Chapter 5: Procedures

A procedure is a generic term that refers to the two types of routines--"sub" routines and functions. Procedures are packaged pieces of code that perform specific operations. Visual Basic.NET has hundreds of procedures that we can use in our programs to perform common tasks such as string manipulation, error checking and even a few mathematical and financial calculations. What's more, we can create our own, user-defined procedures to accomplish specific tasks in our programs.

When we call a procedure in our program, we are telling Visual Basic.NET to execute the code associated with that procedure. Furthermore, we may specify input arguments, or parameters, that we want to pass into the procedure; that is, the value or values we want the routine to work on. When we define a procedure, we must specify four things: a name for the procedure; a comma-separated list of parameters the procedure accepts, if any; the data type of the return value, if any; and the procedure definition, which is the code that executes when the routine is called.

The only difference between a subroutine and a function is that a function returns a value, aptly named the "return value" or "return argument," whereas a subroutine does not. A return value gets sent back from the function to the code that called it. In general, functions are preferred to subroutines and they will be used whenever possible. The distinction between functions and subroutines will become clear when we use them later.

We programmers use procedures to better organize code by breaking it up into smaller tasks. This makes the program code easier to read and debug. Also, procedures that perform common tasks can be called over and over from different sections of the program, reducing duplication of code and making the program easier to maintain. For example, if we wanted to calculate the mean returns for 100 stocks, we could write one function called Average(), and use it a hundred times over, rather than making the calculation in code for each of the 100 stocks. Let's look at the code for an Average() function.

VB

```
Public Function Average(ByVal m_Return1 As Double, _
                        ByVal m_Return2 As Double) As Double
            Return ( m_Return1 + m_Return2 ) / 2
End Function
```

C#

```
public static double Average(double m_Return1, double m_Return2)
{
      return (m_Return1 + m_Return2) / 2.0;
}
```

Let's review the four elements of a function. One, the name of this function is Average(). Two, this function accepts two input arguments, both of type Double, that will have the names m_Return1 and m_Return2 within the function definition. Three, this function returns a value of type Double. And, four, the function definition is the code between the function header, the Public Function Average line, and the function footer, End Function. We could call this function from somewhere else in our program this way:

VB

```
Sub Main()
      Dim m_Average as Double = Average( .015, .005 )
      Console.Writeline( m_Average )
End Sub
```

C#

```
static void Main(string[] args)
{
      double m_Average = Average(.015, .005);
      Console.WriteLine( m_Average );
}
```

Here the value of m_AverageReturn is set equal to the return value of the function Average(). Of course, this program prints out .01.

One way to describe a function is to think about a black box that processes input, much like a mathematical function. In algebra we may use an expression like this:

$$y = f(x_1, x_2, x_3)$$

$f(x)$ is, of course, a function. This function has a name, f. The function accepts input arguments, namely x_1, x_2 and x_3. The function named f has a return value to which y is then set equal. The definition of f exists somewhere else and is, say for example, $f(x_1, x_2, x_3) = 2x_1 + 3x_2 + 4x_3$. Functions in programming are no different.

5.1 Input Arguments

Both functions and subroutines can take input arguments. The input argument list, often called the parameters, has its own syntax that requires separate consideration.

We can declare as many input arguments with their respective data types as are needed, provided we separate each parameter with a comma. The basic syntax is to specify a local name for the value and a data type. For example, here is a simple subroutine that prints out two numbers in the console window:

VB

```
Private Sub PrintPrices(ByVal m_Price1 As Double, _
                        ByVal m_Price2 As Double)
      Console.Writeline( m_Price1 )
      Console.Writeline( m_Price2 )
End Sub
```

C#

```
public static void Print(double m_Price1, double m_Price2)
{
     Console.WriteLine(m_Price1);
     Console.WriteLine(m_Price2);
}
```

We then call the PrintNumbers subroutine, we specify the parameters after the name as follows:

VB

```
Sub Main()
      PrintPrices( 45.23, 65.54 )
End Sub
```

C#

```
static void Main(string[] args)
{
      PrintPrices( 45.23, 65.54 );
}
```

In this example, the values 45.23 and 65.54 are passed to the variables m_Price1 and m_Price2. Within the subroutine definition, the values passed in will be known by the local names m_Price1 and m_Price2. Since this is a subroutine, there is no return value as was the case in the Average() function. The output of this simple program will be:

45.23
65.54

There are times when we may not be required to pass all the arguments in a parameter list to a procedure. This is typically the case when parameters later in the list are dependent on specific values of variables earlier in the list. To declare a parameter as optional, we include the Optional keyword in the parameter declaration. When we declare a parameter as optional, all subsequent parameters in the list must also be optional. Here is an example:

VB

```
Public Function PV( ByVal Rate As Double, _
                    ByVal NPer As Double, _
                    ByVal Pmt As Double, _
                    Optional ByVal FV As Double, _
                    Optional ByVal Due as DateTime) As Double
```

C#

Not supported. Use array parameter, default parameter, or overloading.

