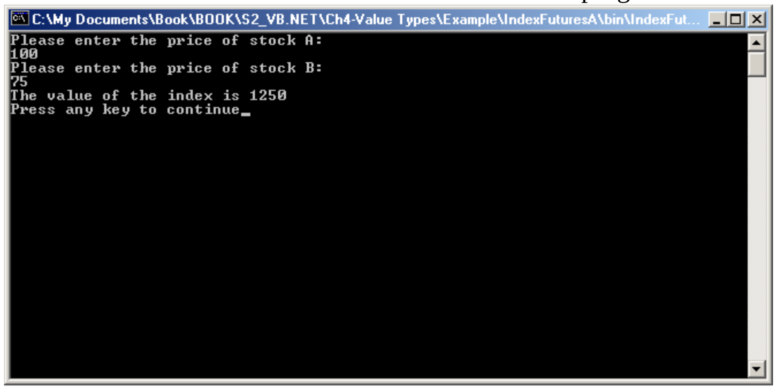
STEP 3 Now, let’s add some code to calculate the index value. To do this, we will need o declare and define some variables and use some mathematical operators according to the formula.

VB

```
Module Module1
    Sub Main()
      Const ORIGINALVALUE = 2000      ' Index original value
      Dim m_IndexValue As Double
      Dim m_SharesA% = 1000          ' 1000 shares of A outstanding
      Dim m_SharesB% = 2000          ' 2000 shares of B outstanding
      Console.WriteLine("Please enter the price of stock A:")
      Dim m_PriceA# = Console.ReadLine
      Console.WriteLine("Please enter the price of stock B:")
      Dim m_PriceB# = Console.ReadLine
 'Calculate the value of the index and print it to the screen.
      m_IndexValue = (((m_PriceA * m_SharesA) + (m_PriceB * _
                        m_SharesB)) / ORIGINALVALUE) * 10
      Console.WriteLine("The value of the index is " & m_IndexValue)
    End Sub
End Module
```

You will notice in the code above, we have included some sample values for the Original Value and the number of shares outstanding. We will allow the user to enter the prices of stocks A and B when the Console.Readline statements are executed. Notice that we have used the double value type for our variables using both the type name and the identifier for illustration purposes. Also, we have declared the original value of the index as a constant.

STEP 4 Once your code is finished, run the program by selecting from the menu bar Debug > Start Without Debugging. This will cause the program to pause before it closes the console window so we can examine the results of our program.



C:\My Documents\Book\BOOK\S2_VB.NET\Ch4-Value Types\Example\IndexFuturesA\bin\IndexFut...

```
Please enter the price of stock A:
100
Please enter the price of stock B:
75
The value of the index is 1250
Press any key to continue_
```

Let's augment this program to calculate the fair value of a futures contract on this index. We can calculate the fair value using the cost-of-carry model:

$$F_{0,t} = S_0(1+R\frac{T}{360}) - \sum_{i=1}^{n} D_i(1+R\frac{\tau_i}{360})$$

where:

$F_{0,t}$ =	Index futures price at time 0 and expire t days in the future.
S_0 =	Value of the market value weighted cash index at time 0.
R =	Interest rate.
T =	Number of days till futures expiration.
D_i =	Amount of the ith dividend.
τ_i =	Number of days the ith dividend will be invested from receipt until futures expiration.

