Chapter 6: Objects

Thus far, we have looked at procedural programming. Event driven programming focuses on the use of events, such as among others button clicks and form loads, to control the execution of code. In event driven programming, different procedures run when different events happen. For the remainder of the book, we will use things called objects and object oriented programming (OOP) although we will still use events to illustrate code execution. OOP focuses on the use of objects to control program flow. As you will see, OOP enables us to perform very large and complex tasks with just a few lines of code.

6.1 Objects and Classes

In previous chapter, we have already looked at several classes and objects in our programs. The buttons that we put on our forms are objects. The button objects, known by default names like Button1 are actually instances of the button class. So, we say that an object is an instance of a class. Microsoft's .NET Framework gives us hundreds of premade classes, like buttons and textboxes as well as other non-visible classes, that we can instantiate and use in our programs.

All .NET languages are object oriented programming languages and as such use what are called "reference types" to encapsulate things. These things, called classes, have both data and functionality tied together within their definitions. Classes have "member variables" that store data and functionalities, or behaviors, held in procedures known as "methods" or "member functions." Classes may also have events associated with them in the way a button has a click event. In a working program, different objects, which again are instances of classes, work together, or talk to each other, through their respective public interfaces. That is to say, private data within an object, which is not accessible from the outside world, is available to the outside programming environment through the object's public interface.

For example, your name is a private piece of data about you. No one can know your name unless they interact with your public interface, your ears and your voice. They can get the value of your name by asking you what your name is and then you will tell them the value of your private name data using your public voice interface. To extend the analogy, when you were born, your parents set the value of your name much like we can set the text property of a button at design time. If you wanted to change, or set, your name during your lifetime, that is at runtime, you would say You.Name = "Gordon Gekko."

In .NET, we can create our own user-defined classes and create objects based on them. For example, we could create a class called StockOption. In a program, an IBM April 80 Call would be an object, that is, an instance of the StockOption class.

The organization of a class can be difficult to envision to programmers not used to thinking in terms of classes and objects. Here are the elements that make up a class.

Member Variables	**Description**
Variable	Simple data.
Constant	Read only values shared by all objects in the class.
Nested Types	Other types—classes, structures, interfaces, enums, etc.
Properties	**Description**
Property	Values of member variables are defined and retrieved through public Get and Set methods of a property.
Member Functions	**Description**
Methods	Procedure that provides the object with functionality.
Constructor	New() method runs when an object is instantiated.
Finalization	Method that runs just before an object is destroyed.
Events	**Description**
Event	Message sent from an event source to listener objects, called an event receiver.

In order to use OOP in .NET, we need to understand four main concepts of object oriented programming: abstraction, encapsulation, inheritance, and polymorphism.

6.2 Abstraction

Abstraction is the process of creating an abstract model of a real world object or thing. The process consists of taking the attributes, or properties, and functionalities, or methods, of an object and turning them into logical pieces of data and functionality.

Again, lets look at a stock option. To turn a stock option into a class in .NET, we need to think about the properties of a stock option; that is, what are the nouns associated with a stock option, like the option symbol, the strike price and the expiration date, as well as the verbs, or functionalities, or behaviours, of a stock option, like calculating implied volatility or calculating and returning the price. When we come up with a list of nouns, the "what it is" of an object, and verbs, the "what it does," we say that the object has been abstracted. So, let's assume for a minute we have fully abstracted a StockOption class into the following nouns and verbs.

Nouns	**Description**
Option symbol	The option symbol consists of a symbol for the underlying, a symbol for the month and a symbol for the strike price.
Expiration month	Derived from the option symbol.
Strike price	Derived from the option symbol.
Underlying symbol	Derived from the option symbol.
Price of the option	We will use the Black Scholes model to set the price.
Market price	The options price observed in the marketplace.
Volatility of the option	We will need to set the volatility.
Interest rate	We will need to set the interest rate.
Greeks	We will need to calculate the greeks.
Time till expiration	Calculated from the expiration month.
Days till expiration	Calculated form the expiration month.
Implied volatility	Calculated form the market price
Verbs	**Description**
Derive expiration month	Symbol for month is found in the option symbol.
Derive strike price	Symbol for strike price is found in the option symbol.
Derive underlying symbol	Symbol for the underlying is found in the option symbol.
Calculate price	Need a procedure to calculate Black Scholes price.
Calculate greeks	Need procedures to calculate the greeks.
Calculate trading days and time till expiration	Need a procedure to calculate the days and time till expiration using trading days.
Calculate the implied vol.	Need a procedure to calculate the implied volatility.

6.3 Encapsulation

Encapsulation refers to the process of containing the abstracted properties and methods into a class, exposing to the outside world only those methods that absolutely must be exposed which are then known collectively as the class' public interface. So, classes hide the implementation of their properties and methods and communicate with the external programming environment through the public interface. In this way, encapsulation protects the object from being tampered with and frees the programmer from having to know the details of the object's implementation.

In our StockOption example, the outside programming environment does not need to be exposed to the method of calculating the price, so this functionality is encapsulated and made unavailable to the outside world. This idea will become clearer as we go along. For right now, let's look at the code to encapsulate just the private variable named strOptionSym to hold the option symbol in the StockOption class, along with a public property called Symbol to get the value of the m_OptionSym.

Let's create a StockOption class.

Step 1 Open a new Windows application named OptionObject and add a single label, named Label1, to Form1.

Step 2 Under the Project menu item, select Add Class. A new class code window will appear.

Step 3 Add the following code to the StockOption class.

VB

```
Public Class StockOption
```

```
        Private m_OptionSym as String
        Public Sub New(ByVal m_Symbol As String)
                m_OptionSym = m_Symbol
        End Sub
        Public ReadOnly Property Symbol() As String
                Get
                        Return m_OptionSym
                End Get
        End Property
End Class
```

C#

```
class StockOption
{
     private String m_OptionSym;
     public StockOption(String m_Symbol)
     {
          m_OptionSym = m_Symbol;
     }
     public String Symbol
     {
          get
          {
               return m_OptionSym;
          }
     }
}
```

Notice that the class name is StockOption. Be careful. StockOption is a class, not an object. In this example, strOptionSym is private, and so we will not be able to get or set the value of it from outside the object itself. We can, however, set the value of m_OptionSym through the constructor method known as the New() subroutine.

So, New() is called the constructor method. Any time an object is instantiated, or born, using the New keyword, the object's constructor method executes. In this case the public subroutine New() accepts a string and sets the value of strOptionSym, our private member variable, equal to it. By requiring that an option symbol be passed to the constructor method, we prevent ourselves, or any other programmer using this class, from creating a new option object without a symbol.

Also, notice that we can get the value of m_OptionSym through the public property Symbol, which has a Get method within it. Public properties provide us with access to private member variables through Get and Set methods. Notice, however, that our Symbol property is ReadOnly, implying that once the strOptionSym member variable is set via the New() method, it cannot be changed.

