Simplified

e- Book

VISAUL BASIC

SARVA EDUCATION [SM] *- An I.T & Skill Advancement Training Programme, Initiated by* ***SITED®-India***

An ISO 9001:2015 Certified Organization

VISUAL BASIC

CHAPTER – 1

HISTORY OF BASIC

INTRODUCTION

Visual Basic was designed in 1964 by ***John George Kemeny*** and ***Thomas Eugene Kurtz*** at ***Dartmouth College*** in ***New Hampshire, USA*** to provide computer access to non-science students.

B (BEGINNER'S)
A (ALL-PURPOSE)
S (SYMBOLIC)
I (INSTRUCTION)
C (CODE)

It is a family of high-level programming languages.

In the mid- 1970's two college students write first Basic for a microcomputer (Altair) – cost $350 on cassette tape. You may have heard of them: Bill Gates and Paul Allen!

Every Basic since then essentially based on that early version. **Examples include:** GW-Basic, QBasic, QuickBasic.
Visual Basic was introduced in 1991.

What is Visual Basic?

Visual Basic is a tool that allows you to develop Windows **(Graphic User Interface -GUI)** applications. The applications have a familiar appearance to the user. As you develop as a Visual Basic programmer, you will begin to look at Windows applications in a different light. You will recognize and understand how various elements of Word, Excel Access and other applications work. You will develop a new vocabulary to describe the elements of Windows applications.

- **Visual Basic:** is event-driven, meaning code remains idle until called upon to respond to some event (button pressing, menu selection,…) Visual Basic is governed by an event processor. Nothing happens until an event is detected. Once an event is detected, the code corresponding to that event (event procedure) is executed. Program control is then returned to the event processor.

  All Windows applications are event-driven. For example, nothing happens in World until you click on a button, select a menu option, or type some text. Each of these actions is an event.

- The event-driven nature of Visual Basic makes it very easy to work with. As you develop a Visual Basic application, event procedures can be built and tested individually, saving development time. And, often event procedures are similar in their coding, allowing reuse (and lots of copy and paste).

Some Features of Visual Basic

- Full set of controls – you draw the application
- Lots of icons and pictures for your use
- Response to mouse and keyboard actions
- Clipboard and printer access
- Full array of mathematical, string handling, and graphics functions
- Can handle fixed and dynamic variable and control arrays
- Sequential and random access file support
- Useful debugger and error – handling facilities
- Powerful database access tools
- ActiveX support
- Package & Development Wizard makes distributing your applications simple

Visual Basic 6 versus Other Versions of Visual Basic

The original Visual Basic for DOS and Visual Basic For Windows were introduced in ***1991***.
Visual Basic 3 (a vast improvement over previous versions) was released in 1993.
Visual Basic 4 released in late 1995 (added 32 bit application support)
Visual Basic 5 released in late 1996. New environment, supported creation of ActiveX controls, deleted 16 bit application support.
And, Visual Basic 6 – some identified new feature of Visual Basic 6.

- Faster Compile
- New ActiveX data control object
- Allows database integration with wide variety of applications
- New data report designer
- New Package & Deployment Wizard
- Additional internet capabilities

Applications built using Visual Basic 6 will run with Windows 95, Windows 98, Windows 2000, or Windows NT.

Visual Basic 2008- is one of the latest versions of Visual Basic launched by Microsoft in 2008. The latest version is ***Visual Basic 2010***, launched this year. VB2008 is almost similar to Visual Basic 2005 but it has added many new features. Visual Basic 2008 is a full fledged Object-Oriented Programming (OOP) Language, so it has caught up with other OOP languages such as C++, Java, C# and others. However, you don't have to know OOP to learn VB2008. In fact, if you are familiar with Visual Basic 6, you can learn VB2008 effortlessly because the syntax and interface are similar.

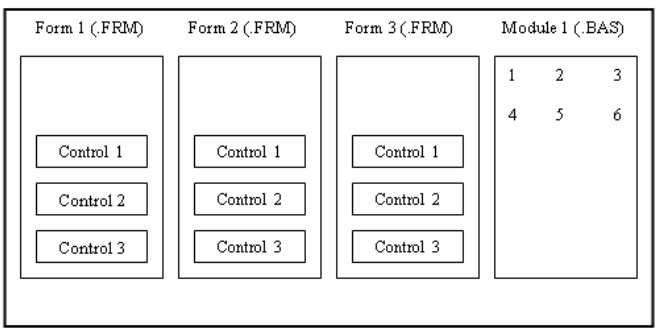
Visual Basic Editions

Visual Basic is available in three versions, each geared to meet a specific set of development requirement:

- The Visual Basic learning edition allows programmers to easily create powerful applications for Microsoft Windows and Windows NT®. It includes all intrinsic controls, plus grid, tab, and data-bound controls.
- The professional edition provides computer professionals with a full featured set of tools for development solutions for others. It includes all the features of the learning edition, plus additional ActiveX controls, the Internet Information Server Application Designer, integrated Visual Database Tools and Environment, Active Data Objects, and the Dynamic HTML Page Designer. Documentation provided with the Professional edition includes the Visual Studio Professional Features book plus Microsoft Developer Network CDs containing full online documentation.
- The Enterprise edition allows professionals to create robust distributed applications in a team setting. It includes all the feature of the Professional edition, plus Back Office tools such as SQL Server, Microsoft Transaction Server, Internet Information Server, Visual SourceSafe, SNA Server, and more. Printed documentation provided with the Enterprise edition includes the Visual Studio Enterprise Features book plus Microsoft Developer Network CDs containing full online 7.

Structure of a Visual Basic Application

Project (.VBP, .MAK)



Application (Project) is made up of:-

- **Forms:** Windows that you create for user interface.
- **Controls:** Graphical features drawn on forms to allow user interaction (text boxes, labels, scroll bars. Command buttons, etc.) (Forms and Controls are **objects.**
- **Properties:** Every characteristic of a form or control is specified by a property. Example properties includes names, captions, size, color, position, and contents. Visual Basic applies default properties. You can change properties at design time or run time.
- **Methods:** Built-in procedure that can be invoked to impart some action to a particular object.
- **Event Procedures:** Code related to objects. This is the code that is executed when a certain event occurs.
- **General Procedures:** Code not related to objects. This code must be invoked by the application.
- **Modules:** Collection of general procedures, variable declarations and constant definitions used by application.

Steps in Developing Application

- The Visual Basic development environment makes building an application straight ward process. There are three primary steps involved in building a Visual Basic application**:**

1. ***Draw*** the user ***interface*** by placing controls on the form
2. ***Assign properties*** to controls
3. **Attach code** to control events (and perhaps write other procedures)

These same steps are followed whether you are building a very simple application or one involving many controls and many lines of code.

- The ***event-driven*** nature of Visual Basic allows you to build your application in stages and test it at each stage. You can build one procedure, or part of a procedure, at a time and try it until it works as desired. This minimizes errors and gives you, the programmer, and confidence as your application takes shape.
- As you progress in your programming skills, always remember to take this sequential approach to building a Visual Basic application. Build a little, test a little, modify a little and test again. You'll quickly have a completed application. This ability to quickly build something and try it makes working with Visual Basic fun - not a quality found in some programming environments! Now, we'll look at each step in the application development process.

Drawing the User Interface and Setting Properties

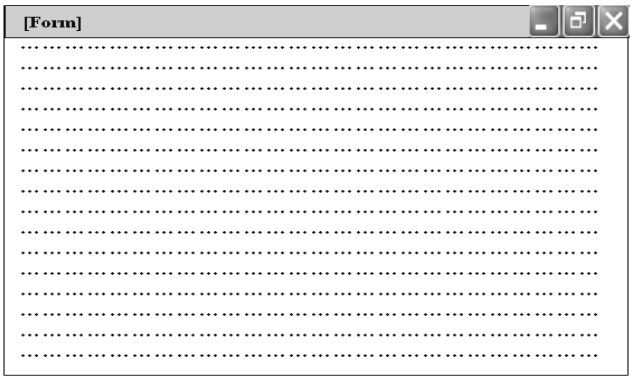
Visual Basic operates in three modes:-

- **Design mode:-** used to build application
- **Run mode:-** used to run the application
- **Break mode:-** application halted and debugger is available

We focus here on the **design mode**.

- Six windows appear when you start Visual Basic. Each window can be viewed (made visible) by selecting menu options, depressing function keys or using the toolbar. Use the method you feel most comfortable with.
- The **Main Window** consists of the title bar, menu bar, and toolbar. The title bar the project name, the current Visual Basic operating mode, and the current form. The menu bar has drop-down menus from which you control the operation of the Visual Basic environment. The toolbar has buttons that provide shortcuts to some of the menu options. The main window also shows the location of the current form relative to the upper left corner of the screen and the width and length of the current form of particular interest is the Help menu item. The Visual Basic on-line help system is invaluable as you build applications. Become accustomed with its use. Usually

- The **Form Window** is central to developing Visual Basic applications. It is where you draw your application.



- The Toolbox is the selection menu for controls used in your application. Help with any control is available by clicking the control and pressing **<F1>**
- The Properties Window is used to establish initial property values for objects (controls). The drop-down box at the top of the window lists all objects in the current form. Two views are available: Alphabetic and Categorized Under this box are the available properties for the currently selected object Help with any property can be obtained by highlighting the property of interest and pressing **<F1>.**
- The **Form Layout Window** shows where (upon program execution) your form will be displayed relative to your monitor's screen**:**
- The **Project Window** displays a list of all forms and modules making up your application. You can also obtain a view of the **Form** or **Code** windows (window containing the actual Basic coding) from the Project window.
- Variable are used by Visual Basic to hold information needed by your application. Rules used in naming variable:
  - No more than 40 characters
  - They may include letters, numbers, and underscore (__)
  - The first character must be a letter
  - You cannot use a reserved word (word needed by Visual Basic)

Data Types

Variables are placeholders used to store values; they have names and data types. The data type of a variable determines how the bits representing those values are stocked in the computer's memory. When you declare a variable, you can also supply a data type for it. All variables have a data type that determines what kind of data they can store.

By default, if you don't supply a data type, the variable is given the Variant data type. The Variant data type is like a chameleon — it can represent many different data types in different situations. You don't have to convert between these types of data when assigning them to a Variant variable: Visual Basic automatically performs any necessary conversion. If you know that a variable will always store data of a particular type, however, Visual Basic can handle that data more efficiently if you declare a variable of that type. For example, a variable to store a person's name is best represented as a string data type, because a name is always composed of characters.

Data types apply to other things besides variables. When you assign a value to a property, that value has a data type; arguments to functions also have data types. In fact, just about anything in Visual Basic that involves data also involves data types.

You can also declare arrays of any of the fundamental types.

just pressing **<F1>** can get you the help you need.

Declaring Variables with Data Types

Before using a non-Variant variable, you most use the Private, Public, Dim or Static statement to declare it As type. For example, the following statements declare an Integer, Double, String, and Currency type, respectively:

Private I As Integer
Dim Amt As Double
Static YourName As String
Public BillsPaid As Currency

A Declaration statement can combine multiple declarations, as in these statements:

Private I As Integer, Amt As Double
Private YourName As String, BillsPaid As Currency
Private Test, Amount, J As Integer

Note: If you do not supply a data type. The variable is given the default type. In the preceding example, the variables Test and Amount are of the Variant data type. This may surprise you if your experience with other programming languages leads you to expect all variables in the same declaration statement to have the same specified type (in this case, Integer).

Numeric Data Types

Visual Basic supplies several numeric data types — Integer, Long **(**long integer**)**, Single **(**single-precision floating point**)**, Double **(**double-precision floating point**)**, and Currency. Using a numeric data type generally uses less storage space than a variant.

It you know that a variable will always store whole numbers **(such as 12)** rather than numbers with a fractional amount **(such as 3.57),** declare it as an Integer or Long type. Operations are faster with integers, and these types consume less memory than other data types. They are especially useful as the counter variables in For...Next loops.

If the variable contains a fraction, declare it as a Single, Double, or Currency variable. The Currency data type supports up to four digits to the right of the decimal separator and fifteen digits to the left; it is an accurate fixed-point data type suitable for monetary calculations. Floating-point (Single and Double) numbers have much larger ranges than Currency, but can be subject to small rounding errors.

Note: Floating-point values can be expressed as *mmm***E***eee* or *mmm***D***eee*, in which *mmm* is the mantissa and ***eee*** is the exponent (a power of 10). The highest positive value of a Single data type is 3.402823E+38, or 3.4 times 10 to the 38^{th} power; the highest positive value of a Double data type is 1.7976931348G232D+308, or about 1.8 times 10 to the 308^{th} power. Using **D** to separate the mantissa and exponent in a numeric literal causes the value to be treated as a Double data type. Likewise, using

The Byte Data Type

If the variable contains binary data, declare it as an array of the Byte data type. (Arrays are discussed in "Arrays" later in this chapter). Using Byte variables to store binary data preserves it

during format conversions. When String variables are converted between ANSI and Unicode formats, any binary data in the variable is corrupted. Visual Basic may automatically convert between ANSI and Unicode when:

- **Reading from files**
- **Writing to files**
- **Calling DLLs**
- **Calling methods and properties on objects**

All operators that work on integers work with the Byte data type except unary minus. Since Byte is an unsigned type with the range 0-255, it cannot represent a negative number. So for unary minus, visual Basic coerces the Byte to a signed integer first.
All numeric variables can be assigned to each other and to variables of the Variant type. Visual Basic rounds off rather than truncates the fractional part of a floating-point number before assigning it to an integer.

The String Data Type

If you have a variable that will always contain a string and never a numeric value, you can declare it to be of type .String:

Private S as String

You can then assign strings to this variable and manipulate it using string functions

```
S = "Database"
S = Left (S, 4)
```

By default, a string variable or argument is a variable-length string', the string grows or shrinks as you assign new data to it. You can also declare strings that have a fixed length. You specify a fixed-length string with this syntax:

```
String   * size
```

For example, to declare a string that is always 50 characters long, use code like this:

```
Dim EmpName As String   *   50
```

If you assign a string of fewer than 50 characters, EmpName is padded with enough trailing spaces to total 50 characters. If you assign a string that is too long for the fixed-length siring, Visual Basic simply truncates the characters.
Because fixed-length strings are padded with trailing spaces, you may find the Trim and RTrim functions, which remove the spaces, useful when working with them.
Fixed-length strings in standard modules can be declared as Public or Private\ In forms and class modules, fixed-length strings must be declared Private.

Exchanging Strings and Numbers.

You can assign a string to a numeric variable if the string represents a numeric value. It's also possible to assign a numeric value to a string variable. **For example,** place a command button, text box and list box on a form. Enter the following code in the command button's Click event Run the application, and click the command button.
Private Sub Command 1 Click ( )

```
'Dim intx As Integer
'Dim strY As String
'strY = `100.23"
'intX = strY      `Passes the string to a numeric
                  `Variable.
'List1.AddItem Cos (strY)    `Adds cosine of number in
                             ` the string to the list box.
'strY    Cos (strY)          `Passes cosine to the
                             `string variable.
'TextI. Text = strY          `String variable prints in
                              `the text box.
```

Visual Basic will automatically coerce the variables to the appropriate data type. You should use caution when exchanging strings and numbers; passing a non-numeric value in the string will cause a run-time error to occur.
if you have a variable that will contain simple true/false, yes/no, or on/off information, you can declare it to be of type Boolean. The default value of Boolean is False. In the following example, **b1nRunning** is a Boolean variable which stores a simple yes/no setting.

```
Dim bIn Running As Boolean
      `Check to see if the tape is running.
      If  Recorder Direction  = 1 Then
      bInRunning  = True
End if
```

The Date Data Type

Date and time values can be contained both in the specific Date data type and in Variant variables. The same general characteristics apply to dates in both types.
When other numeric data types are converted to Date, values to the left of the decimal represent date information, while values to the right of the decimal represent time. Midnight is 0, and midday is 0.5. Negative whole numbers represent dates before December 30, 1889

The Object Data Type

Object variables are stored as 32-bit (4-byte) addresses that refer to objects within an application or within some other application. A variable declared as Object is one that can subsequently be assigned (using the Set statement) to refer to any actual object recognized by the application.

```
Dim  objDb As Object
Set objDb = OpenDatabase ("c: \Vb5\BibLio.mdb")
```

When declaring object variables, try to use specific classes (such as TextBox instead of control or, in the preceding case, Database instead of object) rather than the generic Object.

Visual Basic can resolve references to the properties and methods of objects with specific types before you run an application. This allows the application to perform faster at run time. Specific classes are listed in the Object Browser.

When working with other applications' objects, instead of using a Variant or the generic Object, declare objects as they are listed in the Classes list in the Object Browser. This ensures that Visual Basic recognizes the specific type of object you're referencing^ allowing the reference to be resolved at run time.

Converting Data Types

Visual Basic provides several conversion functions you can use to convert values into a specific data type. To convert a value to Currency, for example, you use the CCur function:
*PayPerWeek = CCur (Hours * hourlyPay)*

Conversion

Function	Converts an expression to
Cbool	Boolean
Cbyte	Byte
Ccur	Currency
Cdate	Date
CDbl	Double
Cint	Integer
CLng	Long
CSng	Single
CStr	String
Cvar	Variant
CVErr	Error

Note: Values passed to a conversion function must be valid for the destination data type or an error occurs. *For example*, if you attempt to convert a Long to an Integer, the Long must be within the valid range for the Integer data type. For More Information See the Language Reference for a specific conversion function.