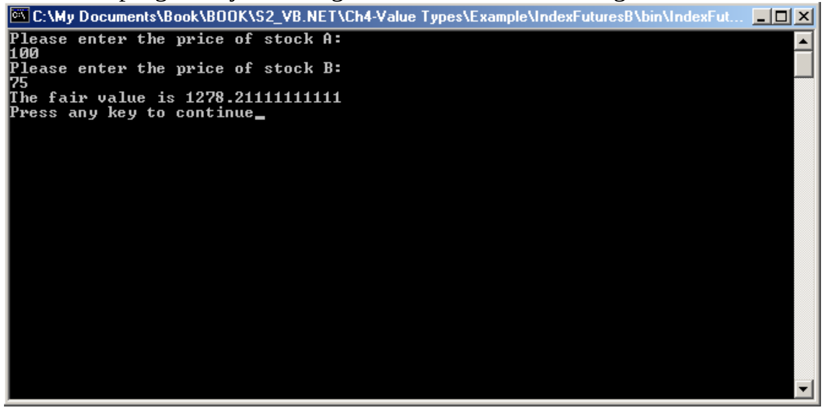
STEP 6 Change the code so as to calculate the fair value of a futures contract.

```
Module Module1
    Sub Main()
      Const ORIGINALVALUE = 2000            ' Index original value
      Dim m_FairValue, m_IndexValue As Double
      Dim m_DaysTillExp As Double = 90     ' 90 days till expiration
      Dim m_Rate As Double = 0.10          ' 10% interest rate
      Dim m_SharesA as Integer = 1000      ' 1000 shares of A total
      Dim m_SharesB as Integer = 2000      ' 2000 shares of B total
      Dim m_DivA as Double = 2.00 ' 2.00 dividend 40 days from now on A
      Dim m_DivB as Double = 1.00 ' 1.00 dividend 50 days from now on B
      Dim m_DaysDivAInvested as Integer = 50
                     ' ( 90 - 40 ) = 50 days to invest dividend
      Dim m_DaysDivBInvested as Integer = 40
                     ' ( 90 - 50 ) = 40 days to invest dividend
      Console.WriteLine("Please enter the price of stock A:")
      Dim m_PriceA as Double = Console.ReadLine
      Console.WriteLine("Please enter the price of stock B:")
      Dim m_PriceB As Double = Console.ReadLine
' Calculate the fair value and print it to the screen.
      m_IndexValue = (((m_PriceA * m_SharesA) + (m_PriceB * _
                      m_SharesB)) / ORIGINALVALUE) * 10

      m_FairValue = (m_IndexValue) * (1 + m_Rate * m_DaysTillExp _
                        / 360) - (m_DivA * (1 + m_Rate * _
                        m_DaysDivAInvested / 360) + m_DivB * (1 + _
                        m_Rate * m_DaysDivBInvested / 360))
```

```
        Console.WriteLine("The fair value is " & m_FairValue)
    End Sub
```

STEP 7 Run the program by selecting from the menu bar Debug > Start Without Debugging.


C:\My Documents\Book\BOOK\S2_VB.NET\Ch4-Value Types\Example\IndexFuturesB\bin\IndexFut...

```
Please enter the price of stock A:
100
Please enter the price of stock B:
75
The fair value is 1278.21111111111
Press any key to continue_
```

Although we are finished programming for the chapter, let's take a little more in depth look at the fair value of a futures contract on a stock index.

No-arbitrage conditions prevent the value of the index futures contract from moving too far away from the fair value. Cash-and-carry strategies prevent the futures price from getting too high relative to the cash stocks and reverse cash-and-carry arbitrage strategies prevent it from getting too low. Identifying opportunities for cash-and-carry arbitrage, however, necessitates the technological infrastructure to monitor the 500 stocks in real time and execute trades simultaneously. For this reason, these types of trading strategies are often referred to as "program trading," since they are computer generated.

In the following two examples illustrating index arbitrage, we assume that the prices of the underlying stocks A and B do not change over the 90 days, although the profit or loss does not in either case depend on the stock prices at expiration. Rather, the profit arises from a discrepancy between the futures price and its fair value on day 0. (Kolb, 1997).

The futures price must be equal to the cash index price plus the charges to carry the cash index forward to expiration. (Kolb, p. 71) The carrying charge is the interest lost by being long the underlying stocks. If the prices do not fall in line with the cost of carry, a trader attempt a cash-and-carry or reverse-cash-and-carry arbitrage.

4.9 Cash and Carry Arbitrage

A cash-and-carry arbitrage strategy involves selling the buying stock and selling the futures contract in a similar but opposite fashion (Kolb, p. 343). Here we replicate the index by weighting our portfolio with 3 parts stock B, $750, and 2 parts stock A, $500.

Time	Cash Market	Futures Market
0 days	Borrow $1250 for 90 days at 10%. Interest owed will be $ 31.25 Buy 5 shares of stock A at $100. Buy 10 shares of stock B at $75.	Sell 1 futures contract at 1285.00.
40 days	Receive $2.00 dividend on each share of stock A totaling $10. Invest proceeds for 50 days at 10%.	
50 days	Receive $1.00 dividend on each share of stock B totaling $10. Invest proceeds for 40 days at 10%.	
90 days	Sell 5 shares of stock A at $100. Sell 10 shares of stock B at $75. Receive total proceeds from invested dividends of $10.14 and $10.11. Total proceeds are $1270.25. Repay debt plus interest of $1281.25.	Buy 1 futures contract at fair value at expiration of 1250, which is the spot index value.