Creating a reference type, such as an object, out of a class is a two-stage process. First, we declare the name of the object, which will actually then be a variable that holds a reference to the location of the object in memory. Second, we create an instance of a class using the New keyword. This is when the constructor method will run. Here is an example of showing the two-stage process:

VB

```
Dim myOption as StockOption
myOption = New StockOption("IBMDP")
```

C#

```
StockOption m_StockOption;
m_StockOption = new StockOption("IBMDP");
```

Alternatively, we can accomplish the process using one line of code:

VB

```
Dim myOption as New StockOption("IBMDP")
```

C#

```
StockOption m_StockOption = new StockOption( "IBMDP" );
```

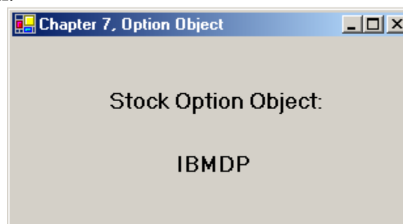
In different situations it will be advantageous to use one of these two methods. We will use both methods over the course of the book. As with variables, it is important to pay close attention to the scope of your reference types, which will dictate in many cases the method of instantiation.
In the Form1 code window, add the following code to the Form1_Load event:

VB

```
Private Sub Form1_Load(ByVal sender As … ) Handles MyBase.Load
    Dim myOption as New StockOption("IBMDP")
    Label1.Text = myOption.Symbol
End Sub
```

Now, when the program is running, myOption is the object, whereas StockOption is the class. We set the value of strOptionSymbol by passing the a string into the constructor, New(), as shown.

Step 4 Run the program.



The content of this program is not earth shattering of course, but congratulate yourself nonetheless; you have just created your first class, your first object and your first object- oriented program.

Of course, a stock option consists of a lot more data and functionality than just a symbol. Also, as we saw in our abstraction of a stock option, some of this other data might not be set from the outside, but rather calculated internally. For example, we would obviously prefer to have the option object derive the strike price internally from the option symbol rather than require that we set it explicitly from the outside. Let's take a look at the fully developed StockOption class found on the CD.

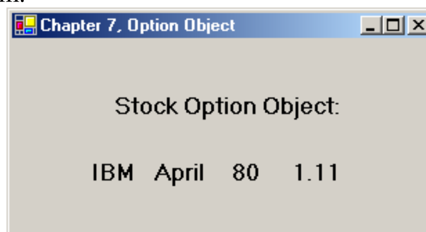
Step 5 Clear the StockOption class of the previous definition and paste in the full StockOption class code from the StockOption.txt file found on the CD.

Step 6 Add three labels to your form and change the Form_Load event code to:

VB

```
Private Sub Form1_Load(ByVal sender As … ) Handles MyBase.Load
    Dim MyOption As StockOption = New StockOption("IBMDP")
    Label1.Text = MyOption.Underlying
    Label2.Text = MyOption.ExpMonth
    Label3.Text = MyOption.Strike
    Label4.Text = MyOption.BSPrice
End Sub
```

Step 7 Run the program.



Once we have completely turned our model into computer code, we say that the class has been encapsulated. A major benefit of OOP is that because the data and methods encapsulated in classes are so closely tied together, we do not need to pass arguments back and forth as inputs to procedures. Rather, member functions can access member variables directly within their definitions. In the StockOption class code, notice that the member methods, such as SetStrikePrice, are able to access the member variables directly. Also, notice that the BlackScholesPrice() method, which contains a method definition setting the

price of all StockOption objects to 1.11, is Overridable. This means that method definitions in classes that inherit from the StockOption class may override the definition in the base, or parent, StockOption class.

7.4 Inheritance

The best way to understand inheritance is to continue the StockOption object example.

A stock option, through abstraction and encapsulation into a class and then instantiation, can be an object in VB.NET. This object built on the StockOption class contains only those properties and methods that are common to all stock options. Ceratianly, the method of calculating the price is not common to all stock options. We calculate the price of a call differently than we calculate the price of a put.

A call option is a stock option. As such, it has methods that are not common to all stock options, such calculation of its price. So, rather than create a whole new CallOption class, we can create a derived, or child class, called CallOption, that inherits all the properties and methods from the base, or parent, StockOption class. The CallOption class then may have some added properties or functionalities, like maybe pricing algorithms that are unique to call options on stocks. Likewise, we could create a PutOption class that inherits from the base StockOption class and has its own specific functionalities added on.

Continuing on then, an American call option is a call option. So, we could create a derived class called AmerCallOption that inherits all the properties and methods from the base CallOption class and so on. For the purposes of this book, however, we will stop with the CallOption class.

A derived class can add functionality beyond that of the base class and it can also override methods of its base class. That is, a derived class may replace a member function definition of the base class with its own new definition. In such cases, the base class definition should indicate which if any methods may be overridden in derived classes using the Overridable inheritance modifier. Here is a table of the inheritance modifiers.

Inheritance Modifier	Description
MustInherit	Indicates an abstract class that cannot be instantiated, only inherited.
MustOverride	Must be overridden in the derived class. Necessitates a MustInherit class.
Overridable	May be overridden in the derived class.
NotOverridable	Prevents overriding in derived classes.
Overrides	Indicates overriding a base class definition.
Shadows	Has the same name as a method in the base class.

In our program, let's create a derived class CallOption that will inherit all the member variables and methods from the base, StockOption class.

Step 8 In your program, add another class module and to it add the following code.

```
Public Class CallOption
    Inherits StockOption

Private m_Delta As Double

Public Sub New(ByVal m_Symbol As String, ByVal m_Quantity As Integer)
        MyBase.New( m_Symbol, m_Quantity)
End Sub

Protected Overrides Sub BlackScholesPrice()
   Dim d1 As Double, d2 As Double, Nd1 As Double, Nd2 As Double
   ' Calculate d1 and d2
   d1 = (Math.Log(m_Stock / m_Strike) + (m_IR + _
        (m_Sigma ^ 2) / 2) * m_TimeTillExp) / _
        (m_Sigma * Math.Sqrt(m_TimeTillExp))
   d2 = d1 - m_Sigma * Math.Sqrt(m_TimeTillExp)
   ' Calculate N(d1) and N(d2)
   Nd1 = NormCDF(d1)
   Nd2 = NormCDF(d2)
   ' Calculate the price of the call
   m_BSPrice = m_Stock * Nd1 - m_Strike * Math.Exp(-m_IR _
                    * m_TimeTillExp) * Nd2
End Sub

Public Overrides Function ToString() As String
   Return m_Quantity & " " & m_Underlying & " " & m_Month & " " _
          & m_Strike.ToString() & " Call"
End Function

Public ReadOnly Property Delta()
    Get
        CalcDelta()
        Return m_Delta
    End Get
End Property

Private Sub CalcDelta()
    m_Delta = NormCDF((Math.Log(m_Stock / m_Strike) + _
             (m_IR + (m_Sigma ^ 2) / 2) * m_TimeTillExp) / _
             (m_Sigma * Math.Sqrt(m_TimeTillExp)))
End Sub

End Class
```

In the derived class CallOption, the BlackScholesCall() method definition overrides the definition in the base StockOption class. Again, notice that the procedure in the CallOption class called BlackScholesPrice() is a member function, and therefore, has direct access to the member variables.