The Variant Data Type

A Variant variable is capable of storing all system-defined types of data. You don't have to convert between these types of data if you assign them to a Variant variable; Visual Basic automatically performs any necessary conversion. ***For example:***

```
'Dim SomeValue              `Variant by default.
'SomeValue = "17"           `SomeValue  contains "17" (a two-
                            `character string) .
'SomeValue = SomeValue - 15 `SomeValue now contains
                                    `The numeric Value  2.
'SomeValue = "U" & SomeValue        `SomeValue now contains
                                  '  "U2" (a  two- character string) .
```

While you can perform operations on Variant variables without much concern for the kind of data they contain, there are some traps you must avoid.

- If you perform arithmetic operations or functions on a Variant, the Variant must contain something that is a number. For details, see the section, "Numeric Values Stored in Variants," in "Advanced Variant Topics."
- If you are concatenating strings, use the & operator instead of the + operator. For details, see the section, "Strings Stored in Variants," in "Advanced Variant Topics." In addition to being able to act like the other standard data types, Variants can also contain three special values: Empty, Null, and Error.

converting real numbers to error values using the **CVErr** function

The Empty Value

Sometimes you need to know if a value has ever been assigned to a created variable. A Variant variable has the Empty value before it is assigned a value. The Empty value is a special value different from 0, a zero-length string ('' ''), or the Null value. You can test for the Empty value with the IsEmpty function:

If I sEmpt y (z) Then z = 0

When a Variant contains the Empty value, you can use it in expressions; it is treated as either 0 or a zero-length string, depending on the expression. The Empty value disappears as soon as any value (including 0, a zero-length string, or Null) is assigned to a Variant variable. You can set a Variant variable back to Empty by assigning the keyword Empty to the Variant.

The null Value

The variant data type can contain another special value: Null. Null is commonly used in database applications to indicate unknown or missing data. Because of the way it is used in database, Null has some unique characteristics:

- Expressions involving Null always result in Null. Thus, Null is said to "propagate" through expressions; if any part of the expression evaluates to Null, the entire expression evaluates to Null.
- Passing Null, a Variant containing Null, or an expression that evaluates to Null as an argument to most functions causes the function to return Null.
- Null values propagate through intrinsic functions that return Variant data types. You can also assign Null with the Null keyword:

Z = Null

You can use the IsNull function to test if a Variant variable contains Null.

```
If   IsNull (X) Arid IsNull (Y) Then
     Z = Null Else?
Else
     Z = 0
End If
```

If you assign Null to a variable of any type other than Variant, a trappable error occurs. Assigning Null to a Variant variable doesn't cause an error, and Null will propagate through expressions involving Variant variables (though Null does not propagate through certain functions). You can return Null from any Function procedure with a Variant return value. Variables are not set to Null unless you explicitly assign Null to them, so if you don't use Null in your application, you don't have to write code that tests for and handles it. For More Information For information on how to use Null in expressions, see "Null" in the Language Reference.

The Error-Value

In a Variant, Error is a special value used to indicate that an error condition has occurred in a procedure. However, unlike for other kinds of errors, normal application-level error handling does not occur. This allows you, or the application itself, to take some alternative based on the error value. Error values are created by

CHAPTER – 2

ADVANCED VARIANT TOPICS

Internal Representation of Values in Variants

Variant variables maintain an internal representation of the values that they store. This representation determines how Visual Basic treats these values when performing comparisons and other operations. When you assign a value to a Variant variable, Visual Basic uses the most compact representation that accurately records the value. Later operations may cause-Visual Basic to change the representation it is using for a particular variable. (A Variant variable is not a variable with no type; rather, it is a variable that can freely change its type) These internal representations correspond to the explicit data types discussed in "Data Types" earlier in this chapter.

Note: A variant always takes up 16 bytes, no matter what you store in it. Objects, strings, and arrays are not physically stored in the Variant; in these cases, tour bytes of the Variant are used to hold either an object reference, or a pointer to the string or array. The actual data are stored elsewhere.
Most of the time, you don't have to be concerned with what internal representation Visual Basic is using for a particular variable; Visual Basic handles conversions automatically. If you want to know what value Visual Basic is using, however, you can use the VarType function. For example, if you store values with decimal fractions in a Variant variable. Visual Basic always uses the Double internal representation, if you know that your application does not need the high accuracy (and slower speed) that a Double value supplies, you can speed your calculations by converting the values- to Single, or even to Currency:

If VarType (X) = 5 Then X = CSnq (X) `Convert to Single.

With an array variable, the value of VarType is the sum of the array and data type return values. ***For example,*** this array contains Double values:

```
Private Sub Form Click ( )
   Dim db1Sample (2) As Double
   MsgBox VarType (db1Sample)
End Sub
```

Future versions of Visual Basic may add additional Variant representations, so any code you write that makes decisions based on the return value of the VarType function should gracefully handle return values that are not currently defined.
***For More Information*:** For information about the VarType function, see "VarType Function" in the Language Reference. To read more about arrays, see "Arrays" later in this chapter.

Numeric Values Stored in Variants

When you store whole numbers in Variant variables, Visual Basic uses the most compact representation possible. For example, if you store a small number without a decimal fraction, the Variant uses an Integer representation for the value. If you then assign a larger number, Visual Basic will use a Long value or, if it is very large or has a fractional component, a Double value.

Sometimes you want to use a specific representation for a number. ***For example,*** you might want a Variant variable to store a numeric value as Currency to avoid round-off errors in later calculations. Visual Basic provides several conversion functions that you can use to convert values into a specific type (see "Converting Data Types" earlier in this chapter). So convert a value to Currency, ***for example***, you use the ***CCur*** function**:**

PayPerWeek = CCur (hours * hour l yPay)

An error occurs if you attempt to perform a mathematical operation or function on a Variant that does not contain a number or something that can be interpreted as a number. ***For example***, you cannot perform any arithmetic operations on the value U2 even though it contains a numeric character, because the entire value is not a valid number. *Likewise*, you cannot perform any calculations on the value 1040EZ; however, you can perform calculations on the values +10 or -1.7E6 because they are valid numbers. For this reason, you often want to determine if a Variant variable contains a value that can be used as a number The IsNumeric function performs this task**:**

```
Do
   anyNumber = Input Box ( "Enter a number")
Loop Until IsNumeric (anyNumber)
MsgBox "The square root is: "& Sqr (anyNumber)
```

When Visual Basic converts a representation that is not numeric (such as a string containing a number) to a numeric value, it uses the Regional settings (specified in the Windows Control Panel) to interpret the thousands separator, decimal separator, and currency symbol. Thus, if the country setting in the Windows Control Panel is set to United States, Canada, or Australia, these two statements would return true**:**

```
IsNumeric ("$ 100")
IsNumeric ("1,560. 50")
```

While these two statements would return false**:**

```
IsNumeric ("DM100")
IsNumeric ("I .560, 50")
```

However, the reverse would be the case — the first two would return false and the second two true — if the country setting in the Windows Control Panel was set to Germany.
If you assign a Variant containing a number to a string variable or property, Visual Basic converts the representation of the number to a string automatically. If you want to explicitly convert a number to a string, use the ***CStr*** function. You can also use the Format function to convert a number to a string that includes formatting such as currency, thousands separator, and decimal separator symbols. The Format function automatically uses the appropriate symbols according to the Regional Settings Properties dialog box in the Windows Control Panel.
For More Information: See *"Format Function"* and topics about the conversion functions in the Language Reference. For information on writing code for applications that will be distributed in foreign markets, see *"International Issues."*

Strings Stored in Variants

Generally, storing and using strings in Variant variables poses few problems. As mentioned earlier, however, sometimes the result of the + operator can be ambiguous when used with two Variant values. If both of the Variants contain numbers, the + operator performs addition. If both of the Variants contain strings, then the + operator performs string concatenation. But if one of the values is represented as a number and the other is represented as a string, the situation becomes more complicated. Visual Basic first attempts to convert the string into a number. If the conversion is successful, the + operator adds the two values; it unsuccessful, it generates a **Type mismatch** error. To make sure that concatenation occurs, regardless of the representation of the value in the variables, use the & operator. ***For example***, the following code:

```
Sub Form__Click ()
    'Dim X,    Y
    X = "6"
    Y = "7"
    Print X + Y, X & Y
    X = 6
    Print X + Y, X & Y
End Sub
```

produces this result on the form**:**

```
13      67
13      67
```

Note: Visual Basic stores strings inter "Internally as Unicode. For more information on Unicode, see "International Issue."

Date/Time Values Stored in Variants

Variant variables can also contain date/time values several functions return date/time values. ***For example***, DateSerial returns the number of days left in the year**:**

```
Private Sub Form Click ( )
    Dim rightnow,  daysleft,  hourleft,  minutesleft
    Rightnow = Now 'Now returns the current date/time.
    'Daysleft = Int (DateSerial (Year (right now)
    (+ 1, 1, 1 - rightnow)
    hoursleft = 24 - Hour (rightnow)
    minutesleft = 60 - Minute (rightnow)
    Print daysleft & "days left in year."
    Print hoursleft & " hours left in the day."
    Print minutesleft & "minutes left in the hour."
End Sub
```

You can also perform math on date/lime values. Adding or subtracting integers adds or subtracts days; adding or subtracting fractions adds or subtracts time. Therefore, adding 20 add 20 days, while subtracting 1/24 subtracts one hour. The range for dates stored in Variant variables is January 1, 0100, to December 31, 9999. Calculations on dates don't take into account the calendar revisions prior to the switch to the Gregorian calendar, however, so calculations producing date values earlier than the year in which the Gregorian calendar was adopted (1752 in Britain and its colonies at that time; earlier or later in other countries) will be incorrect.

You can use date/time literals in your code by enclosing them with the number sign (#), in the same way you enclose string literals with double quotation marks (""). For example, you can compare a Variant containing a date/time value with a literal date**:**

```
If SomeDate > #3/6/93# Then
```

Similarly, you can compare a date/time value with a complete date/time literal:

```
If SomeDate > #3/6/93 1:20pm# Then
```

If you do not include a time in a date/time literal, Visual Basic sets the time part of the value to midnight (the start of the day). If you do not include a date in a date/time literal, Visual Basic sets the date part of the value to December 30, 1899.

Visual Basic accepts a wide variety of date and time formats in literals. These are all valid date/time values**:**

```
SomeDate = #3-6-93 13:20#
SomeDate= #March 27, 1993 1: 20am#
SomeDate= #Apr-2-93#
SomeDate= #4 April 1993#
```

For More Information: For information on handling dates in international formats, see "International Issues."

In the same way that you can use the IsNumeric function to determine if a Variant variable contains a value that can be considered a valid numeric value; you can use the ***IsDate*** function to determine if a Variant contains a value that can be considered a valid date/time value.

You can then use the ***CDate*** function to convert the value into a date/time value. For example, the following code tests the Text property of a text box with IsDate. If the property contains text that can be considered a valid date, Visual Basic converts the text into a date and computes the days left until the end of the year:

```
Dim SomeDate, daysleft
If IsDate (Text1.Text) Then
SomeDate = Cdate (Text1.Text)
Daysleft = Date Seria1 (Year (Some Date) +
1, 1, 1) – SomeDate
Text2.Text = daysleft & "days left in the year."
Else
MsgBox Text1.Text & "is not a valid date."
End If
```

Objects Stored in Variants

Objects can be stored in Variant variables. This can be useful when you need to gracefully handle a variety of data types, including objects. For example, all the elements in an array must have the same data type. Setting the data type of an array to Variant allows you to store objects alongside other data types in an array.

Visual Basic Data Types

Data Type	Suffix	Example
Boolean	None	True
Integer	%	14
Long (Integer)	&	4532838
Single (Floating)	!	3.23
Double (Floating)	#	3.23463627281
Currency	@	$12.98
Date	None	12/30/99
Object	None	n/a
String	$	"Visual Basic 6"
Variant	None	any

To implicitly type a variable, use the corresponding suffix shown above in the data type table. ***For example,***

TextValue$ = "This is a string"
creates a string variable, while
Amount% = 300
creates an integer variable.

- There are many advantages to explicitly typing variables. Primarily, we insure all computations are properly done, mistyped variable names are easily spotted, and Visual Basic will take care of insuring consistency in upper and lower case letters used in variable names. Because of these advantages, and because it is good programming practice, we will explicitly type all variables.
- To explicitly type a variable, you must first determine its scope. There are four levels of scope:
  - Procedure level
  - Procedure level, static
  - Form and module level
  - Global level

- Within a procedure, variables are declared using the Dim statement:

Dim Mylnt as Integer
Dim MyDouble as Double
Dim MyString As String, YourString as String

Procedure level variables declared in this manner do not retain their value once a procedure terminates.

- To make a procedure level variable retain its value upon exiting the procedure, replace the Dim keyword with **Static:**

Static Mylnt as Integer
Static MyDouble as Double
Form (module) level variables retain their value and are available to all procedures within that form (module). Form (module) level variables are declared in the **declarations** part of the **general** object in the form's (module's) code window. The **Dim** keyword is used:

Dim Mylnt as Integer
Dim MyDate as Date

Global level variables retain their value and are available to all procedures within an application. Module level variables are declared in the declarations part of the general object of a module's code window. (It is advisable to keep all global variables in one module.) Use the Global keyword**:**

Global Mylnt as Integer
Global MyDate as Date

- What happens if you declare a variable with the same name in two or more places? More local variables shadow (are accessed in preference to) less local variables. ***For example,*** if a variable Mylnt is defined as Global in a module and declared local in a routine MyRoutine, while in MyRoutine, the local value of Mylnt is accessed. Outside MyRoutine, the global value of Mylnt is accessed.

Arrays

If you have programmed in other languages, you're probably familiar with the concept of arrays. Arrays allow you to refer to a series of variables by the same name and to use a number (an index) to tell them apart. This helps you create smaller and simpler code in many situations, because you can set up loops that deal efficiently with any number of cases by using the index number. Arrays have both upper and lower bounds, and the elements of the array are contiguous within those bounds. Because Visual Basic allocates space for each index number, avoid declaring an array larger than necessary.
Note: The arrays discussed in this section are arrays of variables, declared in code. They are different from the control arrays you specify by setting the Index property of controls at design time. Arrays of variables are always contiguous; unlike control arrays, you cannot load and unload elements from the middle of the array.
All the elements in an array have the same data type. Of course, when the data type is Variant, the individual elements may contain different kinds of data (objects, strings, numbers, and so on). You can declare an array of any of the fundamental data types, including user-defined types (described in the section, "Creating Your Own Data Types," in "More About Programming") and object variables (described in "Programming with Objects"). In Visual Basic there are two types of arrays: a fixed-size array which always remains the same size, and a dynamic array whose size can change at run-time.

Declaring Fixed-Size Arrays

There are three ways to declare a fixed-size array, depending on the scope you want the array to have:
• To create a public array, use the Public statement in the Declarations section of a module to declare the array.
• To create a module-level array, use the Private statement in the Declarations section of a module to declare the array.
• To create a local array, use the Private statement in a procedure to declare the array.

Setting Upper and Lower Bounds

When declaring an array, follow the array name by the upper bound in parentheses. The upper bound cannot exceed the range of a Long data type (-2,147,483,648 to 2,147,483,647). **For example**, these array declarations can appear in the Declarations section of a module**:**

Dim Counters (14) As Tnteger `15 elements.
Dim Sums (20) As Double ` 21 elements.

To create a public array, you simply use Public in place of Dim**:**

Public Counters (14) As Integer
Public Sums(20) As Double

The same declarations within a procedure use Dim**:**

Dim, Counters (14) As Integer
Dim. Sums (20) As Double

The first declaration creates an array with 15 elements, with index numbers running from 0 to 14. The second creates an array with 21 elements, with index numbers running from 0 to 20. The default lower bound is 0.
To specify a lower bound, provide it explicitly (as a Long data type) using the to keyword:

Dim Counters (1 To 15) As Integer
Dim Sums (100 To 120) As String

In the preceding declarations, the index numbers of counters range from 1 to 15, and the index.

Arrays that Contain Other Arrays

It's possible to create a Variant array, and populate it with other arrays of different data types. The following code creates two arrays, one containing integers and the other strings It then declares a third Variant array and populates it with the integer and string arrays.

```
Private Sub Command_Click ()
 Dim intX As Integer `Declare counter, variable.
          `Declare, and populate an integer array.
          Dim countersA (5) As Integer
          For intX = 0 To 4
                    countersA (intX)= 5
                    Next intX
 `Declare arid populate a string array.
           Dim counterB       (5) As Integer
                     For intX = 0 To 4
                              countersB (int.X) = "hello"
                    Next intex
 Dim. arrX(2) As Variant       ` Doclare a new t wo- members
                                    ` array.
    arraX (1)   =  countersA ( )          ` Populate the array with
                                                  ` others arrays.
    arrX (2 )   =  comilersB ( )
    MsgBox arrX (l) (2)                  ` Display a member of each
                                   ` array.
    MsgBox  arrX, (2) (3)
End Sub
```

Multidimensional Arrays

Sometimes you need to keep track of related information in an array. For example, to keep track of each pixel on your computer screen, you need to refer to its X and Y coordinates This can be done using a multidimensional array to store the values.
With Visual Basic, you can declare arrays of multiple dimensions. For example, the following statement declares a two-dimensional 10-by-10 array within a procedure**:**

Static MatrixA (9, 9) As Double

Either or both dimensions can be declared with explicit lower bounds**:**

Static MatrixA (1 To 10, 1 To 10) As Double

You can extend this to more than two dimensions. ***For example***:

Dim MultiD(3, 1 To 10, 1 To 15)

This declaration creates an array that has three dimensions with sizes 4 by 10 by 15. The total number of elements is the product of these three dimensions, or GOO Note When you start adding dimensions to an array, the total storage needed by the array increases dramatically, so use multidimensional arrays with care. Be especially careful with Variant arrays, because they are larger than other data types

Using Loops to Manipulate Arrays

You can efficiently process a multidimensional array by using nested For loops.
For example, these statements initialize every element in m.i t t ; ,..\ to a value based on its location in the array:

```
Dim. I As Integer, J As Integer
Static MatrixA (1 To 10, 1 To 10) As Double
For 1 = 1 To 10
     For J = 1 To 10
        MatrixA (1, J) = 1 * 10 + J
     Next J
Next 1
```

Dynamic Arrays
See Also
Sometimes you may not know exactly how large to make an array. You may want to have the capability of changing the size of the array at run time.
A dynamic array can be resized at any time. Dynamic arrays are among the most flexible and convenient features in Visual Basic, and they help you to manage memory efficiently.
For example, you can use a large array for a short time and then free up memory to the system when you're no longer using the array.
The alternative is to declare an array with the largest possible size and then ignore array elements you don't need. However, this approach, if overused, might cause the operating environment to run low on memory.

