VISUAL BASIC

ATMIYA

Atmiya Infotech Pvt. Ltd.

Contents

CHAPTER – 1 Introduction to VB

CHAPTER – 2 VB Fundamentals

CHAPTER – 3 Working with Forms

CHAPTER – 4 Working with Controls

CHAPTER – 5 Dialog Boxes and Menu Editor

CHAPTER – 6 Functions

CHAPTER – 7 Data Controls

CHAPTER – 8 Error Handling

CHAPTER – 9 ActiveX

CHAPTER – 10 File Handling

CHAPTER – 11 Reports and Graphs

CHAPTER – 12 WIN API, OLE

❑Features of Visual Basic

Data Access Features

ActiveX technologies

Internet capabilities

Rapid Application Development (RAD)

Support for multilingual applications

Interactive debugging

❑Editions of Visual Basic

Learning Edition

Professional Edition

Enterprise Edition

Concept of event-driven programming

Visual Basic is event-driven, meaning code remains idle until called upon to respond to some event (button pressing, menu selection...). An event processor governs Visual Basic. 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.

Procedural Programming vs. Event-Driven Programming

Event-driven programming can best be understood by contrasting it to procedural programming.

- Applications written in procedural languages execute by proceeding logically through the program code, one line at a time. Logic flow can be temporarily transferred to other parts of the program through the GoTo, GoSub, and Call statements, directing the program from beginning to end.

- In contrast, program statements in an event-driven application execute only when a specific event calls a section of code assigned to that event. Events can be triggered by keyboard input, mouse actions, the operating system, or code in the application. For example, consider what happens when the user clicks a command button named Command1 on a form. The mouse click is an event. When the Click event occurs, Visual Basic executes the code in the Sub procedure named Command1_Click. When the code has finished running, Visual Basic waits for the next event.

Difference between design time and run time

As with any programming language, using Visual Basic requires an understanding of some common terminology. The following table lists some key terms used in Visual Basic.

Term	Definition
Design time	Any time an application is being developed in the Visual Basic environment.
Run time	Any time an application is running. At run time, the programmer interacts with the application as the user would.
Forms	Windows that can be customized to serve as the interface for an application or as dialog boxes used to gather information from the user.
Controls	Graphic representations of objects, such as buttons, list boxes, and edit boxes, that user manipulate to provide information to the application.
Objects	A general term used to describe all the forms and controls that make up a program.

List the file types that can be included in a project

File type	Extension	Description
Form files	**.frm** **.frx**	hese files contain the form, the objects on the form, and code that runs when an event occurs on that form.
Visual Basic standard modules	**.bas**	These modules contain Sub and Function procedures that can be called by any form or object on a form. Standard modules are optional.
ActiveX Controls	**.ocx**	ActiveX controls are available from Microsoft and third party vendors. They are added using the Components command on the Project menu. Once added to a project, ActiveX controls appear in the Toolbox.
Visual Basic class modules	**.cls**	These modules contain the class definition, methods, and properties. Class modules are optional and are not covered in this course.

Resource files	**.res**	These files contain binary information used by the application. Resource files are typically used when creating programs for multiple languages.
User Controls	**.ctl** **.ctx**	These files contain source code and binary information for User Controls. These files are compiled into ActiveX controls (.ocx files).
User Documents	**.dob** **.dox**	These files contain base form and binary information for creating ActiveX documents.
ActiveX Designers	**.dsr**	These files contain information about designers that are used in the project. For example, you can create a Data Environment for run-time data access.

☐List the file types used in Graphics

BMP	Bitmap
GIF	Graphics Interchange Format
JPG	Joint Photographic Experts Group
DIB	Device Independent Bitmap

WMF	Windows Metafile
EMF	Enhanced Metafile
CUR	Cursors
ICO	Icons

☐Data Types

Basically Data Types can be divided into two category as

Native	Aggrigate
Boolean Byte Currency Date Double Integer Long Object Single String variant	Array (Static, Dynamic) UDT (User defined Type) Collection

☐Native Data Type

VAR. TYPE	BYTES	SUFFIX	RANGE
Boolean	2		True, False
Byte	1		0 to 255
Currency	8	@	-9.5808 E15 to 9.5807 E15
Date	8	#&ldots;..#	1 jan 100 to 31 dec 9999
Double	8	#	-1.012345678901234E308 to 1.79E308
Integer	2	%	-32,768 to 32,767
Long	4	&	-2E9 to 2E9
Object	4		N/A
Single	4	!	-3.123456E38 to 3.40E38
String	N/A	$	A variable length string can hold 2 GB. A Fixed length string can contain 64K character.
Variant			

☐The Integer Data Type

Integer variables can hold integer values (whole numbers) included

in the range from -32,768 through 32,767. These variables are also known as 16-bit integers because each value of this type takes 2 bytes of memory.