Here, values for FVand Due are not required by the function definition to perform the calculation and return the PV, present value.

5.2 ByRef and ByVal

Let's take a look at the important distinction between ByRef and ByVal, the two methods for passing input arguments to functions.

Passing an input argument "ByVal" means that the original variable, which is being passed as an input argument, will not be changed by the function definition. That is to say, the procedure makes a copy of the value and performs the operations within the procedure definition on the copy, as opposed to the original variable. This is demonstrated by the following example.

VB

```
Sub Main()
      Dim m_StockPx as Double = 52.78
      Increment(m_StockPx)
      Console.WriteLine("Stock price after increment is: " & m_StockPx)
End Sub

Private Sub Increment( ByVal m_Val As Double)
      Console.WriteLine("Increment subroutine was passed: " & m_Val)
      m_Val += 1
      Console.WriteLine("New value is: " & m_Val)
End Sub
```

C#

```
static void Main(string[] args)
{
   double m_StockPx = 52.78;
   Increment(m_StockPx);
   Console.WriteLine("Stock price after increment is: " + m_StockPx);
}

public static void Increment( double m_Val )
{
   Console.WriteLine("Increment subroutine was passed: " + m_Val);
   m_Val += 1;
   Console.WriteLine("New value is: " + m_Val);
}
```

This program outputs:
Increment function was passed: 52.78
New value is: 53.78
Stock price after increment is: 52.78

The m_StockPrice variable is unaffected by the addition within Increment subroutine. This is because only the value of m_StockPrice has been passed to m_Val. m_Val is a completely separate variable. ByVal is the default method for passing values into functions. Now try this example again, but change ByVal in Increment() to ByRef as follows:

VB

```
Private Sub Increment( ByRef m_Val As Double )
```

C#

```
public static void Increment( ref double m_Val )
```

Back in the main function in C# use ref again to call the function,

```
Increment( ref m_StockPx );
```

This time the output is:
Increment was passed: 52.78
New value is: 53.78
Stock price after increment is: 53.78

Here, a reference to the location of m_StockPx in memory is passed to m_Val, not the value of m_StockPx. Therefore, as far as the computer is concerned, both m_Val and m_StockPx are referring to the same physical space, or location, in memory. Hence, when m_Val is incremented, the value of m_StockPx changes, since they are both the same variable.

5.3 ParamArray

There is one additional keyword we can use in procedure declarations--ParamArray. ParamArray enables us to pass an arbitrary number of arguments into function. That is, ParamArray allows an indeterminate number of input arguments passed as either one-dimensional list or as an array of the type specified. Within the function definition, the parameter array is treated as an array of its declared type. To use a ParamArray, just specify the last parameter in a parameter list as a ParamArray.

VB

```
Sub Main()
      Dim m_Prices As Double() = New Double() {52.34, 35.34, 0.15}
      PrintPrices(m_Prices)               ‘ Pass as an array
      PrintPrices(10.5, 95.34, 31.22, 74.23)  ‘ Pass as a list
End Sub

Private Sub PrintPrices(ByVal ParamArray m_StockPrices As Double())
      Dim i As Double
      Console.WriteLine("Porfolio of stocks contains " & _
            m_StockPrices.Length & " stocks. The prices are: ")
      For Each i In m_StockPrices
            Console.WriteLine(" " & i)
      Next i
End Sub
```

C#

```
Not supported.
```

This program calls the function twice, the first time the array, m_Prices, is passed with 3 prices; the second time a list of 4 prices is passed. In Chapter 8, we will take an in depth look at arrays. Also, notice the use of the For Each…Next loop structure as we discussed in the previous chapter.

5.4 Return Values

As said, functions have return values, which do not necessarily have to be numbers; they can return any data type. We can set the return value of a function by using the Return keyword. Here is a function that returns a Boolean, expressing whether or not our stock has hit a support level.