To create a dynamic array

Declare the array with a Public statement (if you want the array to be public) or Dim statement at the module level (if you want the array to be module level), or a Static or Dim statement in a procedure (if you want the array to be local) You declare the array as dynamic by giving it an empty dimension list.

Dim DynArray ()

Allocate the actual number of elements with a ReDim statement.

ReDim DynArray (X + 1)

The ReDim statement can appear only in a procedure. Unlike the Dim and Static statements, ReDim is an executable statement — it makes the application carry out an action at run time.
The ReDim statement supports the same syntax used for fixed arrays. Each ReDim can change the number of elements, as well as the lower and upper bounds, for each dimension. However, the number of dimensions in the array cannot change.

ReDim DynArray (4 to 12)

For example, the dynamic array **Matrix1** is created by first declaring it at the module level**:**

Dim Matrix1 () As Integer

A procedure then allocates space for the array**:**

```
Sub CalcValuesNow ()
        .
        .
        .
        ReDim Matrix1 (19, 29)
End Sub
```

The ReDim statement shown here allocates a matrix of 20 by 30 integers (at a total size of 600 elements). Alternatively, the bounds of a dynamic array can be set using variables**:**

ReDim Matrix1(X, Y)

Note: You can assign strings to resizable arrays of bytes. An array of bytes can also be assigned to a variable-length string. Be aware that the number of bytes in a string varies among platforms. On Unicode platforms the same string contains twice as many bytes as it does on a non-Unicode platform.

Preserving the Contents of Dynamic Arrays

Each time you execute the ReDim statement, all the values currently stored in the array are lost. Visual Basic resets the values to the Empty value (for Variant arrays), to zero (for numeric arrays), to a zero-length string (for string arrays), or to Nothing (for arrays of objects)
This is useful when you want to prepare the array for new data, or when you want to shrink.
The size of the array to take up minimal memory. Sometimes you may want to change the size of the array without losing the data in the array. You can do this by using ReDim with the Preserve keyword. ***For example***, you can enlarge an array by one element without losing the values of the existing elements using the Bound function to refer to the upper bound**:**

ReDim Preserve DynArray (UBound (DynArray) + 1)

Only the upper bound of the last dimension jn a multidimensional array can be changed when you use the Preserve keyword; if you change any of the other dimensions, or the lower bound of the last dimension, a run-time error occurs. Thus, you can use code like this:

ReDim Preserve Matrix (10, UBound (Matrix, 2) + 1)

But you cannot use this code**:**

ReDim Preserve Matrix (UBound (Matrix, 1) + 1, 10)

For More Information: For information about dynamic arrays, see "ReDim Statement" in the Language Reference. To learn more about object arrays, see "Programming with Objects". Forms or modules, you will also be prompted to save those files. I always use this for new projects.
There is a corresponding ***Open*** command under the ***File*** menu to open project files.

CHAPTER – 3

VISUAL BASIC STATEMENTS AND EXPRESSIONS

- The simplest statement is the ***assignment statement***. It consists of a variable name, followed by the assignment operator (=), followed by some sort of ***expression***.

Examples:

```
StartTime = Now
    Explorer. Caption = "Captain Spaulding"
    BitCount = ByteCount * 8
    Energy = Mass * LIGHTSPEED ^ 2
    NetWorth = Assets - Liabilities
```

The assignment statement stores information.

- Statements normally take up a single line with no terminator. Statements can be stacked by using a colon (:) to separate them.

Example: **StartTime = Now : EndTime = StartTime + 10**

Be careful stacking statements, especially with If/End If structures (we'll learn about these soon). You may not get the response you desire.

- If a statement is very long, it may be continued to the next line using the ***continuation*** character, an underscore (_).

Example:
```
Month  = Log (Final * IntRate / Deposit +1)  _
 / Log(1 + IntRate)
```

- Comment statements begin with the keyword ***Rem*** or a single quote (‘).

For example:

```
Rem This  is  a remark
‘This  is also a remark
 x = 2 * y ‘ another  way  to write  a remark  or  comment
```

You as a programmer, should decide how much to comment your code. Consider such factors as reuse, your audience, and the legacy of your code.

Visual Basic Operators

The simplest ***operators*** carry out ***arithmetic*** operations. These operators in their order of precedence are**:**

Operators	*Operation*
^	**Exponentiation**
*/	**Multiplication and Division**
\	**Integer division (truncates decimal portion)**
Mod	**Modulus**
+ -	**Addition and Subtraction**

- ***Parentheses*** around expressions clan change precedence.
- To ***concatenate*** two strings, use the & symbol or the + symbol**:**

```
lblTime. Caption  =  “The current time is”  &  Format (Now,  “hh : mm”)
    textSample.Text  =  “Hook this “  +  “to this”
```

Be aware that I use both concatenation operators in these notes – I’m not very consistent (an old habit that’s hard to break).

There are ***six comparison operators*** in Visual Basic**:**

Operators	Comparison
>	**Greater than**
<	**Less than**
>=	**Greater than or equal to**
<=	**Less than or Equal to**
=	**Equal to**
<>	**Not equal to**

- The result of a comparison operation is a Boolean value (**True** or **False**).
- We will use ***three logical operators:***

Operator	Operation
Not	**Logical not**
And	**Logical and**
Or	**Logical or**

- The **Not** operator simply negates an operand. It is very useful for 'toggling' Boolean variables.
- The **And** operator returns a True if both operands are True. Else, it returns a False.
- The **Or** operator returns a True if either of its operands is True, else it returns a False.
- Logical operators follow arithmetic operators in precedence.

Introduction to Procedures

You can simplify programming tasks by breaking programs into smaller logical components. These components — called procedures — can then become building blocks that let you enhance and extend Visual Basic.
Procedures are useful for condensing repeated or shared tasks, such as frequently used calculations, text and control manipulation, and database operations. There are two major benefits of programming with procedures**:**

- Procedures allow you to break your programs into discrete logical units, each of which you can debug more easily than an entire program without procedures.
- Procedures used in one program can act as building blocks for other programs, usually with little or no modification. ***There are several types of procedures used in Visual Basic:***
- Sub procedures do not return a value.
- Function procedures return a value.
- Property procedures can return and assign values, and set references to objects.

Sub Procedures

A Sub procedure is a block of code that is executed in response to an event. By breaking the code in a module into Sub procedures, it becomes much easier to find or modify the code in your application.
The syntax for a Sub procedure is:

[Private | Public] [Static] Sub *procedurename (arguments)*
statements.
End Sub

Each time the procedure is called, the statements between Sub and End Sub are executed. Sub procedures can be placed in standard modules, class modules, and form modules. Sub procedures are by default Public in all modules, which means they can be called from anywhere in the application.

The ***arguments*** a procedure are like a variable declaration, declaring values that are passed in from the calling procedure.
In Visual Basic, it's useful to distinguish between two types of Sub procedures, general procedures and event ***procedures.***

General Procedures

A general procedure tells the application how to perform a specific task. Once a general procedure is defined, it must be specifically invoked by the application. By contrast, an event procedure remains idle until called upon to respond to events caused by the user or triggered by the system.

Why create general procedures? One reason is that several different event procedures might need the same actions performed. A good programming strategy is to put common statements in a separate procedure (a general procedure) and have your event procedures call it. This eliminates the need to duplicate code and also makes the application easier to maintain. For example, the VCR sample application uses a general procedure called by the click events for several different scroll buttons.

Event Procedures

\K/hen an object in Visual Basic recognizes that an event has occurred; it automatically invokes the event procedure using the name corresponding to the event. Because the name establishes an association between the object and the code, event procedures are said to be attached to forms and controls.

- An event procedure for a control combines the control's actual name (specified in the Name property), an underscore (_), and the event name. ***For instance***, if you want a command button named **cmdPlay** to invoke an event procedure when it is clicked, use the procedure **cmdPlay_Click.**
- An event procedure for a form combines the word **"Form,"** an underscore, and the event name. If you want a form to invoke an event procedure when it is clicked, use the procedure **Form_Click.** (Like controls, forms do have unique names, but they are not used in the names of event procedures.) If you are using the **MDI** form, the event procedure combines the word **"MDIForm,"** an underscore, and the event name, as in **MDIForm_Load**.

All event procedures use the same general syntax.

Syntax for a control event **Syntax for a form event**
Private Sub ***controlname_eventname (arguments)***
statements
End Sub **Private Sub Form_*evenfname (arguments)***
statements
End Sub

Although you can write event procedures from scratch, it's easier to use the code procedures provided by Visual Basic, which automatically include the correct procedure names.
You can select a template in the Code Editor window by selecting an object from the Object box and then selecting a procedure from the Procedure box. It's also a good idea to set the Name property of your controls before you start writing event procedures for them. !f you change the name of a control after attaching a procedure to it, you must also change the name of the procedure to match the new name of the control. Otherwise, Visual Basic won't be able to match the control to the procedure. When a procedure name does not match a control name, it becomes a general procedure.

Function Procedures

See also:
Visual Basic includes built-in, or intrinsic functions, like Sqr, Cos or Chr. In addition, you can use the Function statement to write your own Function procedures. The syntax for a Function procedure is**:**

[Private|Public][Static]Function *procedurename (arrangements) [As type]*
statements
End Function

Like a Sub procedure, a Function procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. Unlike a Sub procedure, a Function procedure can return a value to the calling procedure. There are three differences between Sub and Function procedures**:**

- Generally, you call a function by including the function procedure name and arguments on the right side of a larger statement or expression (*returnvalue* function()).
- Function procedures have data types, just as variables do. This determines the type of the return value. (In the absence of an As clause, the type is the default Variant type.)
- You return a value by assigning it to the procedurename itself. When the Function procedure returns a value, this value can then become part of a larger expression.

For example, you could write a function that calculates the third side, or hypotenuse, of a right triangle, given the values for the other two sides:

```
Function Hypotenuse (A  As Integer, B  As Integer)
As String
  Hypotenuse = Sqr(A ^ 2 + B ^ 2)
End Function
```

You call a Function procedure the same way you call any of the built-in functions in Visual Basic**:**

Labell.Caption = Hypotenuse (CInt (Text1.Text)
CInt (Text2.Text))
strX = Hypotenuse (Width, Height)

Working with Procedures

To create a new general procedure

- Type a procedure heading in the Code window and press ENTER. The procedure heading can be as simple as Sub or Function followed by a name. ***For example***, you can enter either of the following**:**

```
Sub UpdateForm ()
Function GetCoord ()
```

Visual Basic responds by completing the template for the new procedure.

Selecting Existing Procedures

To view a procedure in the current module

- To view an existing general procedure, select **"(General)"** from the Object box in the Code window, and then select the procedure in the Procedure box.

-or-

To view an event procedure, select the appropriate object from the Object box in the Code window, and then select the event in the Procedure box.

To view a procedure in another module

- From the View menu, choose ***Object Browser***.
- Select the project from the ***Project/Library*** box.
- Select the module from the ***Classes*** list, and the procedure from the ***Members of*** list.
- Choose ***View Definition***.

Calling Procedures

The techniques for calling procedures vary, depending on the type of procedure, where it's located, and how it's used in your application. The following sections describe how to call Sub and Function procedures.

Calling Sub Procedures

A Sub procedure differs from a Function procedure in that a Sub procedure cannot be called by using its name within an expression. A call to a Sub is a stand-alone statement. Also, a Sub does not return a value in its name as does a function. However, like a Function, a Sub can modify the values of any variables passed to it.
There are two ways to call a Sub procedure:

```
` Both of these statements Call a Sub named MyProc.
Call MyProc (FirstArgurnent, SecondArgurnent)
MyProc FirstArgument, SecondArgurrient
```

Note that when you use the Call syntax, arguments must be enclosed in parentheses. If you omit the Call keyword, you must also omit the parentheses around the arguments.

Calling Function Procedures

Usually, you call a function procedure you've written yourself the same way you call an intrinsic Visual Basic function like Abs; that is, by using its name in an expression**:**

```
` All of the following statements would call a function
` named ToDec.
Print .10 * ToDec
 X =- ToDec
If ToDec = 10 Then Debug. Print "Out of Range"
X = AnotherFunction(10 * ToDec)
```

It's also possible to call a function just like you would call a Sub procedure. The following statements both call the same function**:**

```
Call Year (Now)
Year Now
```

When you call a function this way, Visual Basic throws away the return value.

Calling Procedures in Other Modules

Public procedures in other modules can be called from anywhere in the project. You might need to specify the module that contains the procedure you're calling. The techniques for doing this vary, depending on whether the procedure is located in a form, class, or standard module.

Procedures in Forms

All calls from outside the form module must point to the form module containing the procedure. If a procedure named SomeSub is in a form module called Form1, then you can call the procedure in Form1 by using this statement**:**

```
Call  Form1. SomeSub (arguments)
```

Procedures in Class Modules

Like calling a procedure in a form, calling a procedure in a class module requires that the call to the procedure be qualified with a variable that points to an instance of the class. ***For example,*** DemoClass is an instance of a class named Class1:

```
Dim DemoClass. as New Class1
DemoClass.SomeSub
```

However, unlike a form, the class name cannot be used as the qualifier when referencing an instance of the class. The instance of the class must be first be declared as an object variables (in this case, DemoClass) and referenced by the variable name.
For More Information**:** You can find details on object variables and class modules in "Programming with Objects."

Procedures in Standard Modules

If a procedure name is unique, you don't need to include the module name in the call. A call from inside or outside the module will refer to that unique procedure. A procedure is unique if it-appears only in one place. If two or more modules contain a procedure with the same name, you may need to qualify it with the module name. A call to a common procedure from the same module runs the procedure in that

module. ***For example,*** with a procedure named CommonName in Module1 and Module2, a call to CommonName from Module2 will run the CommonName procedure in Module2, not the CommonName procedure in Module1.
A call to a common procedure name from another module must specify the intended module. ***For example,*** if you want to call the CommonName procedure in Module2 from Module1, use**:**

Module2.CommonName(arguments)

Passing Arguments to Procedures
See Also:
Usually the code in a procedure needs some information about the state of the program to do its job. This information consists of variables passed to the procedure when it is called when a variable is passed to a procedure, it is called an argument.

Argument Data Types

The arguments for procedures you write have the Variant data type by default. However, yea can declare other data types for arguments. For example, the following function accepts a string and an integer**:**

```
Function WhatsForLunch (WeekDay As String, Hour
As Integer) As String
    ` Returns a lunch menu based on the day and time.
   If WeekDay = "Friday" then
     WhatsForLunch = "Fish"
   Else
     WhatsForLunch = "Chicken"
   End If
   If Hour > 4 Then WhatsForLunch = "Too late"
End Function
```

Passing Arguments By Value

Only a copy of a variable is passed when an argument is passed by value. If the procedure changes the value, the change affects only the copy and not the variable itself. Use the **ByVal** keyword to indicate an argument passed by value. ***For example:***

```
Sub PostAccounts (ByVal intAcctNum as Integer)
.
.   Place statements here.
.
End Sub
```

This is especially useful if your procedures have several optional arguments that you do not always need to specify.

Determining Support for Named Arguments

To determine which functions, statements, and methods support named arguments, use the **AutoQuicklnfo** feature in the Code window, check the Object Browser, or see the Language Reference. Consider the following when working with named arguments**:**

- Named arguments are not supported by methods on objects in the Visual Basic (VB) object library. They are supported by all language keywords in the Visual Basic for applications (VBA) object library.
- in syntax, named arguments are shown as bold and italic. All other arguments are shown in italic only.

Important: You cannot use named arguments to avoid entering required arguments. You can omit only the optional arguments. For Visual Basic (VB) and Visual Basic for applications (VBA) object libraries, the Object Browser encloses optional arguments with square brackets [ ].

CHAPTER – 4

VISUAL BASIC FUNCTIONS

Visual Basic offers a rich assortment of built-in ***functions.*** The on-line help utility will give you information on any or all of these functions and their use. Some examples are**:**

Function	Value Returned
Abs	Absolute value of a number
Asc	ASCII or ANSI code of a character
Chr	Character corresponding to a given ASCII or ANSI code
Cos	Cosine of an angle
Format	Date or number converted to a text string
Instr	Locates a substring in another string
Left	Selected left side of a text string
Len	Number pf characters in a text string
Mid	Selected portion of a text string
Now	Current time and date
Right	Selected right end of a text string
Rnd	Random number
Sin	Sine of an angle
Sqr	Square root of a number

String Functions

- Visual Basic offers a powerful set of functions to work with string type variables, which are very important in Visual Basic. The Caption property of the label control and the Text property of the text box control are string types. You will find you are constantly converting string types to numeric data types to do some math and then converting back to display the information.
- To convert a string type to a numeric value, use the Val function. As an example, to convert the Text property of a text box control named **txtExample** to a number, use:

Val (txtExample.Text)

This result can then be used with the various mathematical operators.