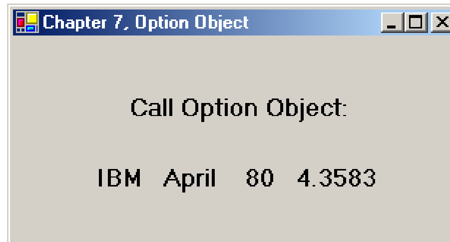
Also, because constructor methods are not inherited, we needed to add a New() method to our derived CallOption class that explicitly calls the constructor of the base class using the MyBase keyword. The MyBase keyword always references the base class within any derived class.

Step 9 Change the Form_Load event code to:

```
  Private Sub Form1_Load(ByVal sender As …) Handles MyBase.Load
          Dim MyCallOption As CallOption = New CallOption("IBMDP", 1)
          Label1.Text = MyCallOption.Underlying
          Label2.Text = MyCallOption.ExpMonth
```

```
            Label3.Text = MyCallOption.Strike
            MyCallOption.IntRate = 0.1       ` default IntRate = .1
            MyCallOption.StockPrice = 80
            MyCallOption.Volatility = 0.25
            Label4.Text = Format(MyCallOption.BSPrice, "#.0000")
  End Sub
```

Step 10 Run the program.



As we mentioned before, your program will have a different price than the one shown here since the time till expiration changes with as time moves forward. Also, the StockOption class sets the IntRate = .1 by default so, in future programs we will not need to set it explicitly.

6.5 Polymorphism

Polymorphism allows us to have one method name, or function name, used in different derived classes, but yet have different implementations, or functionalities, associated with that name depending on the class. In our CallOption class above and the PutOption class also found on the CD, for example, we have inherited a BlackScholesPrice() method from the parent StockOption class, but yet each of the derived classes has their own method for calculation, since the equations for Black Scholes call and put pricing are different.

7.6 Events

Events allow an object, called the publisher or source, to notify other objects, called the subscribers or receivers, when something happens. The most intuitive event is the button click event. When the user clicks a button, the Click event fires and, as we have seen, we can write code that will execute when this happens. Creating events in VB.NET is really quite easy. Here are the four steps to create an event:

1. Create an event member in the publisher class.
2. Raise the event using the RaiseEvent keyword.

```
Public Class Instrument

    Private m_Price As Double

    Public Sub New()
        m_Price = 100
    End Sub

    Public Event OnUpdate(ByVal px As Double)

    Public Sub CauseEvent()
        RaiseEvent OnUpdate(m_Price)
    End Sub

End Class
```

3. Create an instance of the publisher using the WithEvents keyword.

4. Create a method to subscribe to the event using the Handles keyword. This method will run when the event is raised.
5. Cause the event to be raised.

```
Public WithEvents m_Instr As Instrument

Sub Main()
   m_Instr = New Instrument
   m_Instr.CauseEvent()
End Sub

Public Sub OnUpdate_EventHandler(ByVal p As Double) Handles _
                                 m_Instr.OnUpdate
   Console.WriteLine(p)
End Sub
```

6.7 Access Modifiers

In the complete StockOption class, we have changed all the Private access modifiers to Protected, because Private member variables and private methods are not accessible in derived classes. Take a look at the BlackScholesPrice() method.

```
Protected Overridable Sub BlackScholesPrice()
```

Protected member and methods are accessible in derived classes. So, since we intended to create a derived class, CallOption, from our base class StockOption, we needed to use the Protected access modifier. Here are the access modifiers for classes:

Access Modifier	Scope
Public	Accessible anywhere.
Private	Accessible only by methods of the class. Derived class methods cannot access Private properties or methods.
Protected	Accessible by base class and derived class methods.
Friend	Accessible by base class methods, derived class methods and certain other classes.
Shared	Shared members are callable directly from the class without requiring an instance of the class.

6.8 Overloading

The complete StockOption class also contains two New() methods. This is an example of method overloading. We can create as many methods with the same name in a single class as are needed as long as the lists of input arguments are different from one another, either in number of arguments or in the data types of the arguments.

Methods other than New() that are overloaded must include the Overloads keyword. Although not illustrated in the code for StockOption, an example would be,

Public Overloads Function NormCDF(ByVal x as Integer) as Double

where this function overloads the original NormCDF() function because it differs in its parameter list.

```
Public Overloads Function NormCDF(ByVal x as Double) as Double
```

6.9 Nothing

Because the name of an object is really variable holding a reference to the location of the object in memory, we can assign a value of Nothing to the object, which allows the .NET garbage collector to dispose of the unused memory. This method disposes of the instance of the object, but not the name of the object.

VB

```
MyOption = Nothing
GC.Collect()
```

C#

```
m_Option = null;
GC.Collect();
```

We can force garbage collection using the shared Collect method on the GC class.

6.10 Arrays

In VB.NET, arrays are objects that essentially group identical value types together contiguously in memory in one or more dimensions. Hence, the Dim keyword. We can access any one element in an array by referencing the array name, and the element's index, or position or address, within the array. When doing financial modeling, we use arrays frequently, so a good understanding of them and how they work is very important. Arrays come in particularly handy when dealing with data, doing matrix algebra, and creating binomial and trinomial trees.

7.11 One Dimensional Arrays

Although arrays occupy space in memory, simply declaring an array does not create the space. Rather, because an array is a reference type, an array declaration creates a variable that stores a reference to the space in memory occupied by an array. So, creating an array object is again a two-stage process. Here is a sample declaration for an array of doubles:

```
Dim m_ClosingPrices as Double()
```

Then, the New keyword is necessary to create an actual array object. The value in parentheses defines the upper bound for the array. The lower bound is always 0.

```
m_ClosingPrices = New Double(2) {}
```

A simple way to populate an array is to use the initializer list, like this:

```
m_ClosingPrices = New Double(2) {52.5, 51.4, 45.24}
```

Alternatively, the two statements could be combined:

```
Dim m_ClosingPrices as Double() = New Double(2) {}
```

or, using the initializer:

```
Dim m_ClosingPrices as Double() = New Double(2) {52.5, 51.4, 45.24}
```

Since the index of the first element is always 0, the upper bound, in this case 2, will always be one less than the number of elements in the array, in this case 3. We can access any element in a one-dimensional array by referencing its index, or address, within the array.

```
m_ClosingPrices(0) = 52.5
m_ClosingPrices(1) = 51.4
m_ClosingPrices(2) = 45.24
```

If we attempt to access an array element outside the upper bound, say m_ClosingPrices(53), we will get an error message saying that the `index was outside the bounds of the array.` Should the situation arise, we could also declare an array of the user-defined data type QuoteData.

```
Dim qdPriceData As QuoteData() = New QuoteData(10) {}
```

And we could reference the individual elements of a QuoteData array in the following way:

```
qdPriceData(3).m_Close = 43.45
```

6.12 Two Dimensional Arrays

A two-dimensional array object could be instantiated in one line this way:

```
Dim m_Covariance as Double(,) = New Double(1,1) {}
```

We could declare and populate a two-dimensional array using the two line method and the initializer in this way:

```
Dim m_Covariance as Double(,)
m_Covariance = New Double(1,1) {{.057, .83}, {.192, -.12}}
```