P and L	Loss: $11.00	Profit: $35.00
Total Profit of $35.00 – $11.00 = $24.00		

4.10 Reverse Cash and Carry Arbitrage

A reverse cash-and-carry arbitrage opportunity involves selling the underlying stock and buying the futures contract in a similar but opposite fashion (Kolb, p. 343).

Time	Cash Market	Futures Market
0 days	Sell 5 shares of stock A at $100. Sell 10 shares of stock B at $75. Invest proceeds of $1250 for 90 days at 10%. Interest earned will be $31.25.	Buy 1 futures contract at 1255.00.
40 days	Borrow $10.00 for 50 days at 10%. Pay dividend on stock A. Interest owed will be $0.14.	
50 days	Borrow $10.00 for 40 days at 10%. Pay dividend on stock B. Interest owed will be $0.11.	
90 days	Buy 5 shares of stock A back at $100. Buy 10 shares of stock B back at $75. Repay debt plus interest of $20.25. Receive interest of $31.25.	Sell 1 futures contract at fair value at expiration of 1250, which is the spot index value.
P and L	Profit: $11.00	Loss: $5.00.
Total Profit of $11.00 – $5.00 = $6.00		

4.11 Chapter Five Summary

In this chapter you have been exposed to all the different variable types available in .NET. Also, you should now understand how to declare variables using the Dim statement and the various identifiers and access modifiers as well as how to define them. Good programmers will also understand the importance of the Option Strict On, though for simplicity's sake we will neglect it in this book. Also, our variable naming convention require that we add a prefixes to our variable names that indicate the data type of the variable. Variable names should also describe the something about the nature of the value such as m_StockPrice.

Further, we looked at the different operators available to programmers in VB.NET and how some of them could be used in the financial markets. Our example consisted of calculating the cash value of a stock index and the fair value of a futures contract on that index.

In this chapter we learned how to use If..Then..Else statements, Select Case statements, and many different kinds of loops to control program flow. Loops will become more important in future chapters about arrays and data structures. We also looked at how to use a loop to forecast volatility using the GARCH(1,1) equation.

4.12 Chapter Problems

1. What is a variable and what is a constant?
2. When should you use Option Strict?
3. Write a line of code that would declare a variable to hold the value of the gamma of an option.
4. Write a line of code that would calculate the average of five daily returns known as m_MonReturn, m_TuesReturn, m_WedReturn, m_ThursReturn, m_FriReturn.
5. What is a concatenation operator? What is the value of a string variable known as strOptionSymbol, if strOptionSymbol = "INTC" & " " & "Sep" & " " & "50"?
6. What are the two types of structures discussed in this chapter?
7. Assume you bought stock in MMZR at 50, write an if…then…else structure to sell the stock if it goes up by 10% or down by 5%.
8. What is the difference between the following two loops:

```
        Do While x < 10
        x += 1
    Loop
```

and,

```
    Do
        x += 1
    Loop While x < 10
```

9. What are the different repetition structures available in VB.NET?
10. Take a look at the following piece of code:

```
    For x = 0 To 2
        For y = 0 To 3
            Console.WriteLine( x * y )
        Next y
    Next x
```

What would be printed out to the screen?

Project One

Create a VB.NET console application that accepts 5 daily closing stock prices from the user and calculates the mean and standard deviation of the stock's log returns. The formula for the log return is:

$$r_i = \ln\left(S_i / S_{i-1} \right)$$

Of course, the equations for mean and standard deviation are:

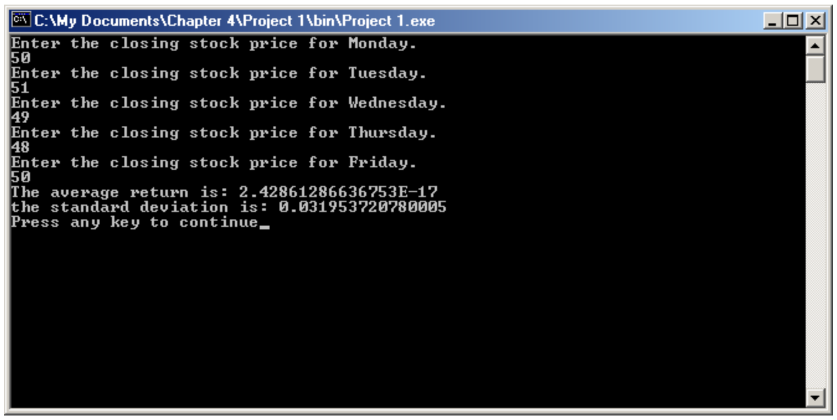
$$\mu_{r;t,T} = \frac{1}{n}\sum_{i=1}^{n} r_i \quad \text{and} \quad \sigma_{r;t,T} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2}$$

Since we won't cover functions until later in the book, so here's a hint. We can calculate the natural log using VB.NET's built-in log function.

```
m_TuesReturn = Math.log( m_TuesPrice / m_MonPrice )
```

Also, the square root can be found by raising the value to the .5 power using the ^ operator.

Be sure to name your variables using the naming conventions.



```
Enter the closing stock price for Monday.
50
Enter the closing stock price for Tuesday.
51
Enter the closing stock price for Wednesday.
49
Enter the closing stock price for Thursday.
48
Enter the closing stock price for Friday.
50
The average return is: 2.42861286636753E-17
the standard deviation is: 0.031953720780005
Press any key to continue_
```

Project Two

To calculate the value of the Dow Jones Industrial Average, a price weighted index, the equation is:

$$DJIA = \frac{\sum_{i=1}^{30} P_i}{Divisor}$$

Futures contracts on the DJIA trade at the Chicago Board of Trade.

Create a console application that calculates the fair value of a two stock, price weighted index according to this formula. Assume that the two stocks, A and B, are priced at 100 and 75 and pay dividends in the amounts and times shown in the chapter.

Project Three

The GARCH(1,1) equation forecasts volatility using two optimized coefficients, alpha and beta, and three values—an estimate of the previous day's variance, r^2, the long run variance, V, and the previous day's forecast, σ^2_t. The estimate of previous day's variance uses the log of the close-to-close method discussed in the chapter. However, as we saw, close-to-close may not be a good representation of intra-period volatility.

Create a VB.NET Windows application that calculates three forecasts for volatility for a user-defined number of days ahead. This time, make the GARCH(1,1) forecast using the close-to-close, the Parkinson, and Garman Klass estimators of one period volatility. Print out the three forecasts in three labels.

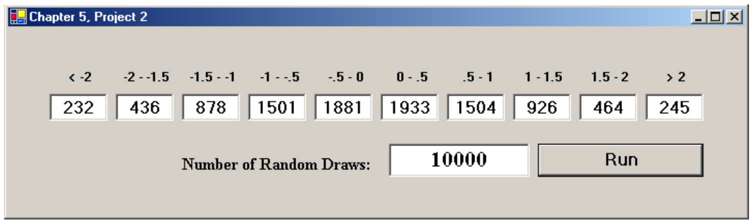
Project Four

Visual Basic.NET has a built-in random number generator, a Random object, which draws uniformly distributed deviates (random numbers) between 0 and 1 through its NextDouble method. In finance, we often wish to use a normal distribution for MonteCarlo simulation. Here is the code to generate a random number drawn from the standard normal distribution using the Random object.

VB

```
Dim m_NormRand as Double
Dim myRandom as Random
myRandom = New Random( DateTime.Now.Millisecond )
m_NormRand = myRandom.NextDouble() + myRandom.NextDouble() +
myRandom.NextDouble() + myRandom.NextDouble() + myRandom.NextDouble() +
myRandom.NextDouble() + myRandom.NextDouble() + myRandom.NextDouble() +
myRandom.NextDouble() + myRandom.NextDouble() + myRandom.NextDouble() +
myRandom.NextDouble() - 6
```

Create a VB.NET Windows application that will use a For…Next loop and a Select Case structure to generate a user-defined number of normally distributed random deviates and put the deviates into 10 bins as shown in the Select Case explanation in the chapter.



< -2	-2 - -1.5	-1.5 - -1	-1 - -.5	-.5 - 0	0 - .5	.5 - 1	1 - 1.5	1.5 - 2	> 2
232	436	878	1501	1881	1933	1504	926	464	245

Number of Random Draws: 10000