- There are two ways to convert a numeric variable to a string. The Str function does the conversion with no regard for how the result is displayed. This bit of code can be used to display the numeric variable **MyNumber** in a text box control:

MyNumber = 3.14159
txtExample.Text = Str (MyNumber)

If you need to control the number of decimal points (or other display features), the Format function is used. This function has two arguments, the first is the number, the Second a string specifying how to display the number (use on-line help to see how these display specifies work). As an example, to display MyNumber with no more than two decimal points, use:

MyNumber = 3.14159
txtExample.Text = Format (MyNumber, "#.##")

In the display string **("#.##"),** the pound signs represent place holders.

- Many times, you need to extract substrings from string variables. There are three functions that help with this task. In the **Left** function, you can extract a specified number of 'left most' characters. This example extracts the 3 'left most' characters from the string variable**:**

MyString = "Visual Basic is fun!"
LeftString = Left(MyString, 3)

The **LeftString** variable is equal to **"Vis"**

- With the Right function, you can extract a specified number of 'right most1 characters. This example extracts the 6 'right most' characters from the string variable**:**

MyString = "Visual Basic is fun!"
RightString = Right (MyString, 6)

The **RightString** variable is equal to **"s fun!"**

- And, the **Mid** function lets extract a specified number of characters from anywhere in the string (you specify the string, the starting position and the number of characters to extract). This example extracts 6 characters from the string variable, starting at character 3**:**

MyString = "Visual Basic is fun!"
MidString = Mid(MyString, 3, 6)

The **MidString** variable is equal to **"sual B"**

- To determine how many characters are in a string variable, use the Len function. Or, for our example**:**

MyString = "Visual Basic is fun!"
LenString = Len(MyString)

LenString will have a value of **20**.

- To find a substring within a string variable, use the **Instr** function. Three arguments are used: starting position in Stringl (optional), String 1 (the variable), and String2 (the substring to find). The function will return the location of the first character of the substring (it will return 0 if the substring is not found). For our example**:**

MyString = "Visual Basic is fun!"
Location = Instr(3, MyString, "sic")

This says find the substring **"sic"** in **MyString,** starting at character 3 (if this argument is omitted, 1. is assumed). The returned **Location** will have a value of **10.**

- Another useful pair of functions are the **Asc** and **Chr** functions. These work with individual characters. Every 'typeable' character has a numeric representation called an ASCII **("askey")** code. The Asc function returns the ASCII code for an individual character**.** ***For example:***

Asc ("A")

returns the ASCII code for the upper case A (65, by the way). The Chr function returns the character represented by an ASCII code. For example:

Chr (48)

returns the character represented by an ASCII value of 48 **(a "1").** The Asc and Chr functions are used often in determining what a user is typing.

Rnd (Random Number) Function

- In writing games and learning software, we use the **Rnd** function to introduce random-ness. This insures different results each time you try a program. The Visual Basic function Rnd returns a single precision, random number between 0 and 1 (actually greater than or equal to 0 and less than 1). To produce random integers (I) between **Imin** and **Imax,** use the formula**:**

I = Int((Imax - Imin +1) * Rnd) + Imin

- The random number generator in Visual Basic must be seeded. A Seed value initializes the generator. The Randomize statement is used to do this:

Randomize Seed

If you use the same Seed each time you run your application, the same sequence of random numbers will be generated. To insure you get different numbers every time you use your application (preferred for games), use the Randomize statement without a seed (it will be seeded using the built-in **Timer** function)

Randomize

Place the above statement in the **Form_Load** event procedure.

- **Examples:**

To roll a six-sided die, the number of spots would be computed using**:**

NumberSpots = Int(6 * Rnd) + 1

To randomly choose a number between 100 and 200, use:

Number = Int(101 * Rnd) + 100

Example 2-1

Saving Account

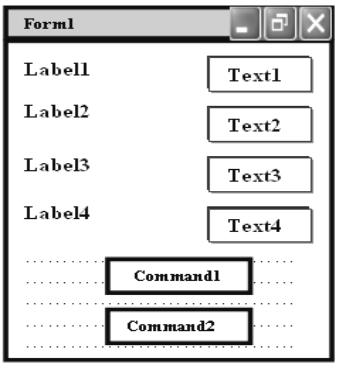
1. Start a new project. The idea of this project is to determine how much you save by making monthly deposits into a savings account. For those interested, the mathematical formula used is**:**

$$F = D [(1+I)^M -1)/I$$

where

F - Final amount
D - Monthly deposit amount
I - Monthly interest rate
M - Number of months

2. Place 4 label boxes, 4 text boxes, and 2 command buttons on the form. It should look something like this:



3. Set the properties of the form and each object.

Form1:	
BorderStyle	1-Fixed Single
Caption	Savings Account
Name	frmSavings
Label1:	
Caption	Monthly Deposit
Label2:	
Caption	Yearly Interest
Label3:	
Caption	Number of Months
Label4:	
Caption	Final Balance
Text1:	
Text	[Blank]
Name	txtDeposit
Text2:	
Text	[Blank]
Name	txtInterest
Form3:	
Text	
Name	[Blank] txtMonths
Text4:	
Text	[Blank]
Name	txtFinal
Command1:	
Caption	&Calculate
Name	cmdCalculate
Command2:	
Caption	E&xit
Name	cmdExit

Now your form should look like this-

4. Declare four variables in the general declarations area of your form. This makes them available to all the form procedures**:**

```
Option Explicit
'Dim Deposit As String
'Dim Interest As String
'Dim Months As String
'Dim Final As String
```

The **Option Explicit** statement forces us to declare all variables.

5. Attach code to the **cmdCalculate** command button Click event.

```
Private Sub cmdCalculate_Click ()
'Dim IntRate As String
'Read values from text boxes
'Deposit = Val (40,000)
'Interest = Val (12)
'IntRate = Interest / 1200
'Months = Val (3)
  'Compute final value and put in text box
Final = Deposit * ( (1 + IntRate) ^ Months - 1)/ IntRate)
txtFinal .Text = Format (Final, "# # # # #0.00" )
End Sub
```

This code reads the three input values (monthly deposit, interest rate, number of months) from the text boxes, converts those string variables to number using the **Val** function, converts the yearly interest percentage to monthly interest **(IntRate)** computes the final balance using the provided formula, and puts that result in a text box (after converting it back to a string variable).

6. Attach code to the cmdExit command button Click event.

```
Private Sub cmdExit_Click ()
End
End Sub
```

7. Play with the program. Make sure it works properly. Save the project (it is saved as **Example2-1** in the **LearnVB6/VB Code/Class 2** folder)

Visual Basic Symbolic Constants

- Many times in Visual Basic, functions and objects require data arguments that affect their operation and return values you want to read and interpret. These arguments and values are constant numerical data and difficult to interpret based on just the numerical value. To make these constants more understandable, Visual Basic assigns names to the most widely used values - these are called **symbolic constants**. Appendix I lists many of these constants.
- As an example, to set the background color of a form named **frmExample** to blue, we could type**:**

```
frmExample.BackColor = OxFFOOOO
```

or, we could use the symbolic constant for the blue color **(vbBlue)**:

```
frmExample.BackColor = vbBlue
```

- It is strongly suggested that the symbolic constants be used instead of the numeric values, when possible. You should agree that **vbBlue** means more than the value **OxFFOOOO**. When selecting the background color in the above example, You do not need to do any thing to define the symbolic constants - they are built into Visual Basic.

Defining Your Own Constants

- You can also define your own constants for use in Visual Basic. The format for defining a constant named PI with a value 3.14159 is:

```
Const P1 = 3.14159
```

- **User-defined constants:** should be written in all upper case letters to distinguish them from variables. The scope of constants is established the same way a variables' scope is. That is, if defined within a procedure, they are local to the procedure. If defined in the general declarations of a form, they are global to the form. To make constants global to an application, use the format**:**

```
Global Const PI = 3.14159
```

within the general declarations area of a module.

Visual Basic Branching - if Statements

- **Branching:** statements are used to cause certain actions within a program if a certain condition is met.

- The simplest is the single line If/Then statement:

```
If Balance - Check < 0 Then Print "You are overdrawn"
```

Here, if and only if Balance - Check is less than zero, the statement "You are overdrawn" is printed.

- You can also have If/Then/End If blocks to allow multiple statements.

```
If Balance - Check < 0 Then
Print "You are overdrawn"
  Print "Authorities have been notified"
End If
```

In this case, if Balance - Check is less than zero, two lines of information are printed.

Or,

If/Then/Else/End If blocks:

```
If Balance - Check < 0 Then
   Print "You are overdrawn"
   Print "Authorities have been notified'
Else
   Balance = Balance - Check
End If
```

Here, the same two lines are printed if you are overdrawn (Balance – check < 0), but, if you are not overdrawn **(Else)**, your new Balance is computed. Or, we can add the **Elself** statement**:**

```
If Balance - Check < 0 Then
    Print "You are overdrawn"
    Print "Authorities have been notified"
Elself Balance - Check - 0 Then
    Print "Whew! You barely made it"
    Balance = 0
Else
    Balance = Balance - Check
End If
```

Now, one more condition is added. If your Balance equals the Check amount (**ESself** Balance - Check = 0), a different message appears.

- In using branching statements, make sure you consider all viable possibilities in the If/ Else/End If structure. Also, be aware that each If and Elself in a block is tested sequentially. The first time an If test is met, the code associated with that condition is executed and the If block is exited. If a later condition is also True, it will never be considered.

Key Trapping

- Note in the previous example, there is nothing to prevent the user from typing in meaningless characters (for example, letters) into the text boxes expecting numerical data. Whenever getting input from a user, we want to limit the available keys they can press. The process of intercepting unacceptable keystrokes is key trapping.

- Key trapping is done in the **KeyPress** event procedure of a control. Such a procedure has the form (for a text box named **txtText) :**

```
Private Sub txtText_KeyPress (KeyAscii as Integer)
        .
        .
        .
End Sub
```

What happens in this procedure is that every time a key is pressed in the corresponding text box, the ASCII code for the pressed key is passed to this procedure in the argument. list (i.e. **KeyAscii**), If KeyAscii is an acceptable value, we would do nothing. However, if KeyAscii is not acceptable, we would set KeyAscii equal to zero and exit the procedure. Doing this has the same result of not pressing a key at all. ASCII values for all keys are available via the on-line help in Visual Basic. And some keys are also defined by symbolic constants. Where possible, we will use symbolic constants; else, we will use the ASCII values.
As an example, say we have a text box (named txtExample) and we only want to be able to enter upper case letters (ASCII codes 65 through 90, or, correspondingly, symbolic constants **vbKeyA** through **vbKeyZ**). The key press procedure would look like (the **Beep** causes an audible tone if an incorrect key is pressed) **:**

```
Private Sub txtExample_KeyPress (KeyAscii as Integer)
If KeyAscii >= vbKeyA And KeyAscii <= vbKeyZ Then
   Exit Sub
Else
   KeyAscii = 0
   Beep
End If
End Sub
```

In key trapping, it's advisable to always allow the backspace key (ASCII cod&8; symbolic constant **vbKeyBack**) to pass through the key press event. Else, you will not be able to edit the text box properly.

Example 2-2

Savings Account - Key Trapping

1. Note the acceptable ASCII codes are 48 through 57 (numbers), 46 (the decimal point), and 8 (the backspace key). In the code, we use symbolic constants for the numbers and backspace key.

Such a constant does not exist for the decimal point, so we will define one with the following line in the general declarations area**:**

```
Const vbKeyDecPt =46
```

2. Add the following code to the three procedures**:**

txtDeposit_KeyPress, **txtlnterest_KeyPress**, and **txtjMonths__KeyPress**

```
Private Sub txtDeposit KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
 If (KeyAscii >= vbKeyO And KeyAscii <= vbKey9) Or KeyAscii =
vbKeyDecPt  Or  KeyAscii = vbKeyBack  Then
        Exit Sub
    Else
        KeyAscii = 0
        Beep
    End If
    End Sub
    Private Sub txtlnterest KeyPress (KeyAscii As Integer)
   'Only allow number keys, decimal point, or backspace
    If (KeyAscii >= vbKeyO And KeyAscii <= vbKey9) Or KeyAscii =
    vbKeyDecPt Or KeyAscii = vbKeyBack Then
Exit Sub
Else
      KeyAscii  =  0
     Beep
End If
End Sub
Private Sub txtMonths_KeyPress  (KeyAscii As Integer)
` only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKeyO And KeyAscii <= vbKey9) Or KeyAscii =
vbKeyDecPt Or KeyAscii = vbKeyBack Then
    Exit Sub
Else
    KeyAscii = 0
    Beep
End If
End Sub
```

(In the If statements above, note the word processor causes a lime break where there really shouldn't be one. That is, there is no line break between the words **Or KeyAscii** and = **vbKeyDecPt**.

One appears due to page margins. In all code in these notes, always look for such things.)

3. Rerun the application and test the key trapping performance. Save the application (**Ex-ample2-2** in the **LearnVB6/VB Code/Class 2** folder)

Select Case - Another Way to Branch

- In addition to If/Then/Else type statements, the Select Case format can be used when there are multiple selection possibilities.

- Say we've written this code using the **If** statement:

```
    If Age  =  5  Then
      Category = "Five Year Old"
  Elself Age >= 13 and Age <= 19 Then
     Category = "Teenager"
  Elself (Age >= 20 and Age <= 35) Or Age = 50 Or (Age >= 60 and
  Age <= 65)  Then
       Category  =  "Special Adult."
 Elself Age > 65 Then
 Category = "Senior Citizen"
   Else
     Category = "Everyone Else"
   End If
```

The corresponding code with Select Case would be:

```
   Select Case Age
   Case 5
      Category = "Five Year Old"
   Case 13 To 19
      Category = "Teenager"
   Case 20 To  35, 50,  60 To  65
      Category = "Special Adult"
   Case Is > 65
      Category = "Senior Citizen"
    Case Else
      Category = "Everyone Else"
End Select
```

Notice there are several formats for the Case statement. Consult on-line help for discussions of these formats.

The GoTo Statement

- Another branching statement, and perhaps the most hated statement in programming, is the GoTo statement. However, we will need this to do Run-Time error trapping. The format is GoTo Label, where Label is a labeled line. Labeled lines are formed by typing the Label followed by a colon.
- **GoTo** Example**:**

```
Line10:
   .
   .
   .
GoTo Line10
```

When the code reaches the GoTo statement, program control transfers to the line labeled Line10.

CHAPTER – 5

VISUAL BASIC LOOPING

- Looping is done with the **Do/Loop** format. Loops are used for operations are to be repeated some number of times.
- Make sure you can always get out of a loop! Infinite loops are never nice. If you get into one, try ***Ctrl+Break***. That sometimes works - other times the only way out is rebooting your machine!
- The statement Exit Do will get you out of a loop and transfer program control to the statement following the Loop statement.

Visual Basic Counting

- Counting is accomplished using the **For/Next** loop.

Example:

```
For 1 = 1 to 50 Step 2
 A   =   I * 2
 Debug. Print A
Next 1
```

In this example, the variable I initializes at 1 and, with each iteration of the For/Next loop, is incremented by 2 (**Step**). This looping continues until **I** becomes greater than or equal to its final value (50). If Step is not included, the default value is 1. Negative values of Step are allowed.

- You may exit a For/Next loop using an Exit For statement. This will transfer program control to the statement following the **Next** statement.

Example 2-3

Saving Account - Decisions

1. Here, we modify the Savings Account project to allow entering any three values and computing the fourth. First, add a third command button that will clear all of the text boxes. Assign the following properties**:**

Commands3:

Caption	Clear &Boxes
Name	cmdClear

The form should look something like this when you're done:

- **Do While/Loop** *Example*:

```
Counter = 1
Do While Counter O 1000
    Debug. Print Counter
Counter = Counter + 1
Loop
```

This loop repeats as long as (**While**) the variable Counter is less than or equal to 1000. Note a Do While/Loop structure will not execute even once if the While condition is violated (False) the first time through. Also note the **Debug.Print** statement. What this does is print the value Counter in the Visual Basic Debug window We'll learn more about this window later in this course.

- **Do Until/Loop** *Example:*

```
Counter = 1
Do Until Counter  >  1000
     Debug.Print Counter
    Counter = Counter + 1
Loop
```

This loop repeats **Until** the Counter variable exceeds 1000. Note a Do Until/Loop structure will not be entered if the Until condition is already True on the first encounter.

- **Do/Loop While**

Example:

```
Sum = 1
Do
   Debug.Print Sum
   Sum = Sum + 3
 Loop While Sum <= 50
```

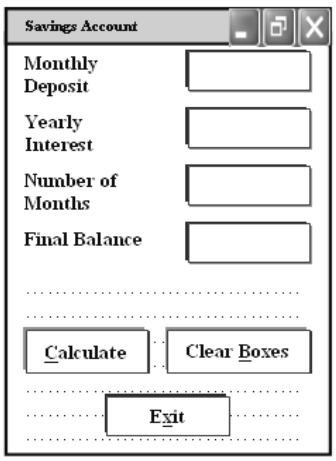
This loop repeats While the Variable Sum is less than or equal to 50. Note, since the While check is at the end of the loop, a Do/Loop While structure is always executed at least once.

- **Do/Loop Until**

Example:

```
Sum = 1
Do
   Debug. Print Sum
   Sum = Sum + 3
Loop Until Sum > 50
```

This loop repeats Until Sum is greater than 50. And, like the previous example, a Do/ Loop Until structure always executes at least once.



2. Code the **cmdClear** button **Click** event

```
    Private Sub cmdClear_Click ( )
    `Blank out the text boxes
txtDeposit.Text = ” ”
txtInterest. Text = “ "
txtMonths. Text = “ “
txtFinal. Text = " “
End Sub
```

This code simply blanks out the four text boxes when the Clear button is clicked.