We can access any element in a two dimensional array by referencing its index, or address, in the array.

```
Sub Main()
      Dim m_Covariance As Double(,)
      m_Covariance = New Double(1, 1) {{0.057, 0.83}, {0.192, -0.12}}
      Console.WriteLine(m_Covariance(0, 0))
      Console.WriteLine(m_Covariance(1, 0))
      Console.WriteLine(m_Covariance(0, 1))
      Console.WriteLine(m_Covariance(1, 1))
End Sub
```

This program prints out the elements of the m_Covariance array as:

```
.057
.192
.083
-0.12
```

As discussed in Chapter 5, we can access each element in a two-dimensional array in VB.NET by writing a nested For…Next loop structure, such as:

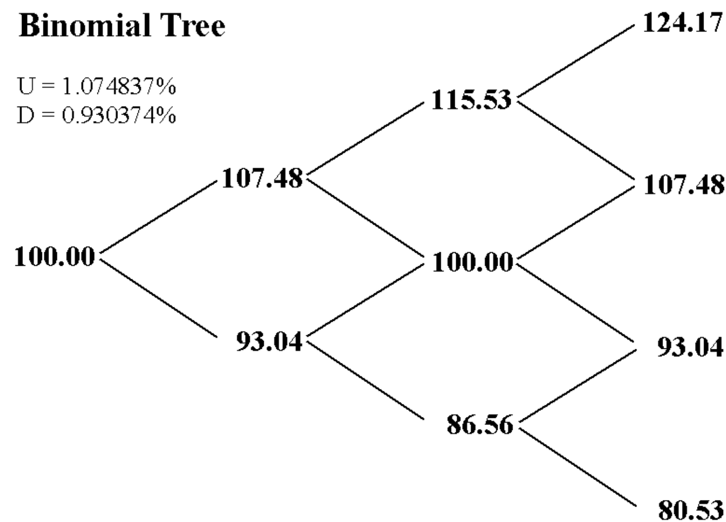
```
For Rows = 0 To 1
      For Cols = 0 To 1
            ' Do something with m_Covariance(Rows, Cols)
      Next Cols
Next Rows
```

6.13 Jagged Arrays

On occasion, the structure of the data in a program may be two-dimensional, but not rectangular. That is, not every row will be the same length. In such cases it may be advantageous from a memory savings standpoint to use a jagged array. A jagged array is an array that has rows of different lengths. In memory, a jagged array is really stored as an array of arrays. Here is how to declare a jagged array in one line.

```
Dim m_BinomTree as Double()() = New Double(2)() {}
```

Binomial trees are valuable tools in derivatives pricing and there are several methods for building binomial trees in code using arrays. Some methods only require a single dimensional array. However, in cases where the entire tree must be maintained in memory, jagged arrays work quite well. In fact, binomial trees fit rather elegantly into jagged arrays and waste no memory space. Here is a graphic showing different potential price paths of a stock:



The initial value in the tree, 100, is calculated using the formula:

$$100 = S_0 \cdot D^0 \cdot U^0$$

The two prices after one step forward, 107.43 and 93.09, are found using:

$$107.48 = S_0 \cdot D^0 \cdot U^1$$

$$93.04 = S_0 \cdot D^1 \cdot U^0$$

As you can see, we calculate individual nodes on the tree by incrementing the exponents of U and D. We can make these calculations and populate a jagged array very easily since the exponents map to the indexes of the array elements. Here is a graphic showing the array elements with their values and indeces.

100.00 (0, 0)	**107.48** (0, 1)	**115.53** (0, 2)	**124.17** (0, 3)
93.04 (1, 0)	**100.00** (1, 1)	**107.48** (1, 2)	
86.56 (2, 0)	**93.04** (2, 1)		
80.53 (3, 0)			

We can initialize the elements of a binomial tree into a jagged array as per the graphic above in this fashion:

```
Sub Main()
   Dim x, y As Integer
   Dim m_StockPrice As Double = 100
   Dim U As Double = 1.074837
   Dim D As Double = 0.930374
   Dim m_BinomTree As Double()() = New Double(3)() {}
   For x = 0 To 3
      m_BinomTree(x) = New Double(3 - x) {}
      For y = 0 To 3 - x
         m_BinomTree(x)(y) = m_StockPrice * D ^ x * U ^ y
         Console.WriteLine("( " & x & ", " & y & " ) = " & _
   m_BinomTree(x)(y))
      Next y
   Next x
End Sub
```

Jagged arrays are held in memory and require that we declare the upper bound of the first dimension first. That is we first declare the number of rows, then we can go through row by row and declare the upper bound of each particular row as in the line of code above:

```
      m_BinomTree(x) = New Double(3 - x) {}
```

6.14 Array Methods

Because arrays in VB.NET are instances of the Array class, and therefore are objects, they have properties and methods associated with them. Here are some of them as well as several functions found in the System.Array namespace. To use these functions we should include an Imports System.Array statement above all the other code in a module.

Array Properties	**Description**	**Example**
Length	Returns the total number of elements in the array.	intA = myArray.Length
Rank	Returns the number of dimensions in the array.	intA = myArray.Rank
Array Methods	**Description**	**Example**
GetLength	Returns the number of elements in a given dimension.	intA = myArray.GetLength(0)
GetUpperBound	Returns the upper bound of a given dimension.	intA = myArray.GetUpperBound(0)
System.Array Functions	**Description**	**Example**
Clear	Sets a range of elements within the array equal to 0.	Clear(SourceArray, 0, 3)
Copy	Makes a copy of all or part of an array given a length.	Copy(SourceArray, TargetArray, 5)
IndexOf	Finds the index number associated with the first occurance of a value.	intA = IndexOf(SourceArray, “IBM”)
Reverse	Reverses some or all of the elements in an array given a starting and ending index.	Reverse(SourceArray, 1, 10)
Sort	Sorts some or all of a one-dimensional array in ascending order.	Sort(SourceArray)

Here is a short console application illustrating some of these methods.

```
Imports System.Array
Module Module1
    Sub Main()
        Dim x As Integer
        Dim m_Returns As Double() = New Double(4) {0.0176, 0.0083, _
0.0232, -0.0241, 0.0077}
        Sort(m_Returns)
        For x = 0 To m_Returns.GetUpperBound(0)
            Console.WriteLine(m_Returns(x))
        Next x
    End Sub
End Module
```

This program declares and populates a one-dimensional array named m_Returns. The Sort() function puts the elements in the order from lowest to highest. The GetUpperBound() member function returns the upper bound, 4, so that the For…Next loop will run 5 times. Also, notice the inclusion of the Imports statement at the top. The function definition for Sort() is found in the System.Array namespace. We will discuss namespaces in greater detail in Chapter 10. This program prints out:

```
-.0241
.0077
.0083
```

```
.0176
.0232
```

6.15 Passing Arrays to Functions

Visual Basic.NET allows us to pass arrays to functions as input arguments and also return them from functions as output arguments, or return values. Here is an example of the basic syntax for passing arrays to and from functions:

```
Sub Main()
 Dim m_Returns As Double() = New Double(9) {0.0203, -0.0136, 0.0012, _
                                            0.0266, -0.0063, -0.0601, _
                                            0.0307, 0.0123, 0.0055, _
                                            0.0441}
 Console.WriteLine(Average(m_Returns))
End Sub
```

Here, we have created an array of doubles and populated the array, using the initializer, with 10 values representing daily returns. Then we have passed the array to a function called Average() which accepts an array of doubles as an input argument and returns a double, which is the average of the elements in the array:

```
Public Function Average(ByRef InArray As Double()) As Double
Dim m_TotalReturn As Double
            Dim x As Integer
            Dim m_Length# = UBound(InArray, 1)
            For x = 0 To m_Length
                  m_TotalReturn += InArray(x)
            Next x
            Return m_TotalReturn / (m_Length + 1)
End Function
```

Because arrays are always passed to functions by reference, it is important to remember that certain operations performed on those arrays, such as matrix transposition and inversion will actually destroy the original matrix. To avoid this situation, it may be necessary to first make a copy of the array within the function definition, and then proceed by making calculations on the new copy of the original array.