VB

```
Public Function SupportLevel( m_Price as Double, _
                              m_StrongSupport as Double ) As Boolean
      If m_Price > m_StrongSupport
            Return True
      Else
            Return False
      End If
End Function
```

C#

```
public static bool SupportLevel(double m_Price, double m_StrongSupport)
```

```
{
     if (m_Price > m_StrongSupport)
     {
          return true;
     }
     else
     {
          return false;
     }
}
```

Functions can return any value type, such as doubles, integers, Booleans, or strings. As we will learn in later chapters, functions can also return reference types like arrays and objects.

5.5 Black-Scholes Option Pricing Formula

In programming VB.NET, and all other languages for that matter, the process of creating our own user-defined procedures is exactly the same as in algebra. However, as you may have noticed we like to give our procedures and input variables more descriptive names than just *f* and *x*'s and *y*'s. Programmers prefer to use names like Command1_Click() or BlackScholesCall() that describe the nature of the operations performed within the procedure definition.

The Black-Scholes price of a call option is a function of several input values, namely S, the price of the underlying stock, X, the strike price, t, the time to expiration, r, the interest rate, and σ, the volatility, so that

$$y = \text{BlackScholesCall}(S, X, t, r, \sigma)$$

The mathematical definition of the Black-Scholes equation for the price of a call option is:

$$BlackScholesCall = SN(d_1) - Xe^{-rt}N(d_2)$$

where:

$$d_1 = \frac{\ln(S/X) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

and:

$$d_2 = d_1 - \sigma\sqrt{T}$$

To make a VB.NET function that calculates the price of a call option according to the Black-Scholes formula, we need four things: a function name; a list of input arguments with their respective data types; a return type; and a function definition.

VB

```
Public Function BlackScholesCall(ByVal m_Stock As Double, _
                                 ByVal m_Strike As Double, _
                                 ByVal m_Time As Double, _
                                 ByVal m_Rate As Double, _
                                 ByVal m_Sigma As Double) _
                                 As Double
      Dim d1, d2, Nd1, Nd2 As Double
      ' Calculate d1 and d2
      d1 = (Math.Log(m_Stock / m_Strike) + (m_Rate + _
            (m_Sigma ^ 2) / 2) * m_Time) / _
            (m_Sigma * Math.Sqrt(m_Time))
      d2 = d1 - m_Sigma * Math.Sqrt(m_Time)
      ' Calculate N(d1) and N(d2)
      Nd1 = NormCDF(d1)
      Nd2 = NormCDF(d2)
      ' Calculate the price of the call
      Return m_Stock * Nd1 - m_Strike * Math.Exp(-m_Rate _
            * m_Time) * Nd2
```

```
End Function
```

C#

```
public static double BlackScholesCall(double m_Stock, double m_Strike,
                                      double m_Time, double m_Rate,
                                      double m_Sigma)
{
  double d1, d2, Nd1, Nd2;
  // Calculate d1 and d2
  d1 = (Math.Log(m_Stock / m_Strike) + (m_Rate +
       (Math.Pow(m_Sigma, 2 )) / 2) * m_Time) /
       (m_Sigma * Math.Sqrt(m_Time));
  d2 = d1 - m_Sigma * Math.Sqrt(m_Time);
  // Calculate N(d1) and N(d2)
  Nd1 = NormCDF(d1);
  Nd2 = NormCDF(d2);
  // Calculate the price of the call
  return m_Stock * Nd1 - m_Strike * Math.Exp(-m_Rate * m_Time) * Nd2;
}
```

The code that calls the BlackScholesCall() function then doesn't need to know how the function calculates the result. It just takes the output it needs and goes on its merry way. This definition of the function will be somewhere else. We could call the function in this fashion:

VB

```
Sub Main()
      Dim m_OptionPrice as Double
      m_OptionPrice = BlackScholesCall( 42, 40, .5, .1, .2 )
      Console.Writeline(m_OptionPrice)
End Sub
```

C#

```
static void Main(string[] args)
{
     double m_OptionPrice;
     m_OptionPrice = BlackScholesCall( 42, 40, .5, .1, .2 );
     Console.Writeline(m_OptionPrice);
}
```

The variable m_OptionPrice then will be set equal to the return value of the function called BlackScholesCall(), which of course calculates the price of a call option according to the parameters, or input arguments, it receives.

The Black-Scholes formula is just one of several methods to calculate the price of an option. We will not, however, cover option pricing theory in depth in this book although we will briefly cover binomial trees in Chapter 8. We refer you to one of several other books on the topic, especially *The Complete Guide to Option Pricing Formulas* by Espen Gaarder Haug (McGraw Hill, 1998), which contains particularly complete coverage of option pricing methods.

Let's create a short Windows application that calculates the price of a call option using the BlackScholesCall() function.

Step 1 Open a new Windows Application in Visual Basic.NET and name it BlackScholes.