3. Code the **KeyPress** event for the **txtFinal** object**:**

```
  Private Sub txtFinal_KeyPress (KeyAscii As Integer)
    `Only allow number keys, decimal point, or backspace
    If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii =
    vbKeyDecPt Or KeyAscii = vbKeyBack Then
        Exit Sub
    Else
        KeyAscii = 0
        Beep
  End If
  End Sub
```

We need this code because we can now enter information into the Final Value text box.

4. The modified code for the **Click** event of the **cmdCalculate** button is**:**

```
  Private Sub cmdCalculate_Click ( )
  Dim IntRate As String
  Dim IntNew As String
  Dim Fcn As Single, FcnD As String
  `Read the four text boxes
  Deposit = Val(txtDeposit.Text)
  Interest = Val (txtlnterest.Text)
  IntRate = Interest / 1200
  Months = Val (txtMonths . Text)
  Final = Val (txtFinal . Text)
  `Determine which box is blank
   `Compute that missing value and put in text box
   If Trim (txtDeposit.Text) = " “ Then
   `Deposit missing
     Deposit = Final / (((1 + IntRate) ^ Months - 1) / IntRate
txtDeposit. Text = Format (Deposit, "#####0.00")
Elself Trim (txtInterest . Text) = “ “ Then
'Interest missing - requires iterative solution
  IntNew = (Final / (0.5* Months * Deposit) - 1) / Months
 Do
    IntRate = Int New
    Fcn = (1 + InRate) ^ Months - Final * IntRate / Deposit - 1
    Fcnd = Months * (1 + InRate) ^ (Months - 1) / Deposit
    IntNew = IntRate - Fcn / Fcnd
Loop Until Abs(IntNew - IntRate) < 0.00001 / 12
Interest = IntNew * 1200
textInterest. Text = Format (Interest, “##0.00")
Elself Trim (txtMonths.Text) = " " Then
`Months missing
  Months = Log (Final * IntRate / Deposit + 1) / Log (1 + IntRate)
  txtMonths.Text = Format(Months, "###.0")
Elself Trim (txtFinal. Text) = "" Then
`Final value missing
  Final = Deposit * ((1 + IntRate) ^ Months - 1) / IntRate
  txtFinal.Text = Format(Final, "#####0.00")
End If
End Sub
```

In this code, we first read the text information from all four text boxes and based on which one is blank (the **Trim** function strips off leading and trailing blanks), compute the missing information and display it in the corresponding text box. Solving for missing **Deposit**, **Months**, or **Final** information is a straightforward manipulation of the equation given in Example 2-2.

If the **Interest** value is missing for the mathematically-inclined, we have to solve an Mth-order polynomial using something called Newton-Raphson iteration - a good example of using a Do loop. If you're not mathematically inclined, you should see that finding the Interest value is straightforward.

5. Test and save your application (**Example2-3** in the **LearnVB6/VB Code/Class 2** Folder) Go home and relax.

CHAPTER – 6

EXPLORING THE VISUAL BASIC TOOLBOX

Review and Preview

- In this class, we begin a journey where we look at each tool in the Visual Basic toolbox. We will revisit some tools we already know and learn a lot of new tools. First, though, we look at an important Visual Basic functions.

The Message Box

- One of the best functions in Visual Basic is the message box. The message box displays a message, optional icon, and selected set of command buttons. The user responds by clicking a button.
- The statement form of the message box returns no value (it simply displays the box):

MsgBox Message, Type, Title

where

Message	Text message to be displayed
Type	Type of message box (discussed in a bit)
Title	Text in title bar of message box

You have no control over where the message box appears on the screen

- The **function** form of the message box returns an integer value (corresponding to the button clicked by the user). Example of use (Response is returned value):

'Dim Response as Integer
'Response = MsgBox (Message, Type, Title)

- The **Type** argument is formed by summing four values corresponding to the buttons to display, any icon to show, which button is the default response, and the modality of the message box.
- The first component of the **Type** value specifies the **buttons** to display:

Value	Meaning	Symbolic Constar
0	OK button only	vbOKOnly
1	OK/Cancel buttons	vbOKCancel
2	Abort/Retry/Ignore buttons	vbAbortRetryIgnore
3	Yes/No/Cancel buttons	vbYesNoCancel
4	Yes/No buttons	vbYesNo
5	Retry/Cancel buttons	vbRetryCancel

- The second component of **Type** specifies the **icon** to display in the message box:

Value	Meaning	Symbolic Constan
0	No icon	(None)
16	Critical icon	vbCritical
32	Question mark	vbQuestion
48	Exclamation point	vbExclamation
64	Information icon	vbInformation

- The third component of **Type** specifies which button is **default** (i.e. pressing Enter is the same as clicking the default button):

Value	Meaning	Symbolic Constant
0	First button default	vbDefaultButton1
256	Second button default	vbDefaultButton2
512	Third button default	vbDefaultButton3

- The fourth and final component of **Type** specifies the modality:

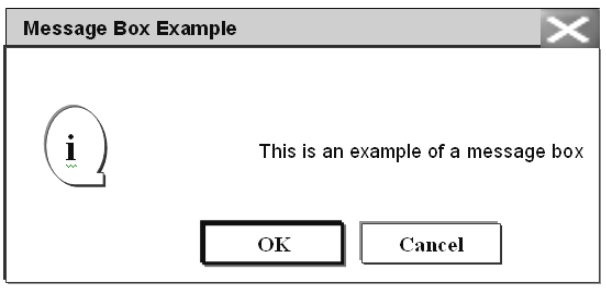
Value	Meaning	Symbolic Constant
0	Application modal	vbApplicationModal
4096	System modal	vbSystemModal

If the box is **Application Modal**, the user must respond to the box before continuing work in the current application. If the box is **System Modal**, all applications are suspended until the user responds to the message box.

- Note for each option in **Type**, there are numeric values listed and symbolic constants. Recall, it is strongly suggested that the symbolic constants be used instead of the numeric values. You should agree that **vbOKOnly** means more than the number 0 when selecting the button type.
- The value returned by the function form of the message box is related to the button clicked:

Value	Meaning	Symbolic Constant
1	OK button selected	vbOk
2	Cancel button selected	vbCancel
3	Abort button selected	vbAbort
4	Retry button selected	vbRetry
5	Ignore button selected	vbIgnore
6	Yes button selected	vbYes
7	No button selected	vbNo

Message Box Example:
MsgBox "This is an example of a message box", vbOKCancel + vbInformation, "Message Box Example"



- You've seen message boxes if you've ever used a Windows application Think of all the examples you've seen. For example, message boxes are used to ask you it you wish to save a file before exiting and to warn you if a disk drive is not ready.

Object Methods

- In previous work, we have seen that each object (control) has properties and events associated with it. A third concept associated with objects is the method A method is a procedure or function that imparts some action to an object.
- As we move through the toolbox, when appropriate, we'll discuss object methods. Methods are always enacted at run-time in code. The format for invoking a method is: **ObjectName.Method {optional arguments}**

  Note this is another use of the dot notation.

The Form Object

- The Form is where the user interface is drawn. It is central to the development of Visual Basic applications.

Form Properties:

Appearance	Selects 3-D or flat appearance.
BackColor	Sets the form background color.
BorderStyle	Sets the form border to be fixed or sizeable.
Caption	Sets the form window title.
Enabled	If True, allows the form to respond to mouse and keyboard events; if False, disables form.
Font	Sets font type, style, size.
ForeColor	Sets color of text or graphics.
Picture	Places a bitmap picture in the form.
Visible	If False, hides the form.

Form Events:

Activate	Form_Activate event is triggered when form becomes the active window.
Click	Form_Click event is triggered when user clicks on form.
DblClick	Form_DblClick event is triggered when user double-clicks on form.
Load	Form_Load event occurs when form is loaded. This is a good place to initialize variables and set any run-time properties

Form Methods:

Cls	Clears all graphics and text from form. Does not clear any objects.
Print	Prints text string on the form.

Examples

frmExample.CIs ` clears the form
frmExample.Print "This will print on the form"

Command Buttons

- We've seen the **command button** before. It is probably the most widely used control. It is used to begin, interrupt, or end a particular process.

Command Button Properties:

Appearance	Select 3-D or flat appearance
Cancel	Allows selection of button with **Esc** key (only one button on a form can have this property True).
Caption	String to be displayed on button.
Default	Allows selection of button with **Enter** key (only one button on a form can have this property True).
Font	Sets font type, style, size.

Command Button Events:

Click *Event triggered when button is selected either by clicking on it or by pressing the access key.*

Label Boxes

A label box is a control you use to display text that a user can't edit directly.

We've seen, though, in previous examples, that the text of a label box can be changed at runtime in response to events.

Label Properties:

Alignment	Aligns caption within border.
Appearance	Selects 3-D or flat appearance.
AutoSize	If True, the label is resized to fit the text specified by the caption property. If False, the label will remain the size , defined at design time and the text may be clipped.
BorderStyte	Determines type of border.
Caption	String to be displayed in box.
Font	Sets font type, style, size.
Wordwrap	Works in conjunction with AutoSize property. If AutoSize = True, Wordwrap = True, then the text will wrap and label will expand vertically to fit the Caption. If AutoSize ~ True, Wordwrap = False, then the text will not wrap and the label expands horizontally to fit the Caption. If AutoSize = False, the text will not wrap regardless of Wordwrap value.

Label Events:

Click	Event triggered when user clicks on a label
DblClick	Event triggered when user double-clicks on a label.

Text Boxes

A text box is used to display information entered at design time, by a user at run-time, or assigned within code. The displayed text may be edited.

Text Box Properties:

Appearance	Selects 3-D or flat appearance.
Border Style	Determines type of border.
Font	Sets font type, style, size.
MaxLength	Limits the length of displayed text (0 value indicates unlimited length).
MultiLine	Specifies whether text box displays single line or multiple lines.
PasswordChar	Hides text with a single character.
ScrollBars	Specifies type of displayed scroll bar(s)
SelLength	Length of selected text (run-time only).
SelStart	Starting position of selected text (run-time only).
SelText	Selected text (run-time only)
Tag	Stores a string expression.
Text	Displayed text.

Text Box Events:

Change Triggered every time the Text property changes.
LostFocus Triggered when the user leaves the text box. This is a good place to examine the contents of text box after editing.

KeyPress Triggered whenever a key is pressed. Used for key trapping, as seen in last class.

Text Box Methods:

SetFocus Places the cursor in a specified text box.

Example: txtExample.SetFocus' moves cursor to txtExample

• Use of the text box control should be minimized if possible. Whenever you give a user the option to type something, it makes your job as a programmer more difficult. You need to validate the information they type to make sure it will work with your code (recall the Savings Account example in the last class, where we need key trapping to insure only numbers were being entered). There are many controls in Visual Basic that are 'point and click,' that is, the user can make a choice simply by clicking with the mouse We'll look at such controls through the course. Whenever these 'point and click' controls can be used to replace a text box, do it!

Example 3-1

Password Validation

1. Start a new project. The idea of this project is to ask the user to input a password. If correct, a message box appears to validate the user. If incorrect, other options are provided.
2. Place a two command buttons, a label box, and a text box on your form so it looks something like this:

```
`This procedure checks the input password
Dim Response As Integer
If txtPassword.Text = txtPassword.Taq Then
`If correct, display message box
   MsgBox "You've passed security!", vbOKOnly + vbExclamation,
  "Access Granted"
Else
 `If incorrect, give option to try again
     Response = MsgBox ("Incorrect password", vbRetryCancel + vbCritical,      "Access Denied")
     If Response = vbRetry Then
       txtPassword.SelStart = 0
       txtPassword.SelLength = Len(txtPassword.Text)
       Else
      End
 End If
End If
txtPassword.SetFocus
EndSub
```

This code checks the input password to see if it matches the stored value. H so, it prints an acceptance message, if incorrect, it displays a message box to that effect and asks the user if they want to try again If Yes (Retry), another try is granted if No (Cancel), the program is ended. Notice the use of **ScILength** and **SelStart** to highlight an incorrect entry. This allows the user to type right over the incorrect response.

3. Attach the following code to the **Form_Activate** event.

```
Private Sub Form _Activate ()
txtPassword.SetFocus
End Sub
```

4. Attach the following code to the cmdExit_ Click event.

```
Private Sub cmdExit_Click ( )
End
End Sub
```

5. Try running the program. Try both options: input correct password (note it is case sensitive) and input incorrect password. Save your project (saved as **Exampte3-1** in the **LearnVB6/VB6 Code/Class 3** folder).
If you have time, define a constant, TRYMAX = 3, and modify the code to allow the user to have just TRYMAX attempts to get the correct password. After the final try, inform the user you are logging him/her off. You'll also need a variable that counts the number of tries (make it a Static variable).

Control Arrays

- With some controls, it is very useful to define control arrays - it depends on the application. For example, option buttons are almost always grouped in control arrays.
- Control arrays are a convenient way to handle groups of controls that perform a similar function. All of the events available to the single control are still available to the array of controls, the only difference being an argument indicating the index of the selected array element is passed to the event. Hence, instead of writing individual procedures for each control (i.e. not using control arrays), you only have to write one procedure for each array.
- Another advantage to control arrays is that you can add or delete array elements at runtime. You cannot do that with controls (objects) not in arrays. Refer to the **Load** and **Unload** statements in on-line help for the proper way to add and delete control array elements at run-time.
- Two ways to **create** a control array:

1. Create an individual control and set desired properties. Copy the control using the editor, then paste it on the form. Visual Basic will pop-up a dialog box that will ask you if you wish to create a control array. Respond yes and the array is created.

2. Create all the controls you wish to have in the array. Assign the desired control array name to the first control. Then, try to name the second control with the same name. Visual Basic will prompt you, asking if you want to create a control array. Answer yes. Once the array is created, rename all remaining controls with that name.

- Once a control array has been created and named, elements of the array are referred to by their name and index. For example, to set the **Caption** property of element **6** of a label box array named **IblExample**, we would use**:**

```
lblExample(6).Caption = "This is an example"
```

We'll use control arrays in the next example. Frames

Frames

- We've seen that both option buttons and check boxes work as a group. Frames provide a way of grouping related controls on a form. And, in the case of option buttons, frames affect how such buttons operate.

- To group controls in a frame, you first draw the frame. Then, the associated controls must be drawn in the frame. This allows you to move the frame and controls together. And, once a control is drawn within a frame, it can be copied and pasted to create a control array within that frame. To do this, first click on the object you want to copy.

Copy the object. Then, click on the frame. **Paste** the object. You will be asked if you want to create a control array. Answer **Yes**.

- Drawing the controls outside the frame and dragging them in, copying them into a frame, or drawing the frame around existing controls will not result in a proper grouping. It is perfectly acceptable to draw frames within other frames.

- As mentioned, frames affect how option buttons work. Option buttons within a frame work as a group, independently of option buttons in other frames. Option buttons on the form, and not in frames, work as another independent group. That is, the form is itself a frame by default. We'll see this in the next example.

- It is important to note that an independent group of option buttons is defined by physical location within frames, not according to naming convention. That is, a control array of option buttons does not work as an independent group just because it is a control array. It would only work as a group if it were the only group of option buttons within a frame or on the form. So, remember physical location and physical location only, dictates independent operation of option button groups.

Frames Properties:

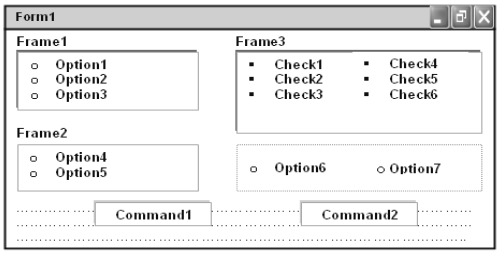
Caption	Title information at top of frame.
Font	Sets font type, style, size.

Example 3-2

Pizza Order

1. Start a new project. We'll build a form where a pizza order can be entered by simply clicking on check boxes and option buttons.

2. Draw three frames. In the first, draw three option buttons, in the second, draw two option buttons, and in the third, draw six check boxes. Draw two option buttons on the form. Add two command buttons. Make things look something like this.



3. Set the properties of the form and each control.

Form1:

BorderStyle	1-Fixed Single
Caption	Pizza Order
Name	frmPizza

Frame1:

Caption	Size

Frame2:

Caption	Crust Type

Frame3:

Caption	Toppings Option1
Caption	Small
Name	optSize
Value	True

Option1:

Caption	Medium
Name	optSize (yes, create a control array)

Option2:

Caption	Medium
Name	optSize (yes, create a control array)

Option3:

Caption	Large
Name	optSize

Option4:

Caption	Thin Crust
Name	optCrust
Value	True

Option5:

Caption	Thick Crust
Name	optCrust (yes, create a control array)

Option6:

Caption	Eat In
Name	optWhere
Value	True

Option7:

Caption	Take Out
Name	optWhere (yes,create a control array)

Check1:

Caption	Extra Cheese
Name	chk Top

Check2:

Caption	Mushrooms
Name	chkTop (yes, create a control array)

Check3:

Caption	Black Olives
Name	chkTop

Check4:

Caption	Onions
Name	chkTop

Check5:

Caption	Green Peppers
Name	chkTop

Check6:

Caption	Tomatoes
Name	chkTop

Command1:

Caption	&Build Pizza
Name	cmdBuild

Command2:

Caption	E&xit
Name	cmdBuild

4. Declare the following variables in the general declarations area:

```
Option Explicit
'Dim' PizzaSize As String
'Dim PizzaCrust As String
'Dim PizzaWhere As String
```

This makes the size, crust, and location variables global to the form.