6.16 The Erase Statement

The Erase statement clears an array and releases the memory used by the array object. To reuse the array after Erase, we can use the ReDim statement.

```
            Erase m_Returns
```

67.17 Using Arrays for Data

When modeling returns, we often determine average rates of return and volatilities. In this case, we need to use continuous rates of return, such that:

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

Given historical returns, we can calculate the average return:

$$\mu_{R;t,T} = \frac{1}{n}\sum_{i=1}^{n} R_i$$

We can calculate the variance of returns:

$$\sigma_{t,T}{}^2 = \frac{1}{n}\sum_{i=1}^{n}(R_i - \overline{R})^2$$

We can calculate the skew:

$$Skew = \frac{n}{(n-1)(n-2)}\sum_{i=1}^{n}\left(\frac{R_i - \overline{R}}{s}\right)^3$$

The skewness of a distribution characterizes the degree of asymmetry around its mean. Positive skewness indicates an asymmetric tail extending toward more positive values. Negative skewness indicates asymmetric tail extending toward negative values.

We can calculate the kurtosis:

$$Kurtosis = \left\{\frac{n(n+1)}{(n-1)(n-2)(n-3)}\sum_{i=1}^{n}\left(\frac{R_i - \overline{R}}{s}\right)^4\right\} - \frac{3(n-1)^2}{(n-2)(n-3)}$$

Returns the kurtosis of a data set. Kurtosis characterizes the relative peakedness or flatness of a distribution compared with the normal distribution. Positive kurtosis indicates a relatively peaked distribution. Negative kurtosis indicates a relatively flat distribution.

Step 1 In VB.NET, open a new Windows application called DataArray.

Step 2 Add five labels to the form.

Step 3 Add five modules and in them, place the functions for Average(), Var(), VarP(), Skew() and Kurtosis() from the CD.

Step 4 In the form load event, add the following code to pass an array of return data into each of the functions.

```
Private Sub Form1_Load(ByVal sender As …) Handles MyBase.Load
 Dim m_Returns As Double() = New Double(9) {0.0203, -0.0136, 0.0012, _
  0.0266, -0.0063, -0.0601, _
  0.0307, 0.0123, 0.0055, _
  0.0441}
        Label1.Text = Format(Average(m_Returns), "#.#####")
        Label2.Text = Format(Var(m_Returns), "#.#####")
        Label3.Text = Format(VarP(m_Returns), "#.#####")
        Label4.Text = Format(Skew(m_Returns), "#.#####")
        Label5.Text = Format(Kurtosis(m_Returns), "#.#####")
End Sub
```

Step 5 Run the program.

Chapter 8, Data Array Example

Average	.00607
Variance	.00085
Pop Variance	.00077
Skew	-1.21773
Kurtosis	2.29979

As with any calculations you make in code, be sure to verify your them against Excel's built in functions--Average(), Var(), VarP(), Skew() and Kurt().

6.18 Using Arrays for Matrix Algebra

We often use matrix algebra when doing financial research. For example, modern portfolio management techniques often make use of covariance matrices. Using matrix notation, we can calculate the variance of a portfolio in the following manner:

$$\sigma_P^2 = \omega' \Omega \, \omega$$

Where Ω is the covariance matrix and ω is the vector of portfolio weights. A covariance matrix of course exists in two dimensions, where:

$$\Omega = Cov(r_A, r_B) = \frac{1}{n} \sum [r_{A,i} - E(r_A)][r_{B,i} - E(r_B)]$$

So that, for example, a covariance matrix for a 3 asset portfolio is:

$$\Omega = \begin{bmatrix} .0025 & -.0011 & -.001 \\ -.0011 & .0058 & .0003 \\ -.001 & .0003 & .0048 \end{bmatrix}$$

and,

$$\omega = \begin{bmatrix} .3 \\ .5 \\ .2 \end{bmatrix}$$

To calculate the portfolio variance, σ_P^2, we need to employ some specialized matrix math functions that handle the algorithms using two dimensional arrays. Fortunately, the CD contains several math functions that manipulate two-dimensional arrays.

Step 6 In VB.NET, open a new Windows application called MatrixArray.

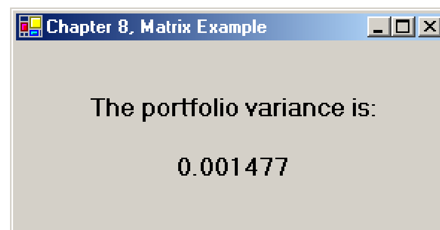
Step 7 Add at least one label to the form.

Step 8 Add two modules and paste in the code for the MMult2by1() and MMult1by1() functions.

Step 9 Add the following code to the Form1_Load event.

```
Private Sub Form1_Load(ByVal sender As …) Handles MyBase.Load
Dim m_Covar As Double(,) = New Double(2, 2) { _
{0.0025, -0.0011, -0.001}, _
{-0.0011, 0.0058, 0.0003}, _
{-0.001, 0.0003, 0.0048}}
    Dim m_Weights As Double() = New Double(2) {0.3, 0.5, 0.2}
    Dim m_PortVar As Double
    m_PortVar = MMult1by1(MMult2by1(m_Covar, m_Weights), m_Weights)
    Label1.Text = Val(m_PortVar)
End Sub
```

Step 10 Run your program. Again, be sure to verify your calculations against Excel's built in function MMult().



6.19 Using Arrays for Trees

Here we will show a simple example using a jagged array to price an American call option using a binomial tree. The call option has the following attributes: stock price, S, is 100, strike price, X, is 100, time till expiration is three months, interest rate, I, is .1 and the annualized volatility, sigma, is .25. The binomial tree will consist of three steps, as in the example using a jagged array previously in the chapter. Each step in the tree then will be three calendar months or 21 trading days, .25 of a year, divided by 3, so that the change in time for each step is $t = .25 / 3 = .083333$. This option will expire a total of three steps, one for each of the three months. So, N = 3. We calculate U and D thusly:

$$U = e^{\sigma\sqrt{t}} = e^{.25\sqrt{08333}} = 1.074837$$

$$D = e^{-\sigma\sqrt{t}} = e^{-.25\sqrt{08333}} = .930374$$

We will add a variable A to shorten the calculations:

$$A = e^{-I \cdot t} = e^{-.1 \cdot .083333} = .991701$$

Also, the probability of an up move is found, such that:

$$P = \frac{(e^{I \cdot t} - D)}{U - D} = \frac{e^{.1 \cdot .083333} - .930374}{1.074837 - .930374} = .5399892$$

Once we have declared and defined the necessary variables, we can calculate the terminal payoffs for the option for each outcome by calculating the intrinsic value in this way:

$$Tree_{x,N-x} = \max(S_0 \cdot D^x \cdot U^{N-x} - X, 0)$$

American style options require that a decision be made at each node as to whether to exercise the option. So, we must compare the intrinsic value of the option with the risk neutral valuation at each node:

$$Tree_{x,y-x} = \max(S_0 \cdot D^x \cdot U^{y-x} - X, A \cdot (P \cdot Tree_{x,y-x+1} + (1-P) \cdot Tree_{x+1,y-x})$$

where y = N-1

The value of the call option will then be Tree(0)(0).