Step 2 Once the IDE for your new program is ready, in the Project menu, click on Add Module to add a code module.

Step 3 In the module, type in the BlackScholesCall() function code as shown previously or copy it from the file named BlackScholesCall.txt from the CD and paste it in. Your module should look like this:

VB

```
Module BlackScholes
      Public Function BlackScholesCall(ByVal m_Stock …)  As Double
                  ' Function definition in here.
```

```
      End Function
End Module
```

Step 4 Since the BlackScholesCall() function itself, calls another function called NormCDF() we will have to add this function which is an approximation of the cumulative normal distribution function. Again, in the Project menu, click on Add Module. Add the following code to the this new module:

VB

```
Module NormalCDF
  Public Function NormCDF(ByVal X As Double) As Double
      ' Calculate the cumulative probability distribution
      ' function for standard normal at X
      Dim a, b, c, d, prob As Double
      a = 0.4361836
      b = -0.1201676
      c = 0.937298
      d = 1 / (1 + 0.33267 * Math.Abs(X))
      prob = 1 - 1 / Math.Sqrt(2 * 3.1415926) * Math.Exp(-0.5 * X * _
            X) * (a * d + b * d * d + c * d * d * d)
      If X < 0 Then prob = 1 - prob
      Return prob
  End Function
End Module
```

C#

```
public static double NormCDF(double X)
{
     // Calculate the cumulative probability distribution
     // function for standard normal at X
     double a, b, c, d, prob;
     a = 0.4361836;
     b = -0.1201676;
     c = 0.937298;
     d = 1 / (1 + 0.33267 * Math.Abs(X));
     prob = 1 - 1 / Math.Sqrt(2 * 3.1415926) * Math.Exp(-0.5 * X * X) *
            (a * d + b * d * d + c * d * d * d);
     if ( X < 0 )
         prob = 1 - prob;
     return prob;
}
```

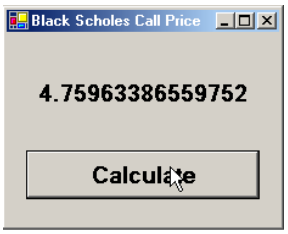
Step 5 At the top of your code window, click on the Form1.vb [Design] tab to return to the GUI development window. From the Toolbox, add a label, named Label1, and a command button, named Button1, to your form.

Step 6 Double click on the command button to bring up the code stub for the Button1_Click event. To this event subroutine, add the following code.

VB

```
Private Sub Button1_Click(ByVal sender …) Handles Button1.Click
        Dim m_CallPrice As Double
        m_CallPrice = BlackScholesCall(42, 40, 0.5, 0.1, 0.2)
        Label1.Text = m_CallPrice
End Sub
```

Step 7 Run the program.



The program you have just created illustrates the use of two functions, BlackScholesCall() and NormCDF(), and a subroutine, Button1_Click(). Again, notice that the two functions accept input arguments and have a return value whereas the subroutine does not have a return value.

5.6 Math Class

If you program in Excel, you should be well-versed in prebuilt mathematical functions. .NET too has numerous built-in mathematical functions that we can call in our programs. The following table summarizes the available functions found in the Math namespace that may be important in quantitative finance. To call these functions, we need to precede the function name with the class name Math and a dot (.). That is, the fully qualified function name for the Max() function, for example, is Math.Max().

Math Class Functions	Description	Example
Abs()	Returns the absolute value of x.	m_Error = Math.abs(m_Forecast – m_Actual)
Ceiling()	Returns the integer greater than or equal to the input argument.	m_Sell = Math.Ceiling(m_StockPrice)
Exp()	Returns e (the base of natural logarithms) raised to the power of x.	m_FV = m_PV * Math.Exp(m_R * m_Time)
Floor()	Returns the integer less than or equal to the input argument.	m_Buy = Math.Floor(m_StockPrice)
Log()	Returns the natural logarithm of x.	m_Rate = Math.log (m_Tuesday / m_Monday)
Max()	Returns the maximum of two input arguments.	m_Price = Math.Max (0, x)
Min()	Returns the minimum of two input arguments.	m_Price = Math.Min (0, x)
Sign()	Returns: 1 if x is greater than 0; 0 if x equals 0; -1 if x is less than 0.	m_BUYSELL = Math.Sign (m_Moonphase)
Sqrt()	Returns the square root of x.	m_StDev = Math.Sqrt(m_Variance)

5.7 String Class

A string is often times just a thing that we print out or pass from one part of our program to another without regard for its contents. Other times, however, we need to know what's inside. We might need to verify its contents, modify it in some way, or extract a specific piece of information from it. When dealing with options quotes, for example, we sometimes need to parse out the stock symbol, the expiration month and the strike price, which are all strung together in one long option symbol. The String class functions summarized below.