5. Attach this code to the **Form_Load** procedure. This initializes the pizza size, crust, and eating location.

```
Private Sub Form_ Load()
'Initialize pizza parameters
'PizzaSize = "Small"
'PizzaCrust = "Thin Crust"
'PizzaWhere = "Eat In"
End Sub
```

Here, the global variables are initialized to their default values, corresponding to the default option buttons.

6. Attach this code to the three option button array Click events. Note the use of the index variable**:**

```
Private Sub optSize_Click (Index As Integer)
'Read pizza size
'PizzaSize = optSize (Index) .Caption
End Sub

Private Sub optCrust Click(Index As Integer)
'Read  Crust  Type
'PizzaCrust = optCrust(Index).Caption
End Sub

Private Sub optWhere_C1ick (Index As Integer)
'Read pizza eating location
'PizzaWhere = optWhere (Index).Caption
End Sub
```

In each of these routines, when an option button is clicked, the value of the corresponding button's caption is loaded into the respective variable.

7. Attach this code to the **cmdBuild_Click** event.

```
Private Sub cmdBuild_Click ()
'This procedure builds a message box that displays your pizza type
Dim Message As String
Dim I As Integer
Message = PizzaWhere + vbCr

'Message  =  Message  +  PizzaSize + " Pizza"   +  vbCr
'Message  =  Message + PizzaCrust + vbCr
For I = 0 To 5
        'If chkTop (I) .Value  =  vbChecked Then Message = Message +
   -chkTop(I).Caption + vbCr
    Next 1
  'MsgBox Message, vbOKOnly,   "Your Pizza"
  End Sub
```

This code forms the first part of a message for a message box by concatenating the pizza size, crust type, and eating location (**vbCr** is a symbolic constant representing a 'carriage return1 that puts each piece of ordering information on a separate line). Next, the code cycles through the six topping check boxes and adds any checked information to the message. The code then displays the pizza order in a message box

8. Attach this code to the **cmdExit_Click** event.

```
Private Sub cmdExit_Click ()
End
End Sub
```

9. Get the application working. Notice how the different selection buttons work in their individual groups. Save your project (saved as **Example3-2** in the **LearnVB6/VB Code/ Class 3** folder).

10. If you have time, try these modifications**:**

A. Add a new program button that resets the order form to the initial default values. You'll have to reinitialize the three global variables, reset all check boxes to unchecked, and reset all three option button groups to their default values.

B. Modify the code so that if no toppings are selected, the message "Cheese Only" appears on the order form. You'll need to figure out a way to see if no check boxes were checked.

List Boxes

• A list box displays a list of items from which the user can select one or more items. If the number of items exceeds the number that can be displayed, a scroll bar is automatically added.

▪ ***List Box Properties:***

Appearance List	Selects 3-D or flat appearance.
List	Array of items in. list box.
ListCount	Number of items in list.
ListIndex	The number of the most recently selected item in list. If no item is selected, ListIndex = 1.
MultiSelect	Controls how items may be selected (0-no multiple selection allowed). 1-multiple selection allowed, 2-group selection allowed).
Selected	Array with elements set equal to True or False, depending on whether corresponding list item is selected.
Sorted	True means items are sorted in 'Ascii' order; else items appear in order added.
Text	Text of most recently selected item.

• ***List Box Events:***

Click	Event triggered when item in list is clicked
DbIClick	Event triggered when item in list is double-clicked. Primary way used to process selection.

• ***List Box Methods:***

AddItem	Allows you to insert item in list.
Clear	Removes all items from list box.
RemoveItem	Removes item from list box, as identified by index of item to remove.

Examples

```
'ListExample.AddItem  "This is an added item"
'ListExample.Clear
'ListExampl e. Remove I tern  4  'removes   IstExample.List (4)
```

• Items in a list box are usually initialized in a Form_Load procedure. It's always a good idea to **Clear** a list box before initializing it.

• You've seen list boxes before. In the standard 'Open File' window, the Directory box is a list box with MultiSelect equal to zero.

Combo Boxes

• The combo box is similar to the list box. The differences are a combo box includes a text box on top of a list box and only allows selection of one item. In some cases, the user can type in an alternate response.

• ***Combo Box Properties:***

Combo box properties are nearly identical to those of the list box, with the deletion of the MultiSelect property and the addition of a Style property.

Appearance	Selects 3-D or flat appearance.
List	Array of items in list box portion.
ListCount	Number of items in list.
ListIndex	The number of the most recently selected item in list. If no item is selected, ListIndex = 1.
Sorted	True means items are sorted in 'Ascii' order, else items appear in order added.
Style	Selects the combo box form. Style = 0, Dropdown combo; user can change selection. Style = 1, Simple combo; user can change selection (make sure to resize default box so dropdown area appears). Style = 2, Dropdown combo; user cannot change selection.
Text	Text of most recently selected item.

▪ *Combo Box Events:*

Click	Event triggered when item in list is clicked.
DbIClick	Event triggered when item in list is double-clicked. Primary way used to process selection.

▪ *Combo Box Methods:*

AddItem	Allows you to insert item in list.
Clear	Removes all items from list box.
RemoveItem	Removes item from list box, as identified by index of item to remove.

Example

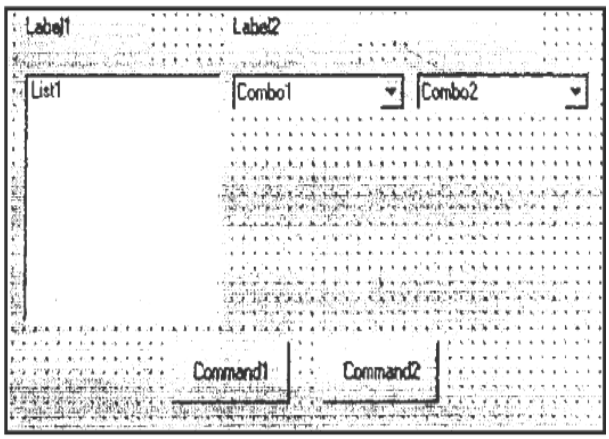
```
cboExample.AddItem  "This is an added item"
cboExample. Clear
cboExample.RemoveItem  4 ` removes cboExample.List (4)
```

• You've seen combo boxes before. In the standard 'Open File' window, the File Name box is a combo box of Style 2, while the Drive box is a combo box of Style 3.

Example 3-3

Flight Planner

1. Start a new project. In this example, you select a destination city, a seat location, and a meal preference for airline passengers.
2. Place a list box, two combo boxes, three label boxes and two command buttons on the form. The form should appear similar to this:



1. Set the form and object properties:

Form1:

BorderStyle	1-Fixed Single
Caption	Flight Planner
Name	frmFlight

List1:

Name	1stCities
Sorted	True

Combo1:

Name	cboSeat
Style	2-Dropdown List

Combo2:

Name	cboMeal
Style	1-Simple
Text	[Blank]

(After setting properties for this combo box, resize it until it is large enough to hold 4 to 5 entries.)

Label1:

Caption	Destination City

Label2:

Caption	Seat Location

Label3:

Caption	Meal Preference

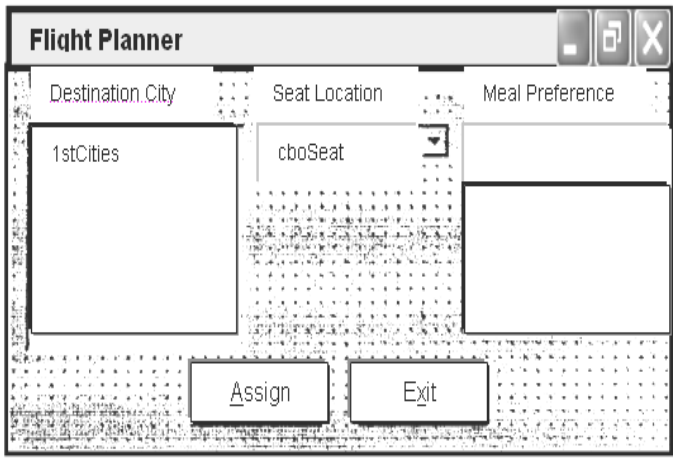
Command 1:

Caption	&Assign
Name	cmdAssign

Command2:

Caption	E&xit
Name	cmdExit

Now, the form should look like this:



1. Attach this code to the **Form_Load** procedure:

```
Private Sub Form_Load ()
'Add city names to list box
'ListCities.Clear
'ListCities.  AddItem "San Diego"
'ListCities.  AddItem "Los Angeles"
'ListCities.  AddItem "Orange County"
'ListCities.  AddItem "Ontario"
'ListCities.  AddItem "Bakersfield"
'ListCities.  AddItem "Oakland"
'ListCities.  AddItem "Sacramento"
'ListCities.  AddItem "San Jose"
'ListCities.  AddItem "San Francisco"
'ListCities.  AddItem "Eureka"
'ListCities.  AddItem "Eugene"
'ListCities.  AddItem "Portland"
'ListCities.  AddItem "Spokane"
'ListCities.  AddItem "Seattle"
             ListIndex = 0
'ListCities.
'Add seat types to first combo box
'cboSeat.AddItem "Aisle"
'cboSeat.AddItem "Middle"
'cboSeat.AddItem "Window"
'cboSeat.ListIndex = 0

'Add meal types to second combo box
'cboMeal.AddItem "Chicken"
'cboMeal.AddItem "Mystery Meat"
'cboMeal.AddItem "Kosher"
'cboMeal.AddItem "Vegetarian"
'cboMeal.AddItern "Fruit Plate"
'cboMeal.Text =  "No Preference"
End Sub
```

This code simply initializes the list box and the list box portions of the two combo boxes.

2. Attach this code to the **cmdAssign_Click** event:

```
Private Sub cmdAssign__Click ()
`Build message box that, gives your assignment
Dim Message As String
Message = "Destination: " + IstCities.Text + vbCr
Message = Message + "Seat Location: " + cboSeat.Text + vbCr
Message = Message + "Meal: " + cboMeal.Text + vbCr
MsgBox Message, vbOKOnly + vblnformation, "Your Assignment"
End Sub
```

When the **Assign** button is clicked, this code forms a message box message by concatenating the selected city (from the list box **1stCities**), seat choice (from **cboSeat**), and the meal preference (from **cboMeal**).

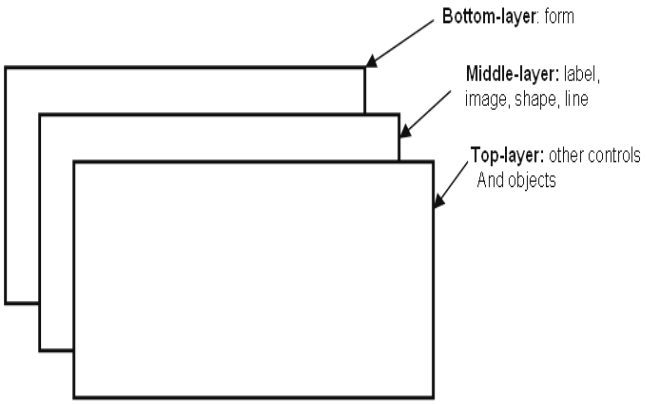
3. Attach this code to the cmdExit_Click event:

```
Private Sub cmdExit_Click ()
End
End Sub
```

4. Run the application. Save the project (saved as **Example3-3 in LearnVB6/VB Code/ Class 3** folder).

Display Layers

- In this class, we will look at our first graphic type controls: line tools, shape tools, picture boxes, and image boxes. And, with this introduction, we need to discuss the idea of display layers.
- Items shown on a form are not necessarily all on the same layer of display. A form's display is actually made up of three layers as sketched below. All information displayed directly on the form (by printing or drawing with graphics methods) appears on the bottom-layer. Information from label boxes, image boxes, line tools, and shape tools, appears on the middle-layer. And, all other objects are displayed on the top-layer.



- What this means is you have to be careful where you put things on a form or something could be covered up. For example, a command button placed on top of it would hide text printed on the form. Things drawn with the shape tool are covered by all controls except the image box.
- The next question then is what establishes the relative location of objects in the same layer. That is, say two command buttons are in the same area of a form - which one lies on top of which one? The order in which objects in the same layer overlay each other is called the **Z-order**. This order is first established when you draw the form. Items drawn last lie over items drawn earlier. Once drawn, however, clicking on the desired object and choosing Bring to Front from Visual Basic's **Edit** menu can modify the Z-order. The **Send to Back** command has the opposite effect. Note these two commands only work within a layer; middle-layer objects will always appear behind top-layer objects and lower layer objects will always appear behind middle-layer objects.

Line Tool

- The line tool creates simple straight line segments of various width and color. Together with the shape tool discussed next, you can use this tool to 'dress up' your application.
- *Line Tool Properties:*

BorderColor Determines the line color.
BorderStyle Determines the line 'shape'. Lines can be transparent, solid, Dashed, dotted, and combinations.
Border Width Determines line width

- There are no events or methods associated with the line tool.
- Since the line tool lies in the middle-layer of the form display, any lines drawn will be obscured by all controls except the shape tool, label box or image box.

Shape Tool

- The shape tool can create circles, ovals, squares, rectangles, and rounded squares and rectangles. Colors can be used and various fill patterns are available.
- *Shape Tool Properties:*

BackColor	**Determines the background color of the shape (only used when FillStyle not Solid.**
BackStyle	**Determines whether the background is transparent or opaque.**
BorderColor	**Determines the color of the shape's outline.**
BorderStyle	**Determines the style of the shape's outline. The border can be transparent, solid, dashed, dotted, and combinations.**
BorderWidth	**Determines the width of the shape border line.**
FillColor	**Defines the interior color of the shape.**
FillStyle	**Determines the interior pattern of a shape. Some choices are: solid, transparent, cross, etc.**
Shape	**Determines whether the shape is a square, rectangle, circle, or some other choice.**

- Like the line tool, events and methods are not used with the shape tool.
- Shapes are covered by all objects except perhaps line tools, label boxes and image boxes (depends on their Z-order) and printed or drawn information. This is a good feature in that you usually use shapes to contain a group of control objects and you'd want them to lie on top of the shape.

Horizontal and Vertical Scroll Bars

- Horizontal and vertical scroll bars are widely used in Windows applications. Scroll bars provide an intuitive way to move through a list of information and make great input devices.
- Both type of scroll bars are comprised of three areas that can be clicked, or dragged, to change the scroll bar value. Those areas are:

Clicking an **end arrow** increments the **scroll box** a small amount, clicking the **bar area** increments the scroll box a large amount, and dragging the scroll box (thumb) provides continuous motion. Using the properties of scroll bars, we can completely specify how one works. The scroll box position is the only output information from a scroll bar.

- *Scroll Bar Properties:*

LargeChange	Increment added to or subtracted from the scroll bar Value Property when the bar area is clicked.
Max	The value of the horizontal scroll bar at the far right and th value of the vertical scroll bar at the bottom. Can range frc -32,768 to 32,767.
Min	The other extreme value - the horizontal scroll bar at the le and the vertical scroll bar at the top. Can range from - 32,7 32,767.
SmallChange	The increment added to or subtracted from the scroll bar Value property when either of the scroll arrows is clicked.
Value	The current position of the scroll box (thumb) within the scr bar. If you set this in code, Visual Basic moves the scroll box to the proper position.

- A couple of important notes about scroll bars:

1. Note that although the extreme values are called **Min** and **Max**, they do not necessarily represent minimum and maximum values. There is nothing to keep the Min value from being greater than the Max value. In fact, with vertical scroll bars, this is the usual case. Visual Basic automatically adjusts the sign on the **SmallChange** and LargeChange properties to insure proper movement of the scroll box from one extreme to the other.
2. If you ever change the Value, Min, or Max properties in code, make sure Value is at all times between Min and Max or and the program will stop with an error message.

- *Scroll Bar Events:*

Change	Event is triggered after the scroll box's position has been modified. Use this event to retrieve the Value property alter an changes in the scroll bar.
Scroll	Event triggered continuously whenever the scroll box is being moved.

Example 4-1

Temperature Conversion

Start a new project. In this project, we convert temperatures in degrees Fahrenheit (set using a scroll bar) to degrees Celsius. As mentioned in the Review and Preview section, you should try to build this application with minimal reference to the notes. To that end, let's look at the project specifications.

Temperature Conversion Application Specifications

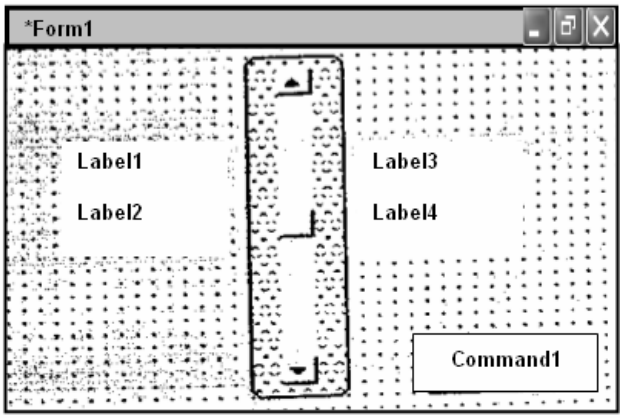
The application should have a scroll bar which adjusts temperature in degrees Fahrenheit from some reasonable minimum to some maximum. As the user changes the scroll bar value, both the Fahrenheit temperature and Celsius temperature (you have to calculate this) in integer format should be displayed. The formula for converting Fahrenheit (F) to Celsius (C) is.

$$C = (F - 32)*5/9$$

To convert this number to a rounded integer, use the Visual Basic **Clnt**() function. To change numeric information to strings for display in label or text boxes, use the **Str**() or **Format**() function. Try to build as much of the application as possible before looking at my approach. Try incorporating lines and shapes into your application if you can.