Step 11 In VB.NET open a new console application called TreeArray.

Step 12 Add the following code:

```
Imports System.Math
Module Module1
Sub Main()
    Dim x, y As Integer

    Dim N As Integer = 3
    Dim m_StockPrice As Double = 100
    Dim m_Strike As Double = 100
    Dim m_TimeStep As Double = .25 / N
    Dim m_IntRate = 0.1
    Dim m_Sigma = 0.25

    Dim U As Double = Exp(m_Sigma * m_TimeStep ^ 0.5)
    Dim D As Double = Exp(-m_Sigma * m_TimeStep ^ 0.5)
    Dim A As Double = Exp(-m_IntRate * m_TimeStep)
    Dim P As Double = (Exp(m_IntRate * m_TimeStep) - D) / (U - D)

    Dim m_BinomTree As Double()() = New Double(N)() {}
    For x = 0 To N
        m_BinomTree(x) = New Double(N - x) {}
        m_BinomTree(x)(N - x) = Max((m_StockPrice * D ^ x * _
                                 U ^ (N - x)) - m_Strike, 0)
    Next x
    For y = N - 1 To 0 Step -1
        For x = 0 To y
            m_BinomTree(x)(y - x) = Max((m_StockPrice * D ^ (x) * _
                                U ^ (y - x)) - m_Strike, A * (P * _
                                m_BinomTree(x)(y - x + 1) + (1 - P) * _
                                m_BinomTree(x + 1)(y - x)))
        Next x
    Next y
    Console.WriteLine("The price of the call option is: " & _
                      m_BinomTree(0)(0))
End Sub
End Module
```

Step 13 Run the program by selecting Start Without Debugging from the Debug menu item.

The value of the call option using this method is 6.6468, which rounds to 6.65. Here is a map of the values of m_BinomTree()().

6.65 (0, 0)	10.59 (0, 1)	16.36 (0, 2)	24.17 (0, 3)
2.15 (1, 0)	4.01 (1, 1)	7.48 (1, 2)	
0 (2, 0)	0 (2, 1)		
0 (3, 0)			

We can increase the accuracy of our pricing model by increasing the number of steps, N. For example, if we change N to 20, so that t = .25/20 = .0125, the value of the call option is 6.19

6.20 Calculating At-The-Money Volatility

Vary rarely, if ever, in financial markets can we look at an at-the-money (ATM) option and calculate its implied volatility. Yet, in our discussions about markets, we often talk in terms of ATM volatility. Quantitative research papers frequently use time series of ATM volatility and, what's more, many mathematical models assume the reader understands that volatility means at-the-money volatility. But, what is ATM volatility if it cannot be observed in the marketplace? The answer is that ATM volatility is a value we must calculate from the implied volatilities of the puts and calls with the strikes surrounding ATM, those nearest, above and below, the price of the underlying. Furthermore, since time is always moving forward and expirations are continuously drawing nearer, we have to include volatilities for the nearby and second nearby expirations to come up with a constant maturity ATM volatility. That is, if we wish to refer to an ATM volatility that is, for example, 30 calendar days out, which is somewhat difficult to envision, since only on one day a month will an expiration be exactly 30 days away, we need a mathematical construct to interpolate between options in the nearby and second nearby expirations.

In this section we will use the Chicago Board Options Exchange's market volatility index (VIX) methodology for calculating ATM volatility. As described by Robert Whaley in his paper "The Investor Fear Gauge," the VIX represents the ATM volatility for the S&P 100 (OEX) index. The CBOE computes the value of the VIX from the prices of eight puts and calls with the strikes nearest, above and below, the price of the underlying for the nearby and second nearby expirations (Whaley, p1). The implied volatilities derived from these eight options are then weighted to form a 30 calendar day, 22 trading day, constant maturity, ATM implied volatility for the OEX index. The prices used for these eight options will be the midpoints between the respective bids and offers.

While the implied volatilities for these eight options should be calculated using a cash dividend-adjusted binomial method to account for the facts that OEX index options are American-style and that the underlying index portfolio pays discrete cash dividends, we will use the traditional Black-Scholes model for European options to derive all required implied volatilities. Forecasting dividends for the 100 stocks that make up the index is beyond the scope of this book. As you can imagine, this will of course lead to small deviations from the value of the actual VIX.

If it happens that the implied volatilities for these eight options are calculated using calendar days, then each must be converted to a trading day implied volatility. If the number of calendar days to expiration in $Days_C$ and the number of trading days till expiration is $Days_T$, then $Days_T$ is calculated as follows:

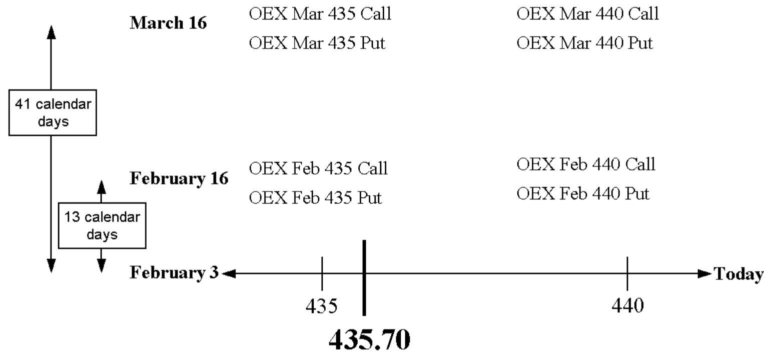
$$Days_T = Days_C - 2 \cdot \text{int}(Days_C / 7)$$

To convert calendar day volatilities to trading day volatilities, we multiply the eight by the square root of the ratio of number of calendar days to the number of trading days thusly:

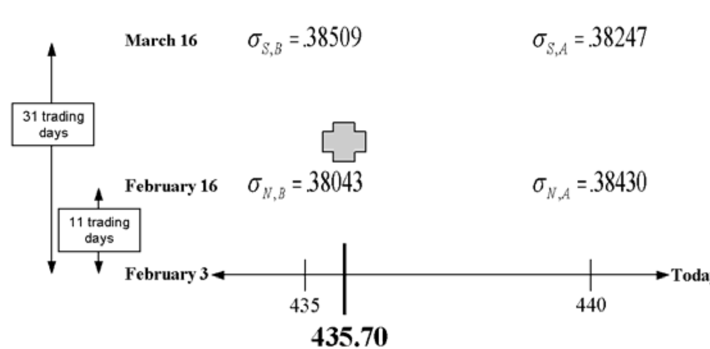
$$\sigma_T = \sigma_C \left(\sqrt{\frac{N_C}{N_T}} \right)$$

Foretunately, the StockOption class already assumes trading days for time to expiration, so we will not need to make this adjustment.
In practice, the risk-free interest rate we should use in the calculation is the continuous yield of the T-bill with the maturity most closely matching the option's expiration. If the time till expiration is shorter than 30 days, however, the 30 day T-bill rate is used. The StockOption class sets the default interest rate to .1 and we will just use that.