String Class Functions	Description	Example

Concat()	Concatenates an array of strings into a delimited string.	m_String = String.Concat("Buy low,", " Sell high.")
Length()	Returns the integer length of String.	m_Int = m_String.Length()
Substring()	Returns a string starting at the position designated first and continuing to the second parameter number of characters.	m_SubString = m_String.Substring(4, 2)
Split()	Returns an array of strings consisting of the delimited strings (or words) of an input argument string.	m_ArrayString = m_String.Split(",")
Comp()	Returns -1, 0, or 1, depending upon the result of a string comparison.	m_Bool = String.Compare(stringA,stringB)

Many of these functions are helpful in parsing strings. Parsing is the process of extracting smaller pieces or substrings from a string. Here are some examples showing how to parse strings using the string functions in the table.

5.8 The Substring Functions

The substring functions accept two input arguments—a string and a length.

5.9 Formatting Numbers for Output

The ToString() function enables us to converts and formats dates and numbers into strings. It gives us a much greater degree of control over how our data is presented for either screen or printer output. In .NET, we can choose from predefined named formats or create our own user-defined format for finer control.

VB

```
Console.WriteLine( Math.PI.ToString("##.####") )
```

C#

```
Console.WriteLine( Math.PI.ToString("##.####") );
```

The backslash allows the computer to interpret the next character literally instead of as a format character. Here is a quick code example that will print out the value of m_Volatility as 0.1235.

VB

```
Sub Main()
      Dim m_Volatility as Double = 0.1234567
      Console.WriteLine( m_Volatility.ToString( "0.####" ))
End Sub
```

5.10 Conversion Functions

In VB.NET Option Strict On requires explicit conversion of data types in cases where data loss could occur. This includes any conversion between numeric types and string types. For example, data loss may occur when a string variable is converted to a double or any other data type with less precision or smaller capacity. If Option Strict is On, an error will occur if an implicit conversion exists in our program. VB.NET provides the Convert class for explicit conversion. Also, as we mentioned earlier, in order to clarify and simplify the algorithms and logic in our example programs, we have almost always left (in VB) Option Strict by default Off. However, production applications you create should include Option Strict On and explicit type conversions through the use of these conversion functions.

As you can probably imagine, not all conversions are possible. We clearly cannot convert "IBMDP" into a double. Here is a short program illustrating the use of a conversion function.

VB

```
Option Strict On
Dim m_Int as Integer = Convert.ToInt32( m_String )
```

C#

```
int m_Int = Conver.ToInt32( m_String );
```

5.11 Validation Functions

.NET's validation functions allow us to check the data type of a value before we perform an operation. You may have noticed in previous programs that errors occur if the user enters a bad value. If, for example, instead of entering 1000, a number to be used in a calculation, the user enters XYZ, the program will end with an error since the calculation requires a number not a string. Validation functions allow us to check first to see that the user inputted values are correct. If not we can prompt the user with a message box to reenter the values properly.

Validation Function	Description	Example
IsInfinity()	Returns true or false indicating whether a value is infinity.	myBool = Double.IsInfinity(m_Double)
IsNaN()	Returns true or false indicating whether a value is a number.	myBool = Double.IsNaN(m_Double)

Here is a short program that will keep prompting the user to enter a valid numeric value until it gets it.

VB

```
Sub Main()
      Do While 1
            Console.WriteLine("Please enter a volatility: ")
            If Not Double.IsNaN(Console.ReadLine()) Then
                Console.WriteLine("Thank you for the valid input.")
                Exit Do
            Else
                MessageBox.Show("Please enter a valid value.")
            End If
      Loop
End Sub
```

5.12 DateTime Structure

.NET provides a wealth of date functions as part of the DateTime structure. These can be used to manipulate dates, which, as you can probably imagine, become very valuable in modeling fixed income securities, futures and options. Here are several of the date functions:

DateTime Functions	Description	Example
AddDays		
AddMilliseconds		
Now	Returns the current date and time from the computer's built-in clock	

This program calculates the number of days between these two dates, 106. We can convert calendar days to trading days using the formula:

5.13 MessageBox Class

The MesssageBox.Show procedure displays a dialog box with a message, an OK button, an optional icon, and a title. MessageBox can also return the value of the button pressed by the user.