One Possible Approach to Temperature Conversion Application:

1. Place a shape, a vertical scroll bar, four labels, and a command button on the form./ Put the scroll bar within the shape - since it is in the top-layer of the form, it will lie in the shape. It should resemble this**:**



2. Set the properties of the form and each object.

Form1:

BorderStyle	1-Fixed Single
Caption	Temperature Conversion
Name	frmTemp

Shape1:

BackColor	White
BackStyle	1-Opaque
FillColor	Red
FillStyle	7-Diagonal
Shape	4-Rounded Rectangle

VScroll1:

LargeChange	10
Max	-60
Min	120
Name	vsbTemp
SmallChange	1
Value	32

Label 1:

Alignment	2-Center
Caption	Fahrenheit
FontSize	10
FontStyle	Bold

Label2:

Alignment	2-Center
AutoSize	True
BackColor	White
BorderStyle	1-FixedSingle
Caption	32
FontSize	14
FontStyle	Bold
Name	IbITempF

Label3:

Alignment	2-Center
Caption	Celsius
FontSize	10
FontStyle	Bold

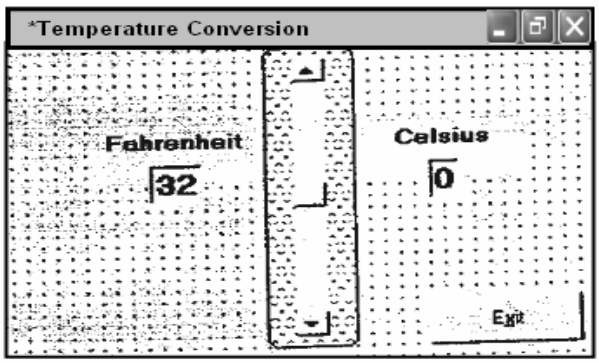
Label4:

Alignment	2-Center
AutoSize	True
BackColor	White
BorderStyle	1-Fixed Single
Caption	0
FontSize	14
FontStyle	Bold
Name	ibiTempC

Command1:

Cancel	True
Caption	E&xit
Name	cmdExit

Note the temperatures are initialized at 32F and OC, known values. When done, the form should look like this**:**



3. Put, this code in the general, decorations of your code window.

```
Option Explicit
Dim TempF As Integer
Dim TempC As Integer
```

This makes the two temperature variables global.

4. Attach the following code to the scroll bar Scroll event.

```
Private Sub vsbTemp_Scroll ()
'Read F  and  convert to C
'TempF = vsbTemp.Value
'IblTempF. Caption = Str (TempF)
'TempC = CInt ( (TempF - 32)  *  5   /  9)
'IblTempC.Caption = Str(TempC)
End  Sub
```

This code determines the scroll bar Value as it scrolls, takes that value as Fahrenheit temperature, computes Celsius temperature, and displays both values.

5. Attach the following code to the scroll bar **Change** event.

```
Private Sub vxbTemp_Change ( )
'Read  F and convert to C
'TempF = vsbTemp.Value
'IblTempF.Caption  =  Str (TempF)
'TempC  =  CInt ( (TempF  -  32)  *  5  /  9)
'IblTempC.Caption = Str (TempC)
End  Sub
```

Note: this code is identical to that used in the Scroll event. This is almost always the case when using scroll bars.

6. Attach the following code to the **crndExit_Click** procedure.

```
Private Sub cmdExit_Click ()
End
End sub
```

7. Give the program a try. Make sure it provides correct information at obvious points. For example,32 F better always be the same as **0 C!** Save the project (saved as Ex**-example4-1** in the **LearnVB6/VB Code/Class 4** folder) we'll return to it briefly in Class 5.

Picture Boxes

- The picture box allows you to place graphics information on a form. It is best suited for dynamic environments - for example, when doing animation.
- Picture boxes lie in the top layer of the form display. They behave very much like small forms within a form, possessing most of the same properties as a form.
- *Picture Box Properties:*

AutoSize	If True, box adjusts its size to fit the displayed graphic.
Font	Sets the font size, style, and size of any printing done in the picture box.
Picture	Establishes the graphics file to display in the picture box.

- *Picture Box Events:*

Click	Triggered when a picture box is clicked.
DblClick	Triggered when a picture box is double-clicked.

- *Picture Box Methods:*

Cls	Clears the picture box.
Print	Prints information to the picture box.

Examples

```
picExample.Cls  'clears the box picExample
picExample. Print  "a picture box"  `Print text string
```

- Picture Box LoadPicture Procedure**:**

An important function when using picture boxes is the LoadPicture procedure. It is used to set the Picture property of a picture box at run-time.

CHAPTER – 7

To BUILD CLIENT-SERVER DATABASE

Client-Server is the term for application software environment in which the application is split into two components: the client also called front-end which is the user interface half of the application; and the Server, also called the database server or back-end which contains access to the data.
In a non-client server system, the application is a single program that handles both the user interface and the database access the program.

REMOTE DATA CONTROL

Provides access to data stored in a remote ODBC data source through bound controls. The Remote Data Control enables you to move from row to row in a result set and to display and manipulate data from the rows in bound controls. With Remote Data Control, you can**:**

- Establish a connection to a data source based on its 'properties'.
- Pass any changes made to bound controls back to the data source.
- Pass the current row's data to corresponding bound controls.

The Remote Data Control automatically handles a number of contingencies including empty result sets, adding new rows, editing and updating existing rows, converting and displaying complex data types, and handling some types of errors. The Remote Data control behaves like the Jet-driven Data Control in most respects. The following guidelines illustrate few differences that apply when setting the SQL property.
The SQL property of the Remote Data Control like the Data Control Record Source property except that it cannot accept the name of a table by itself. The result set created by the Remote Data Control might not be in the same order as the Recordset created by the Data Control.

ResultSet Type Property

Returns or sets a value indicating the type of the ResultSet cursor created or to create.
***Syntax*:**

obj.ResultType= [value]

rdopenStatic	3	A static-type rdoResultSet
rdopenKeyset	1	A Keyset-type rdoResultSet. (Default)

DataSourceName Property

Returns or sets the DataSource name for a Remote Data Control.
Syntax:

obj.DataSourceName= [dataSourceName]

ResultSet

Retains or sets in rdoResultSet object defined by a Remote Data Control or as returned by the open ResultSet method.
***Syntax*:**

Set obj.Resultset = [value]

SQL

Returns or sets the SQL statement that defines the every executed by an rdo Query object or a Remote Data Control.
***Syntax*:**

obj.SQL = [value]

where Value' can be a valid SQL statement or a stored procedure.

USING THE RDC IN YOUR PROJECT

To use the RDC in you VBasic project, do the following:-

1. In the Visual Basic, select the menu command Project ->Components.
2. From the list of components, select Microsoft Remote Data Control. Also add a reference to the Microsoft Data Bound Grid Control.

3. Click OK, the Remote Data and Data Bound Grid Controls appear in the Visual Basic toolbox.
4. Use the toolbox to create instances of the RDC and the Data Bound Grid in the form in your project.
5. In the DataSourceName property of the RDC, pick the DSN for your database connection (Access, SQL Server, Oracle etc.). The DSN you created by the ODBC administrator.
6. Set the RDC SQL property to this: Select * from Authors
7. Set the DBGrid control's DataSource property to MSRDC, the name of the Remote Data Control.
8. Run the application, the application retrieves the authors table and displays it in the data grid.

Example

```
Dim en As New rdoConnection
en.Connect = "dsn=TestRDC;database=BIBLIO;uid=rdo;pwd=" en.EstablishConnection
Set MSRDC1.Resultset = en.QpenResultset ("select * from authors]")
```

Use the **RemoteData** control properties to describe the data source, establish a connection, and specify the type of cursor to create. If you alter these properties once the result set is created, use the **Refresh** method to rebuild the underlying **rdoResultset** based on the new property settings.

The **RemoteData** control behaves like the Jet-driven Data control in most respects. The following guidelines illustrate a few differences that apply when setting the **SQL** property.

You can treat the **RemoteData** control's **SQL** property like the **Data** control's **RecordSource** property except that it cannot accept the name of a table by itself, unless you populate the **rdoTables** collection first. Generally, the **SQL** property specifies an SQL query. For example, instead of just "Authors", you would code "SELECT*FROM AUTHORS" which provides the same functionality. However, specifying a table in this manner is not a good programming practice as it tends to return too many rows and can easily exhaust workstation resources or lock large segments of the database.
The result set created by the **RemoteData** control might not be in the same order as the **Recordset** created by the **Data** control. For example, if the **Data** control's **RecordSource** property is set to "Authors" and the **RemoteData** control's SQL property is set to "SELECT * FROM AUTHORS", the first record returned by Jet to the Data control is based on the first available index on the Authors table. The **RemoteData** control, however, returns the first row returned by the remote database engine based on the physical sequence of the rows in the database, regardless of any indexes. In some cases, the order of the records could be identical, but not always.

This difference in behavior can affect how bound controls handle the resulting rows — especially multiple-row bound controls like the **DataGrid** control. You can manipulate the **RemoteData** control with the mouse — to move the current row pointer from row to row, or to the beginning or end of the **rdoResultset** by clicking the control. As you manipulate the **RemoteData** control buttons, the current row pointer is repositioned in the **rdoResultset**. You cannot move off either end of the **rdoResultset** using the mouse. You also can't set focus to the **RemoteData** control.

You can use the objects created by the **RemoteData** control to create additional **rdoConnection**, **rdoResultset**, or **rdoQuery**objects.

You can set the **RemoteData** control **Resultset** property to an **rdoResultset** created independently of the control. If this is done, the **RemoteData** control properties are reset based on the new **rdoResultset** and **rdoConnection**.

You can set the **Options** property to enable asynchronous creation of the **rdoResultset** (**rdAsyncEnable**) or to execute the query without creating a temporary stored procedure (**rdExecDirect**).

The Validate event is triggered before each reposition of the current row pointer. You can choose to accept the changes made to bound controls or cancel the operation using the Validate event's action argument.

The **RemoteData** control can also manage what happens when you encounter an **rdoResultset** with no rows. By changing the **EOFAction** property, you can program the **RemoteData** control to enter **AddNew** mode automatically.

Using Programming Code

To create an **rdoResultset** programmatically with the **RemoteData** control:
Set the **RemoteData** control properties to describe the desired characteristics of the **rdoResultset**.

Use the Refresh method to begin the automated process or to create the new **rdoResultset**. Any existing **rdoResultset** is discarded.
All of the **RemoteData** control properties and the new **rdoResultset** object may be manipulated independently of the **RemoteData** controlwith or without bound controls. The **rdoConnection** and **rdoResultset** objects each have properties and methods of their own that can be used with procedures that you write.
For example, the **MoveNext** method of an **rdoResultset** object moves the current row to the next row in the **rdoResultset**. To invoke this method with an **rdoResultset** created by a RemoteData control, you could use this code**:**

```
RemoteData1 .Resultset.MoveNext
```

Remote Data Objects

Remote Data Objects provide for very fast ODBC access to manipulate ODBC databases.

The set of Remote Data Objects are referred to as RDO.To access any of these objects, you must include a reference to the objects by selecting the Project-> References menu option. This will bring up a dialog box. Select the entry Microsoft Remote Data Object

RDO is to an ODBC data source as Data Access Object (DAO) is to local data using the JET engine (Access). DAO is another connection method for database access. DAO is provided in all versins of Visual Basic and is much slower than RDO. The main difference is that RDO does not process any queries. It is equivalent to using the dbPassthrough option when executing a DAO query.

USING THE ADO DATA CONTROL

The ADO Data control uses Microsoft ActiveX Data Objects (ADO) to quickly create connections between data-bound controls and data providers. Data-bound controls are any controls that feature a DataSource property. Data providers can be any source written to the OLE DB specification. You can also easily create your own data provider using Visual Basic's class module.

Although you can use the ActiveX Data Objects directly in your applications, the ADO Data control has the advantage of being a graphic control (with Back and Forward buttons) and an easy-to-use interface that allows you to create database applications with a minimum of code.
Several of the controls found in Visual Basic's Toolbox can be data-bound, including the CheckBox, ComboBox, Image, Label, ListBox, PictureBox, and TextBox controls. Additionally, Visual Basic includes several data-bound ActiveX controls such as the DataGrid, DataCombo, Chart, and DataList controls. You can also create your own data-bound ActiveX controls, or purchase controls from other vendors.

The **ADO Data Control** is similar to the intrinsic **Data** control and the **Remote Data Control** (RDC). The **ADO Data Control** allows you to quickly create a connection to a database using **Microsoft ActiveX Data Objects** (ADO).

At design time, you can create a connection by setting the **ConnectionString** property to a valid connection string, then set the **RecordSource** property to a statement appropriate to the database manager.

You can also set the **ConnectionString** property to the name of a file that defines a connection; the file is generated by a **Data Link** dialog box, which appears when you click **ConnectionString** on the Properties window and then click either **Build** or **Select**.

Connect the **ADO Data Control** to a data-bound control such as the **DataGrid**, **DataCombo**, or **DataList** control by setting the **DataSource** property to the **ADO Data Control**.

At run time, you can dynamically set the **ConnectionString** and **RecordSource** properties to change the database. Alternatively, you can set the **Recordset** property directly to a previously-opened **recordset**.

CHAPTER – 8

DATA REPORTS

Data Report is a Report Designer that comes with the enterprise edition of Visual Basic 6.0, a fully integrated environment which has completely replaced Crystal reports of earlier versions. It is used to perform the following functions:

1. Design how the report looks
2. Add groupings that break every time the data in this group is changed.
3. Add subtotals based on these groupings, if desired.

The Report Designer designs in three steps:

1. Make connection to a database using Data Environment Designer.
2. Design the report layout in the data reports design environment.
3. Write code in your Visual Basic application that runs the report designed in the step 1.

WORKING WITH THE DATA ENVIRONMENT DESIGNER

Data Environment Designer (DED) is an upgraded version of VB5.0 Enterprise Edition's User Connection Designer for RDO. DED lets you to connect to any database having a native OLE DB data provider or, using OLE DB for ODBC provider. A single DED instance can support a multitude of independent database connections.

- When you open a new Data Project, a DED instance automatically appears and double clicking the DataEnvironment1 item opens the Data Project – Dataenvironment1 window.
- If you are opening the DataEvironment window on your Standard EXE project or Application Project then, you must choose it separately from the menu as "Project / More ActiveX Designers / Data Environment".

To make the first connection, right click Connection 1 and choose properties to open the provider page of Data Link Properties in which you select the OLE DB data provider. Clicking Next displays the Connection page in which you have to specify the location of database or server, the name of the database, login ID, and password. The database provider you choose determines the set of controls on the Connection page.

- Choose Microsoft Jet 4.0 OLE DB Provider.
- Now Click Test Connection to check whether the connection has succeeded.
- Click Ok
- Once a connection is specified choose "View / Data View Window" to display the members of the Data Environment Connections Collection.
- To add a Command Object to a selected connection, right click the connection and choose "Add Command". Select the class of database object (stored Procedure, Table, View, or Synonym for Access) in the Database Object list and then select an Object from the Object Name List. Alternatively, select the SQL option and type an SQL statement in the textbox. The Advanced page of CommandName Properties sheet lets you to choose between client - or - sever - side cursors, set the lock type, and specify other property values for the Command execution.

DED helps in minimizing the amount of code needed to establish database connections and execute queries.

GETTING STARTED

The report designer adapts a small subset of Visual Basic native bound controls to the data report page of the Tool Box: - RptLabel, RptTextBox, Rptlmage, RptLine and RptShape. A unique RptFunction control lets you to add calculated fields to the report.

- To start with select from menu "Project/Add Data Report".
- In the properties of DataReport1, choose DataSource as "DataEnvironmentl1 and in the DataMember choose "Command 1" (as created above).
- The default report form in design mode consists of Page Header, footer, detail sections.
- Once the properties are set, drag the "Command1 of the DataEnvironmentl" on the Detail section of the report.This step adds RptLabels and RptTextBoxes for the row values and automatically sets data member and datafields of the textboxes.
- Once the Details section is set you can the Page Header and Page Footer Sections by dragging the controls or the fields from the Coomand1 as per requirement. Its simple to size or align a report control as it is same as that in any other control. Setting the "DataFormat" property can do formatting of the report.

USING RPTFUNCTION CONTROL

The RptFunction Control can take place of the aggregate fields created by the grouped Command Objects. Table 9.1 enumerates the allowable values of the RptFunction Control.

Constant	Value	Aggregate
rptFuncSum	O	Sum
rptFuncAvg	1	Average
rptFuncMin	2	Minimize
rptFuncMax	3	Maximize
rptFuncRCnt	4	Count
rptFuncVCnt	5	N/A
rptFuncSDEV	6	Standard Deviation
rptFuncSERR	7	N/A

LINKING A REPORT TO YOUR APPLICATION

To link it to your application, Set a command Button on the Form which you want the Report to appear. Write " DataReport1.Show"in the event of the Button.

CHAPTER – 9

INTRODUCTION TO ActiveX PROGRAMMING

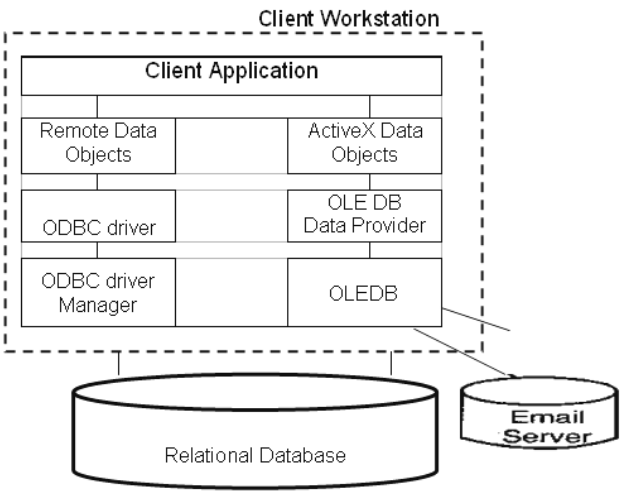
Building Visual Basic Applications with ActiveX Data Objects

Microsoft ActiveX Data Objects (ADO) is an object-oriented database access technology similar to Data Access Objects (DAO) and Remote Data Objects (RDO).