NOTE

> **You can indirectly specify that an undeclared variable is of type Integer by appending a % symbol to its name. However, this feature is supported by Visual Basic 6 only for compatibility with older Visual Basic and QuickBasic programs. All new applications should exclusively use variables declared in an explicit way. The same suggestion of course applies to other data types, including Long (&), Single(!), Double(#), Currency(@), and String($).**

All the integer constants in your code are implicitly of type Integer, unless their value is outside the range for this data type, in which case they are stored as Long.

The Long Data Type

Long variables can hold integer values in the range from -2,147,483,648 through 2,147,483,647 and are also known as 32-bit integers because each value takes 4 bytes of memory. As I mentioned previously, you're encouraged to use Longs in your applications as the preferred data type for integer values. Long variables are as fast as Integer variables, and in most cases they prevent the program from breaking when dealing with numbers larger than expected. One example is when you have to process

strings longer than 32,767 characters: In this case, you must use a Long index instead of an Integer variable. Watch out for this quirk when you convert code written for older Visual Basic versions.

CAUTION

> **For historical reasons, Visual Basic lets you enforce a particular data type as the default data type using the Deftype directive, so you might be tempted to use the DefLng A-Z directive at the beginning of each module to ensure that all undeclared variables are Long. My advice is: don't do that! Using Deftype directives instead of carefully declaring all your variables is a dangerous practice. Moreover, Deftype directives impair code reusability in that you can't safely cut and paste code from one module to another without also copying the directive.**

The Boolean Data Type

Boolean variables are nothing but Integers that can hold only values 0 and -1, which stand for False and True, respectively. When you use a Boolean, you are actually wasting 15 out of 16 bits in the variable, because this information could be easily held in one single bit. That said, I suggest you use Boolean instead of Integer variables whenever it makes sense to do so because this increases the readability of your code. On a few occasions, I have also experienced a slight improvement in performance, but usually it's negligible and shouldn't be a decisive factor.

The Byte Data Type

Byte variables can hold an integer numeric value in the range 0 through 255. They take only one byte (8 bits) each and are therefore the smallest data type allowed by Visual Basic. Visual Basic 4 introduced the Byte data type to ease the porting of 16-bit applications to Windows 95 and Microsoft Windows NT. Specifically, while Visual Basic 4 for the 32-bit platform and later versions are source-code compatible with Visual Basic 3 and Visual Basic 4 for the 16-bit platform applications, they store their strings in Unicode instead of ANSI format. This difference raised a problem with strings passed to API functions because Visual Basic 3 programmers used to store binary data in strings for passing it to the operating system, but the Unicode-to-ANSI automatic conversion performed by Visual Basic makes it impossible to port this code to 32-bit without any significant change.

The Single Data Type

Single variables can hold decimal values in the range from -3.402823E38 through -1.401298E-45 for negative values and 1.401298E-45 through 3.402823E38 for positive values. They take 4 bytes and are the simplest (and least precise) of the floating point data types allowed by Visual Basic.

Contrary to what many programmers believe, Single variables aren't faster than Double variables, at least on the majority of Windows machines. The reason is that on most systems, all floating point operations are performed by the math coprocessor, and the time spent doing the calculations is independent of the original format of the number. This means that in most cases you should go with Double values because they offer a better precision, a wider range,

fewer overflow problems, and no performance hit.

The Single data type is a good choice when you're dealing with large arrays of floating point values, and you can be satisfied with its precision and valid range. Another good occasion to use the Single data type is when you're doing intensive graphical work on your forms and in PictureBox controls. In fact, all the properties and methods that deal with coordinates-including CurrentX/Y, Line, Circle, ScaleWidth, ScaleHeight, and so on-use values of type Single. So you might save Visual Basic some conversion work if you store your coordinate pairs in Single variables.

The Double Data Type

Double variables can hold a floating point value in the range -1.79769313486232E308 through -4.94065645841247E-324 for negative values and 4.94065645841247E-324 through 1.79769313486232E308 for positive values. They take 8 bytes and in most cases are the preferable choice when you're dealing with decimal values. A few built-in Visual Basic functions return Double values. For example, the Val function always returns a Double value, even if the string argument doesn't include a decimal point. For this reason, you might want to store the result from such functions in a Double variable, which saves Visual Basic an additional conversion at run time.