The title parameter is simply the text that appears across the title bar of the message box. This defaults to your application's name. The MessageBox.Show function can have one, two, or three buttons. The function returns the value of which button your user pressed. Before we talk about these values, however, we need to take a quick detour and talk about .NET's predefined constants.

The following is an example call to MessageBox.Show:

VB

```
Sub Main()
     Dim myResponse As MsgBoxResult
     m_Response = MessageBox.Show("Continue?", vbYesNo + vbQuestion, _
                  "Continue")
End Sub
```

Or more simply,

VB

```
MessageBox.Show("Please enter valid data.", , "Option Calculator")
```

5.14 Random Class

The Random object's Next Double() method in VB.NET returns a random number from a uniform distribution between 0 and 1. For example,

VB

```
Sub Main()
      Dim m_NormRand as Double
      Dim m_Random as Random
      myRandom = New Random( DateTime.Now.Millisecond )
      m_NormRand = m_Random.NextDouble()
      Console.WriteLine(m_NormRnd)
End Sub
```

Several mathematical methods have been developed for accomplishing the task of generating standard normal deviates, including the well-known rejection method with Box-Muller transformation, which can be converted into program code. However, we prefer a much simpler method, as shown here:

VB

```
   Function StdNormRnd() as Double
         Dim m_NormRand as Double
         Dim m_Random as Random
         m_NormRand = m_Random.NextDouble() + m_Random.NextDouble() +
          m_Random.NextDouble() + m_Random.NextDouble() +
          m_Random.NextDouble() + m_Random.NextDouble() +
          m_Random.NextDouble() + m_Random.NextDouble() +
          m_Random.NextDouble() + m_Random.NextDouble() - 6
   End Function
```

In all cases, this method will suffice and the StdNormRand() function above will be used in this book. Here are some other functions for random numbers from distributions other than the standard normal. A normal distribution with mean and standard deviation,

VB

```
Function NormRnd( m_Mean As Double, m_StdDev As Double ) As Double
      Return StdNormRnd() * m_StdDev + m_Mean
End Function
```

And, the lognormal.

VB

```
Function LogNormalRnd(m_Mean As Double, m_StdDev As Double) As Double
      Return Exp(m_Mean + m_StdDev * StdNormRnd())
End Function
```

5.15 Implied Volatility

Most often in financial markets, we are interested in calculating the volatility implied by an option's price as opposed to the price itself, since the price can be observed in the market. Rather than passing the stock

price, the strike, time, interest rate and volatility into a function to get the price, we would rather pass the option price, the stock price, the strike, time, and interest rate into a function and get the volatility.

Analyzing and forecasting volatility is an important facet of automated derivatives trading. We have looked at some ways of forecasting volatility based upon estimates of past, or historical, volatility. If the implied volatility of an option, as observed from its market price, deviates substantially from our forecast of volatility between now and expiration, there may be a trading opportunity. That is, if the implied volatility is substantially higher than our forecast, we may consider selling the option. Alternatively, if the implied volatility is substantially lower than our forecast, we may consider buying the option.

Let's augment the program we started earlier in this chapter to calculate the implied volatility of a call option given an options symbol, a price for the underlying stock and the price of the call option. We will use some of the string manipulation functions to determine the month of expiration and the strike price from an option symbol according to the following tables:

Expiration Month Codes												
	Jan	**Feb**	**Mar**	**Apr**	**May**	**Jun**	**Jul**	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**
Calls	A	B	C	D	E	F	G	H	I	J	K	L
Puts	M	N	O	P	Q	R	S	T	U	V	W	X

Strike Price Codes (Abbr.)												
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
5	10	15	20	25	30	35	40	45	50	55	60	65
N	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
70	75	80	85	90	95	100	7.5	12.5	17.5	22.5	27.5	32.5

Step 8 Add three more modules to your program to hold the TimeTillExp(), StrikePrice() and ImpliedVolatilityCall() functions. Type in the function definitions for the three functions as follows. Alternatively, you can copy and paste in the code from the CD. Here is the code for the TimeTillExp() function:

VB

```
Module ExpirationTime
    Public Function TimeTillExp(ByVal strOptionSym As String) As Double
      ' Find the second to last character in the string.
Dim strMonthStrike As String = Right(strOptionSym, 2)
Dim chrMonth As Char = Left(strMonthStrike, 1)
Dim strMonth As String
        Dim dtExpDate As Date
        Select Case chrMonth
            Case "A", "a", "M", "m"  ' Use a Select...Case structure to
                strMonth = "January" ' transform the month character
            Case "B", "b", "N", "n"  ' into the appropriate string.
                strMonth = "February"
            Case "C", "c", "O", "o"
                strMonth = "March"
            Case "D", "d", "P", "p"
                strMonth = "April"
            Case "E", "e", "Q", "q"
                strMonth = "May"
            Case "F", "f", "R", "r"
                strMonth = "June"
            Case "G", "g", "S", "s"
                strMonth = "July"
            Case "H", "h", "T", "t"
                strMonth = "August"
            Case "I", "i", "U", "u"
                strMonth = "September"
            Case "J", "j", "V", "v"
```

```
                strMonth = "October"
            Case "K", "k", "W", "w"    ' Assume all options expire on
                strMonth = "November"  ' the 15th of the month.  If the
            Case "L", "l", "X", "x"    ' date has passed for the
                strMonth = "December"  ' current year, find the date
End Select                          ' for the following year.
      dtExpDate = DateValue(strMonth & " 15, " & Year(Today()))
      If Today() > dtExpDate Then _
  dtExpDate = DateValue(strMonth & " 15, " & (Year(Today()) + 1))
      Return (DateDiff(DateInterval.Day, Today(), dtExpDate)) / 365
    End Function
End Module
```

Notice that the TimeTillExp() function makes use of several functions we have looked at in this chapter, including Right() and Left() to find the second to last character in the option symbol string, DateValue() to convert a string representation of a date into a variable of data type Date, Year() to determine the year corresponding to the date returned by Today(), and DateDiff() to calculate the number of days between Today() and the expiration date, assuming options always expire on the 15th of the month, which simplifies this example. Here is the code for the ImpliedVolatilityCall() function.

VB

```
Module ImpliedVol
Public Function ImpliedVolatilityCall(ByVal m_MarketPrice As Double,
                                      ByVal m_Stock As Double, _
                                      ByVal m_Strike As Double, _
                                      ByVal m_Time As Double, _
                                      ByVal m_InterestRate As Double) _
                                      As Double
        Dim ImpliedVol, LowVol, HighVol, epsilon, mu, _
            TheoreticalPrice, PreviousPrice As Double
        HighVol = 10
        ImpliedVol = HighVol
        TheoreticalPrice = BlackScholesCall(m_Stock, m_Strike, _
                                m_Time, m_InterestRate, HighVol)
        epsilon = TheoreticalPrice - m_MarketPrice
        mu = TheoreticalPrice - PreviousPrice
        Do While (Math.Abs(epsilon) > 0.0000001)
            If Math.Abs(mu) < 0.0000001 Then Exit Do
            If epsilon > 0 Then
                ImpliedVol = HighVol
                HighVol = HighVol - (HighVol - LowVol) / 2
            Else
                LowVol = HighVol
                HighVol = LowVol + (ImpliedVol - LowVol) / 2
            End If
            PreviousPrice = TheoreticalPrice
            TheoreticalPrice = BlackScholesCall(m_Stock, m_Strike, _
m_Time, m_InterestRate, HighVol)
            epsilon = TheoreticalPrice - m_MarketPrice
            mu = TheoreticalPrice - PreviousPrice
        Loop
        Return HighVol
    End Function
End Module
```

And finally, here is the code for the StrikePrice() function.

VB

```
Module Strike
    Public Function StrikePrice(ByVal strOptionSym As String) As Double
        Dim chrStrike = Right(strOptionSym, 1)
        Select Case chrStrike
            Case "A", "a"
                Return 5
            Case "B", "b"
                Return 10
            Case "C", "c"
                Return 15
            Case "D", "d"
                Return 20
            Case "E", "e"
                Return 25
            Case "F", "f"
                Return 30
            Case "G", "g"
                Return 35
            Case "H", "h"
                Return 40
            Case "I", "i"
                Return 45
            Case "J", "j"
                Return 50
            Case "K", "k"
                Return 55
            Case "L", "l"
                Return 60
            Case "M", "m"
                Return 65
            Case "N", "n"
                Return 70
            Case "O", "o"
                Return 75
            Case "P", "p"
                Return 80
            Case "Q", "q"
                Return 85
            Case "R", "r"
                Return 90
            Case "S", "s"
                Return 95
            Case "T", "t"
                Return 100
            Case "U", "u"
                Return 7.5
            Case "V", "v"
                Return 12.5
            Case "W", "w"
                Return 17.5
            Case "X", "x"
                Return 22.5
            Case "Y", "y"
                Return 27.5
            Case "Z", "z"
                Return 32.5
        End Select
```

```
    End Function
End Module
```