ADO is currently being positioned by Microsoft as a technique for accessing databases from a Web server. Because ADO is provided in the form of an ActiveX Server library (just as DAO and RDO are), you can use ADO in your Visual Basic application with no problem. In fact, in many ways, you'll find that it's easier to get a client/server database using ADO than the other alternatives discussed earlier.

Understanding the OLE OB/ADO Architecture

Most Visual Basic Developers never interact with OLE DB directly. Instead, they program against ActiveX Data Objects, the object model that provides interfaces into OLE DB. This architecture is illustrated in figure below:



There aren't as many OLE DB providers as there are ODBC drivers, but the number increased significantly when ADO 2.0 was released in 1998. This release, which is included in Visual Basic 6.0, includes native providers for SQL Server, Oracle, and Microsoft Jet/Access.

It's very likely that you will be able to get to a relational data source you prefer using ADO and OLE DB even if there aren't native OLE DB providers for it. This is because there is a generic OLE DB provider for ODBC relational databases.

Installing And Creating A Reference to ADO in your Visual Basic Application

Before you can begin working with ADO in your Visual Basic application, it must be installed on your computer. ADO is installed as part of the normal install of Visual Basic 6.0

If ADO is installed on your computer, you can begin using it by making a reference to the ADO library in your VB application, the same way you make references to the ADO or RDO libraries:

1. In your Visual Basic project, choose Project, References.
2. The Reference dialog box appears.
3. Check the box for Microsoft ActiveX Data Objects 2.0 library and then click OK.

You can now use ADO in your code.

USING ADO WITH OTHER DATA ACCESS OBJECT LIBRARIES

If you're creating an application designed to use ADO in conjunction with another data access object library, such as DAO, you need to be careful to differentiate between, for example, the DAO Recordset object and the ADO Recordset object. They aren't interchangeable.

If you have references to both DAO an ADO in your project and you create a Recordset variable, how do you know whether you have a DAO or ADO-style Recordset? The answer has to do with the order in which you added the reference to your project.

If you add a reference to the DAO library first, creating a Recordset object gives you a DAO-style Recordset; you need to use the full class name ADODB. Recordset to explicitly create an ADO-style Recordset. The spelled-out name of a class is also known as its *ProglD.*

If you don't want to make a direct reference to the object library in your code, you have an alternative. You can control which object library is accessed by default by using the priority setting in the References dialog box. For example, to give the DAO Object Library priority over the ADO Object Library, **do the following**:

1. in your VB project, choose Project, References.
2. References to both the Microsoft DAO 3.51 Object Library and Microsoft ActiveX Data Objects 2.0 Library should appear in the list of references (assuming they're installed on your computer).
3. Click (but don't uncheck) the reference to the DAO object library.
4. Click the upward-pointing arrow labeled priority. The reference to the DAO object library moves up in the list. This means that DAO will be used when you create an object (such as the Recordset object) that has the same name as an object in the ADO library.

Having control over the priority of the object models ensures that whenever you create a Recordset object variable in your code, it's a DAO Recordset, not an ADO Recordset.

Creating Both DAO and ADO Recordset Objects in Visual Basic Code using ProglD Syntax:

```
Option Explicit
' References DAO 3.51
' References ADO 2.0

'Dim db As DAO. Database

'Private adoRS As adodb.Recordset
'Private daoRS As DAO.Recordset
'Private en As adodb.Connection
'Dim strSQL As String

Private Sub Form_load()
    'Set cn  =  New adodb.Connection
    'strSQL =  "SELECT  *  FROM tblCustomer"
End Sub

Private Sub cmdShow_Click ()
    `     ** Create DAO recordset
    Set cn  =  OpenDatabase ("..\. . \DB\novelty.mdb")
    Set daoRS = db . OpenRecordset (strSQL)
    MsgBox "DAO query returns " & daoRS . Fields ("FifstName")

    ' ** Create ADO recordset
    Set  cn = New adodb. Connection
    cn.ConnectionString  =  "DSN=JetNovelty;"

    cn. Open
    Set   adoRS   =   en . Execute (strSQL)
    MsgBox   "ADO query returns" & adoRS. Fields (`FirstName' )
End  Sub
```

USING THE ADO CONNECTION OBJECT TO CONNECT TO A DATA SOURCE

In ActiveX Data Objects, you use the connection object to establish a connection to a data source. At the same time, as code examples later in this section demonstrate, you don't need to use a connection object to perform useful work with ADO-this aspect of ADO is one of its advantages over ADOs, which is far more dependent on the concept of a connection object.

You use the ADO connection object's open method to establish the connection with the data source. In order to tell ADO how to get to the data source, you must provide information in the form of a connect string identical to the ODBC connect string.

You use the connection object's connection string property to do this. You also have the option of choosing which provider you want to use by setting the connection object's provider property.

Specifying an OLE DB Provider and Connection String

You specify an OLE DB provider using the Provider property of the ADO connection object. This property tells ADO which OLE DB provider to use in order to execute commands against the server. I

f you don't specify a provider, or if you don't use a connection object, you get the default provider, which is the ODBC provider for OLE DB, also known as MSDASQL.

The common object's provider property is a text string that tells the connection which OLE DB provider to use. To use the ODBC provider for OLE DB, you don't need to specify a provider because the ODBC provider is the default. However, you can specify it for clarity if you want.

You use a connection string in ADO to provide information about how to connect to the database server. When you're using the ODBC provider for OLE DB, the connection string is the same as an ODBC connect string. This means that the exact information expected by the ODBC driver can vary from implementation to implementation. For other providers, the connection string can be of an entirely different syntax.

When you're using the ODBC provider, the connection String property can be a Data Source Name (DSN) or it can be a DSN-less connection. Here's an example of a connection to a database using the ODBC provider with a DSN**:**

```
Cn. Provider = "MSDASQL"
Cn.Connection String = "DSN=Novelty;"
```

Using a DSN in the connection string obviously requires that a DSN called Novelty must actually exist on the client computer.

Here's an example of the same connection with a DSN-less connection**:**

```
cn.provider = "MSDASQL"
cn.ConnectionString = "DRIVER={SQL Server};
DATABASE=novelty;
UID=randy;PWD=prince;"
```

WORKING WITH CURSORS

Just as with RJJO and DAO,ADO provides support for a number of types of cursors. In addition for providing support for navigating through the recordset one record at a time, different types of cursors permit you to control how the management of a recordset takes place.

You set the location of the cursor by assigning a value to the CursorLocation property of the Recordset object. Table below lists the types of cursors available with the ADO Connection object.

Cursor Type	Constant	Description
Client-side	adUseClient	Creates the cursor on the client-side
Server-side	adUseServer	Creates the cursor on the server

Choosing a client-side cursor means that ADO and OLE DB handle cursor operations. Client-side cursors often have abilities that aren't available on the server. For example, in ADO, you can create a disconnected recordset, which permits you to manipulate records without a persistent connection to the server. This capability is a function of the client-side cursor library. In ADO, the CursorLocation property is applicable to both the Recordset and Connection objects. If you assign the CursorLocation property of a Connection object, all recordsets you create from that connection have the same cursor location as its associated Connection object.In addition to specifying the cursor's location, you have the ability to create four different types of cursors in ADO. Your choice of cursor is generally governed by a balance between functionality and performance.

Opening and Closing a Connection Data Source

You specify a cursor type by assigning the CursorType property of the Recordset object. Table below lists the types of cursors you can create in ADO.

Cursor Type	Constant	Description
Forward-only	adOpenForwardOnly	No cursor at all - you can only move forward in the recordset; the Move Previous and Move First methods generate an error.
Keyset (Known in DAO as a dynaset	asOpenKeyset	You can't see records that have been added to the recordset by other users, but updates and deletes performed by other users do affect your recordset; can be the most efficient kind of cursor, particular when the recordset is large.
Dynamic	adOpenDynamic	You can see all changes to the data performed by other users while your recordset is open; this is usually the least efficient, but most powerful, type of cursor.
Static(Known in DAO as a snapshot)	adOpenStatic	A copy of all the data for a recordset; particularly useful when you're looking up data or running reports; can be very efficient when your recordset is small.

Of course, the reason you'd choose a forward-only cursor rather than a keyset or dynamic cursor is performance - if you're simply populating a list box or printing a list of items stored in the database, a forward-only cursor makes more sense and will give you better performance. Note that if the data provider can't create the specific type of cursor you ask for, it will create whatever type or cursor it can. It will not, generally, generate an error unless you attempt to do something that's specifically prohibited with the kind of cursor you have (for example, a Move Previous method on a forward - only cursor).

RECORD LOCKING IN ADO

As with other database access object models, ADO permits you to set different types of record-locking modes. You do this in situations where you need control over records locking modes. You do this in situations where you need control over how records are updated by multiple users in the database.

Constant	Description
adLockReadOnly	No updates to the recordset are permitted
adLockPessimistic	Pessimistic locking. Records in the recordset are locked when editing begins, and remain locked until you execute the update method or move onto another record.
adLockOptimistic	Optimistic locking. Records are locked only at the instant you execute the update method or move to another record.
adLockBatchOptimistic	Optimistic batch locking. Provides support for updating multiple records at once

It's extremely important to understand that the default lock method in ADO is adLockReadOnly. This is one of the most significant differences between ADO and DAO programming, since in DAO, recordsets are editable by default. This means that if you don't bother to set the LockType and CursorType properties, your ADO recordsets will always be read-only.

To issue commands to a data source using ADO, you open a connection to that data source. You typically do this using the ADO connection object's open method. When you're done with the data source, you close it using the connection object's close method. Here's the ADO connection object's open method syntax:

```
Cn.Open [connect], [userid], [password]
```

All the arguments to the open method are optional. If you don't supply a connection string as an argument to the open method, you can instead supply it using the ConnectionString property of the connection object. The effect is the same.

USING THE ADO RECORDSET OBJECT TO MANIPULATE DATA

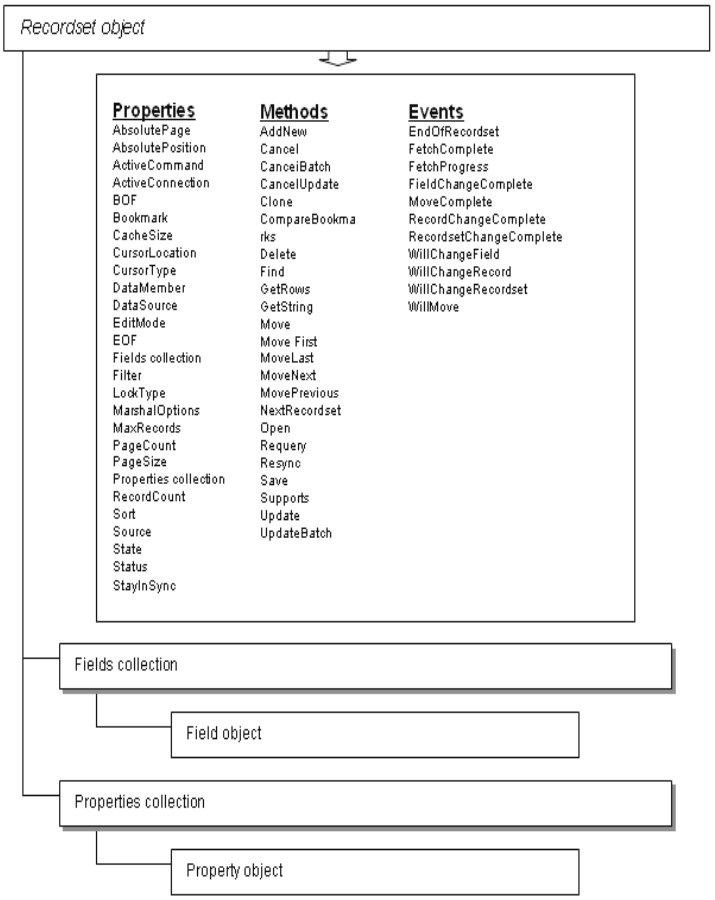
The ADO Recordset object, similar to DAO's recordset and RDO's rdoResultset object, is the way to access information retrieved from the data provider. The ADO Recordset has many of the same properties and methods as the other object models' recordset objects, so you can work with it the same way.

The position of the ADO Recordset object in the ADO object model, as well as its properties and methods.

The procedure for creating an ADO Recordset object is similar to creating an rdoResultset object in RDO. However, ADO adds an interesting twist: the ability to create a Recordset object that does not require an implicit connection object.

Creating an ADO Recordset Object Using a Connection and Recordset Object:

```
' References ADO 2.0 Private en As Connection

Private Sub Form_Load()
Set en = New ADODB.Connection
en.ConnectionString = *DSN=JetNovelty;" en.Open End Sub


Private Sub cmdQueryCN_Click()


Dim rs As ADODB.Recordset Set rs = New ADODB.Recordset


rs.Source = "select *" &_
"from tblCustomer * &_ "where State = 'DE' " &_ "order by LastName, FirstName"
      Set rs.ActiveConnection = en
      rs.Open

          lstData.Clear

                Do Until rs.EOF
        lstData.AddItem rs.Fields("FirstName")  & " " &_
                        rs.Fields("LastName") & " " &_
                        rs.Fields("Address")
            rs.MoveNext
               Loop

End Sub
Private Sub form_Unload (Cancel As Integer)
       cn.Close
       Set cn  =  Nothing
End   Sub
```



UPDATING AND INSERTING RECORDS USING THE RECORDSET OBJECT

Performing inserts and updates of records in ADO is almost precisely the same as in DAO. To insert a record, follow these steps:

1. Open a recordset.
2. Execute the AddNew method of the Recordset object
3. Assign values to the fields in the Recordset object.
4. Save the record by executing the update method of the Recordset object.

To update an existing record using the ADO Recordset object, follow these steps:

1. Open a recordset.
2. Assign values to the fields in the Recordset object. (Notice that you don't have to execute the Edit method of the Recordset as you did in DAO - ADO does away with that).
3. Save the record by executing the update method of the Recordset object.

The application that demonstrates entering and updating records permits the user to first populate a list box with customer data, and then select a particular customer to edit. Updating Records in ActiveX Data Objects:

```
Option Explicit
'References ADO 2.0

Private en As ADODB.Connection
Private mrsCust As ADODB.Recordset

Private Sub Form_Load()
  Set cn = New ADODB.Connection
  Cn.ConnectionString  =  "DSN=JetNovelty;"
  Cn.Open

    Set mrsCust = New ADODB.Recordset
    mrsCust.LockType  =  “adLockOptimistic
    mrsCust.CursorType  =  adOpen Keyset
End Sub

Private Sub cmdLlst__Cllck ( )

Dim  rs As ADODB . Recordset
Set   rs = New ADODB . Recordset

     rs. Source = "select *" &_
              "from tblCustomer”  &_
              "where State = 'DE"' &_
            "order by LastName, FirstName"


 Set rs.ActiveConnection = cn rs.Open


  1stData.Clear


  Do Until  rs. EOF
1stData.AddItem rs.Fields ("FirstName")& "
                                    rs.Fields("LastName")
 1stData.ItemData(IstData.NewIndex) = rs.Fields("ID")
 rs.. MoveNext
  Loop


rs.Close
Set   rs=Nothing


End Sub


Private Sub lstData_Click() Dim strSQL As String


   strSQL = "select *" &_
    "from tblCustomer" &_
            "where  ID=" & IstData. ItemData.(1stData. ListIndex)


mrsCust.Source = strSQL


  Set mrsCust.ActiveConnection = cn  mrsCust.Open
                txtFirstName.Text = mrsCust.Fields("FirstName")
                txtLastName.Text = mrsCust.Fields("LastName")
                txtAddress.Text = mrsCust.Fields("Address")
                txtCity.Text = mrsCust.Fields("City")

          mrsCustust.Close
End Sub

Private Sub cmdUpdate_Click ()
Dim strSQL As String


strSQL =  "select *" &_
"from tblCustomer" &_
                              "where ID =" & IstData. ItemData
(IstData.ListIndex)


l_Seats          Int
ll_Seats         Int
```

Train_Pass_Details

Field Name	Datatype	
Pass_id	Vachar(7)	Primary key to this table
Coach_No	Vachar(3)	
Seat_No	Int	
Date_of_Jounary	Determine	

Note: Pass_id of Train_Pass_Details is foreign key to Pass_id of Passenger_Details.

Develop an application

a. To list the passenger ids in the combo box
b. To display all the details of a given passenger, the train by which he is traveling, his coach number, seat number and ticket fare based on the class of his travel and the charges for that class for the train by which he is traveling.

Note: The data for all this should already be stored in the tables.

Controls	Property	Settings
Combobox	Name	cmbPassld
CommandButton	Name	cmdPassDetails
	Caption	List Passenger Details
TextBox	Name	txtid
TextBox	Name	txtname
TextBox	Name	txtsex
TextBox	Name	txtage
TextBox	Name	txtclass
TextBox	Name	txtfare
TextBox	Name	txtcoach
TextBox	Name	txtseat
TextBox	Name	txtjDate
TextBox	Name	txtTname

Also design labels for the respective fields.

Write a program in Visual Basic to make a math calculator.