The calculations will be clearer if we look at an example. Let's assume today is February 3rd and the S&P 100 (OEX) index is at 435.70. The options with the nearest strikes above and below would be the 435s and 440s. If we take the midpoints of the bids and asks of the puts and calls for the next two expirations, February 16 and March 16, for both these strikes, we will have eight option prices and eight trading-day volatilities.



Now, we need to average the eight implied volatilities to arrive at a single ATM volatility 22 days hence, denoted by the gray X on the graphic. First, we average the call and put volatilities in each of the quadrants respectively to reduce the number of volatilities to four.



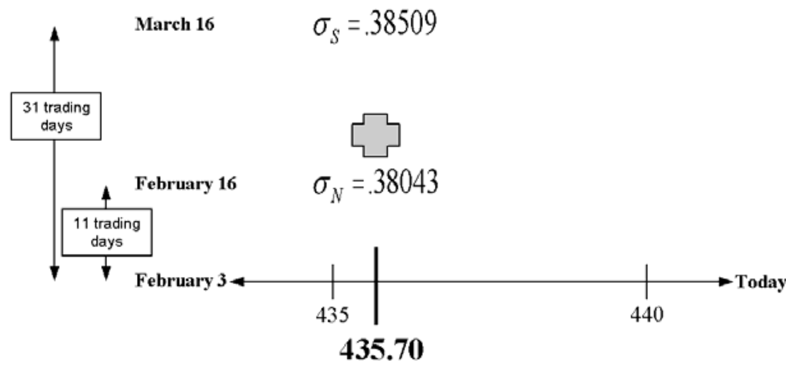
In this graphic, the suffix N refers to the nearby expiration, S to the second nearby, and A and B mean above and below the current price of the underlying. In the upcoming formulae P stands for the price of the underlying and X means strike price, so that XA refers to the strike price above the price of the underlying and XB to the strike price below. Also, in upcoming formulae, N refers to the number of trading days so that NN and NS refer to the number of trading days till the nearby and second nearby expirations respectively.

Second, we average the two volatilities across each of the two expirations. The average of the two nearby volatilities to arrive at the ATM volatility for the nearby expiration is found using:

$$\sigma_N = \sigma_{N,B}\left[\frac{X_A - P}{X_A - X_B}\right] + \sigma_{N,A}\left[\frac{P - X_B}{X_A - X_B}\right]$$

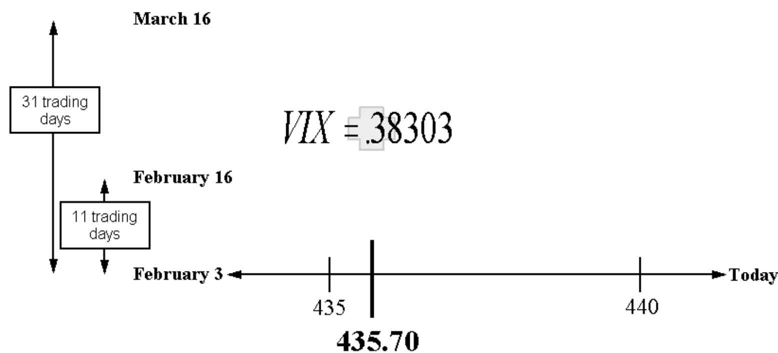
and the ATM volatility for the second nearby expiration is found using:

$$\sigma_S = \sigma_{S,B}\left[\frac{X_A - P}{X_A - X_B}\right] + \sigma_{S,A}\left[\frac{P - X_B}{X_A - X_B}\right]$$



Third and last, we average the two remaining volatilities two arrive at a constant maturity 22 trading hence using:

$$VIX = \sigma_N \left[\frac{N_S - 22}{N_S - N_N} \right] + \sigma_S \left[\frac{22 - N_N}{N_S - N_N} \right]$$



(These calculations are all taken from Whaley, p. 12 ff.)
Now, let's create a VB.NET Windows application that uses option objects to calculate the constant maturity ATM volatility for IBM using the VIX methodology, again assuming no dividends.

Step 11 Open a new VB.NET Windows application called ATMExample.

Step 12 On the menu bar, select Project, Add Class three times and paste in the code for the StockOption, CallOption and PutOption classes.

Step 13 Now, we will need eight put and call objects and eight corresponding prices. This will require 16 text boxes laid out in a fashion similar to the graphic shown for the VIX calculation. Name the text boxes with the following scheme: the name of the text box for the nearby call option with the strike below the underlying price should be txtCallNB for Call, Nearby, Below. The text box for the second nearby put with the strike price above the underlying price should be txtPutSA for Put, Second, Above. Here is a picture showing the respective names for the text box controls.

Chapter 7, ATM Example

Second Nearby

Call	Price	Call	Price
txtCallSB	txtCallSBprice	txtCallSA	txtCallSAprice
Put	**Price**	**Put**	**Price**
txtPutSB	txtPutSBprice	txtPutSA	txtPutSAprice

Nearby

Call	Price	Call	Price
txtCallNB	txtCallNBprice	txtCallNA	txtCallNAprice
Put	**Price**	**Put**	**Price**
txtPutNB	txtPutNBprice	txtPutNA	txtPutNAprice

Calculate ATM Volatility

ATM Volatility

Underlying Price txtUnderlyingPrice

At the Money Volatility lblATMvol

Step 14 Add the following code to the Button1_Click event to read in the price of the underlying IBM stock and create eight put and call objects and set their MarketPrices and StockPrices.

```
Dim UnderlyingPrice As Double = txtUnderlyingPrice.Text
Dim CallNB As New CallOption(txtCallNB.Text, 1)
CallNB.MarketPrice = txtCallNBprice.Text
CallNB.StockPrice = UnderlyingPrice

Dim PutNB As New PutOption(txtPutNB.Text, 1)
PutNB.MarketPrice = txtPutNBprice.Text
PutNB.StockPrice = UnderlyingPrice

Dim CallNA As New CallOption(txtCallNA.Text,1)
CallNA.MarketPrice = txtCallNAprice.Text
CallNA.StockPrice = UnderlyingPrice

Dim PutNA As New PutOption(txtPutNA.Text,1)
PutNA.MarketPrice = txtPutNAprice.Text
PutNA.StockPrice = UnderlyingPrice

Dim CallSB As New CallOption(txtCallSB.Tex,1t)
CallSB.MarketPrice = txtCallSBprice.Text
CallSB.StockPrice = UnderlyingPrice

Dim PutSB As New PutOption(txtPutSB.Text,1)
PutSB.MarketPrice = txtPutSBprice.Text
PutSB.StockPrice = UnderlyingPrice

Dim CallSA As New CallOption(txtCallSA.Text,1)
CallSA.MarketPrice = txtCallSAprice.Text
CallSA.StockPrice = UnderlyingPrice
```

```
Dim PutSA As New PutOption(txtPutSA.Text,1)
PutSA.MarketPrice = txtPutSAprice.Text
PutSA.StockPrice = UnderlyingPrice
```

As mentioned earlier, the StockOption class already calculates the time till expiration using trading days as opposed to calendar days, so no conversion of the volatilities will be necessary.