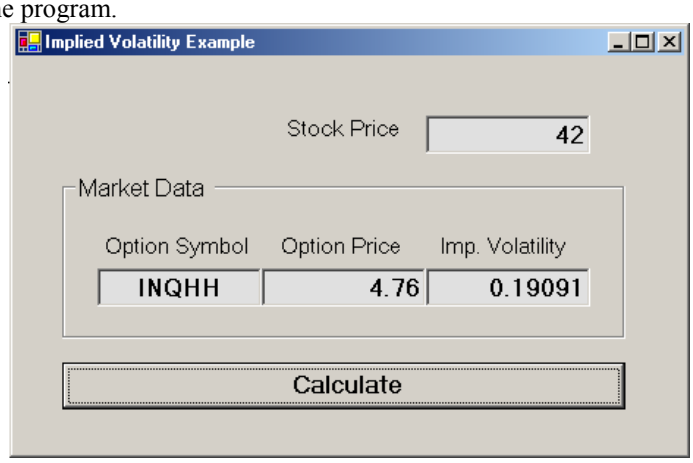
Step 9 On your form, place four textboxes named txtStockPrice, txtOptionSymbol, txtOptionPrice, and txtImpliedVol. In the Button1_Click event, change the code to the following:

VB

```
Private Sub Button1_Click(ByVal sender …) Handles Button1.Click
        Dim m_ImpVol As Double
        Dim strOptionSymbol$ = txtOptionSymbol.Text
        Dim m_StockPrice# = txtStockPrice.Text
        Dim m_OptionPrice# = txtOptionPrice.Text
        Dim m_TimeTillExp# = TimeTillExp(strOptionSymbol)
        Dim m_Strike# = StrikePrice(strOptionSymbol)
        Dim m_Rate# = 0.1
        m_ImpVol = ImpliedVolatilityCall(m_OptionPrice, _
 m_StockPrice, m_Strike, m_TimeTillExp, m_Rate)
        txtImpliedVol.Text = Format(m_ImpVol, "0.#####")
End Sub
```

Notice that our code employs several function calls to our user defined functions as well as to the Format() function.

Step 10 Run the program.



The results you obtain will be different than the one shown here since the time to expiration is always changing. However, if you pick an expiration around six months in the future, a strike price of 40, a stock price of 42 and an option price of 4.76, your implied volatility should be around 20%.

5.16 Chapter Five Summary

A procedure is a piece of code that performs a specific task. Functions return a value to the calling statement. Subroutines are exactly the same as functions except that they do not return a value. In general, functions are preferred.

Procedures are the building blocks of VB.NET programs. Modularizing our code into separate procedures, or blocks of code, enables reusability and cuts down on errors and debugging time.

5.17 Chapter Problems

1.) What is a subroutine? What is a function? What is the difference between a subroutine and a function?

2.) Write a line of code that calculates the number of days between January 7, 2002 and November 9, 2002 and assigns the value to a variable named intNumDays.

3.) What function would we use to find the date 37 days from today?

4.) Write a line of code that assigns the value of the log of 1.05 to a variable named m_Return.
5.) What function could we use to make sure that a user-entered value is actually a number?
6.) How could we print out a randomly drawn number from the standard normal distribution to five decimal places?

Project One

Create a Visual Basic.NET Windows application that calculates the price and greeks of a call option using the BlackScholesCall() function and the functions for the Greeks found on the CD included with this book. Allow the user to input an option symbol and parse it as in the chapter example.

The project should allow the user to input the stock price and the volatility. You can simply set the value of the interest rate in your code. Your program should calculate the other two input arguments necessary to calculate the prices and greeks of an option--the expiration and the strike--from the option symbol using the functions discussed in the chapter.

Project Two

The lognormal distribution assumes that the natural logarithm of the price-relative from time t to t+h is drawn from a normal distribution with mean, μ, and standard deviation, σ. The volatility of a stock then is the sample standard deviation of the logs of the price-relatives.

To simulate the price path of a stock, we need to first draw a random number, Z, from the standard normal distribution. Using the following equation, we can then derive a random stock price at time t+h.

$$S_{t+h} = S_t e^{(\mu(h)+\sigma Z\sqrt{h})}$$

Create a VB program that will allow the user to enter the initial stock price, the mean and standard deviation, and the change in time, h, and will generate a random series of 10 successive prices so that each new price is depends on the previous one. Remember that Volatility is an annualized number based on 256 trading days, so a change in time of one day would be 1/256 = .0039. Be sure to include this in your calculations. Also, use the Format() function so that your random prices print out in a readable fashion. Try using the MsgBox and the IsNumeric() function to validate user inputs.