Step 15 Once these eight option objects are created, we need to average the call and put volatilities in each of the quadrants respectively to reduce the number of volatilities to four. For this we will need four new variables of type double. Add the following code to the Button1_Click event.

```
Dim m_VolNB, m_VolNA, m_VolSB, m_VolSA As Double
m_VolNB = (CallNB.ImpliedVol + PutNB.ImpliedVol) / 2
m_VolNA = (CallNA.ImpliedVol + PutNA.ImpliedVol) / 2
m_VolSB = (CallSB.ImpliedVol + PutSB.ImpliedVol) / 2
m_VolSA = (CallSA.ImpliedVol + PutSA.ImpliedVol) / 2
```

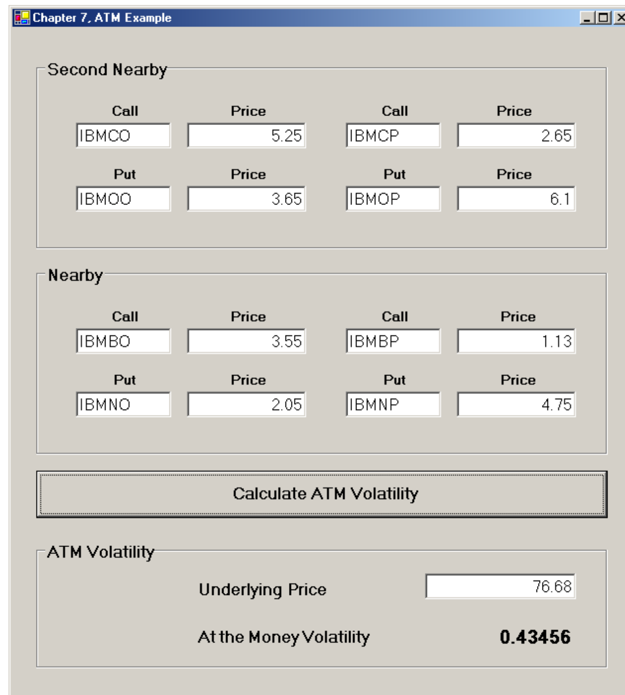
Step 16 Now we will need to weight the above and below volatilities to arrive at an average volatility for each of the two expirations, nearby and second nearby.

```
Dim m_NearbyVol, m_SecondVol As Double
m_NearbyVol = m_VolNB * ((CallNA.Strike - UnderlyingPrice) / _
    (CallNA.Strike - CallNB.Strike)) + m_VolNA * ((UnderlyingPrice - _
     CallNB.Strike) / (CallNA.Strike - CallNB.Strike))
m_SecondVol = m_VolSB * ((CallSA.Strike - UnderlyingPrice) / _
    (CallSA.Strike - CallSB.Strike)) + m_VolSA * ((UnderlyingPrice - _
     CallSB.Strike) / (CallSA.Strike - CallSB.Strike))
```

Step 17 And, finally, we can calculate the ATM constant maturity volatility.

```
Dim ATMVol As Double = m_NearbyVol * ((CallSA.DaysTillExp - 22) / _
   CallSA.DaysTillExp - CallNA.DaysTillExp))  + m_SecondVol * ((22 - _
   CallNA.DaysTillExp) / (CallSA.DaysTillExp - CallNA.DaysTillExp))
lblATMvol.Text = Format(ATMVol, "0.#####")
```

Step 18 Run the program.



The results you get will be different from the results shown since the time to expiration calculations are continuously changing.

As you can see, creating and managing multiple objects can be quite a difficult task code wise. Suppose for example, we had a portfolio of 100 options. How much coding woud we have to do then? Obviously, we will need a superior method for dealing with this situation. In the following chapter, we will discuss arrays, which are a convenient way to hold multiple value types, that is, variables. In later chapters, we will look at data structures, which provide convenient methods for dealing with groups of objects such as portfolios of options.

6.21 Chapter Six Summary

In this chapter we introduced the concepts of classes and objects. Object oriented programming necessitates that we understand the ideas of abstraction, encapsulation, polymorphism and inheritance. Further we used the StockOption and CallOption classes to illustrate these concepts as well as access modifiers and method overloading. Lastly, we built a complex model using eight put and call objects to calculate the ATM volatility for IBM. This was a complex program!

6.22 Chapter Problems

1. In OOP, what is meant by the term abstraction?
2. What is encapsulation?
3. What is polymorphism?
4. What is inheritance?
5. What are the differences between the access modifiers Public, Private and Protected?
6. What is a jagged array?
7. How do we pass an array to a function?
8. Why is declaring an array a two-stage process?
9. Because arrays are reference types, what is the danger with passing arrays to functions?

Project One

Create a Stock class. Although this class will be comparatively simple, you should add private member variables for, at least, the ticker, price, dividend and dividend date, along with public properties for each of them. You should set the ticker in the constructor function New(). Then create a VB.NET Windows application that creates an object based upon the Stock class using user-entered values. Override the ToString() method to print out the ticker and the price in a label.

In VB.NET, the overridable ToString() method is inherited by every class by default. ToString() allows us to simply print out a string representation of an object. Within your Stock class, you can implement this method in the following way:

```
Public Overrides Function ToString() As String
    Return strTicker & " " & str(m_StockPrice)
End Function
```

Project Two

Create a Windows application that sorts an array of 20 returns. Then, print out the fifth lowest return in a text box. Hard code the returns in a one-dimensional array.

Project Three

Create an Instrument class. Add private member variables for a bid and ask price along with public readonly properties for each of them. You should set the bid and ask in the constructor function. Then create a .NET Windows application that creates an object based upon the Instrument class.

Project Four

Create a class called Statistics. This class should contain shared functions for mean, standard deviation, skew and kurtosis. The parameter for each function should be an array of doubles. Put this class in a .dll file and use the .dll in console application to calculate the stats on an array of data.

Project Five

Create a Windows application that creates a two-dimensional covariance matrix given three one dimensional arrays of returns for three stocks. So, the covariance matrix should look like the following:

$$\begin{bmatrix} \sigma_{a,a} & \sigma_{a,b} & \sigma_{a,c} \\ \sigma_{b,a} & \sigma_{b,b} & \sigma_{b,c} \\ \sigma_{c,a} & \sigma_{c,b} & \sigma_{c,c} \end{bmatrix}$$

Hard code the three arrays of returns and use the Covariance() function on the CD to make the calculations. Print out the matrix in labels on the form.

Project Six

To the CallOption and PutOption classes, add methods for the Greeks. Build a Windows application that accepts user inputs for an options symbol, a stock price, and a volatility, and calculates the Black Scholes price and greeks for either the call or a put. Your program should print out in labels all the necessary information including the price and all the Greeks.