The String Data Type

Visual Basic manages two different types of strings: conventional variable-length strings and fixed-length strings. You declare them in different ways:

```
Dim VarLenStr As String

Dim FixedLenStr As String * 40
```

The first, obvious difference is that in any given moment a variable-length string takes only the memory that it needs for its characters (actually, it takes 10 additional bytes for holding other information about the string, including its length), whereas a fixed-length string always takes a fixed amount of memory (80 bytes, in the preceding example).

String constants are enclosed within quotes, and you can embed quotes within the string by doubling them:

```
Print "<My Name Is ""Tarzan"">" ' displays <My Name Is "Tarzan">
```

Visual Basic additionally defines a number of intrinsic string constants, such as vbTab (the Tab character) or vbCrLf (the carriage return-line feed pair). Using these constants usually improves the readability of your code as well as its performance because you don't have to use a Chr function to create the strings.

The Currency Data Type

Currency variables can hold decimal values in a fixed-point format, in the range from -922,337,203,685,477.5808 through 922,337,203,685,477.5807. They differ from floating-point variables, such as Single and Double, in that they always include four decimal digits. You can think of a currency value as a big integer that's 8 bytes long and whose value is automatically scaled by a factor of 10,000 when it's assigned to the variable and when it's read back and displayed to the user.

☐The Date Data Type

Date variables can hold any date between January 1, 100, through December 31, 9999, as well as any time value. They take 8 bytes, exactly like Double variables. This isn't a casual resemblance because internally these date/time values are stored as floating-point numbers, in which the integer part stores the date information and the decimal part stores the time information. (For example, 0.5 means 12 A.M., 0.75 means 6 P.M., and so on.) Once you know how Date variables store their values, you can perform many meaningful math operations on them. For example, you can truncate date or time information using the Int function, as follows:

```
MyVar = Now ' MyVar is a Date variable.

DateVar = Int(MyVar) ' Extract date information.

TimeVar = MyVar - Int(MyVar) ' Extract time information.
```

You can also add and subtract dates, as you would do with numbers:

```
MyVar = MyVar + 7 ' Advance one week.

MyVar = MyVar - 365 ' Go back one (nonleap) year.
```

VBA provides many functions for dealing with date and time information in more advanced ways, which I'll cover in Chapter 5. You can also define a Date constant using the format #mm/dd/yyyy#, with or without a time portion:

```
MyVar = #9/5/1996 12.20 am#
```

The Object Data Type

Visual Basic uses object variables to store reference objects. Note that here we are talking about storing a reference to an object, not storing an object. The difference is subtle but important, and I'll talk about it at length in Chapter 6. There are several types of object variables, but they can be grouped in two broad categories: generic object variables and specific object variables. Here are a few examples:

```
' Examples of generic object variables

Dim frm As Form ' A reference to any form

Dim midfrm As MDIForm ' A reference to any MDI form

Dim ctrl As Control ' A reference to any control

Dim obj As Object ' A reference to any object

' Examples of specific object variables

Dim inv As frmInvoice ' A reference to a specific type of form

Dim txtSalary As TextBox ' A reference to a specific type of control

Dim cust As CCustomer ' A reference to an object defined by a

  ' class module in the current project

Dim wrk As Excel.Worksheet ' A reference to an external object
```

The most evident difference when dealing with object variables (as

opposed to regular variables) is that you assign object references to them using the Set keyword, as in the following code:

```
Set frm = Form1

Set txtSalary = Text1
```

CAUTION

> **One of the most common errors that programmers make when dealing with object variables is omitting the Set command during assignments. What happens if you omit this keyword depends on the object involved. If it doesn't support a default property, Visual Basic raises a compile-time error ("Invalid use of property"); otherwise, the assignment succeeds, but the result won't be the one you expect:**
>
> ```
> frm = Form1 ' A missing Set raises a compiler error.
>
> txtSalary = Text1 ' A missing Set assigns Text1's Text
> property to txtSalary's Text property.
> ```

Object variables can also be cleared so that they don't point to any particular object anymore. You do this by assigning them the special Nothing value:

```
Set txtSalary = Nothing
```

The Variant Data Type

Automatic data coercion is always dangerous because you might not get the results that you expect. For example, if you use the + operator on two Variants that hold numeric values, Visual Basic interprets the + as the addition operator. If both values are strings, Visual Basic interprets the + as the append operator. When one data type is a string and the other is a number, Visual Basic tries to convert the string to a number so that an addition can be performed; if this isn't possible, a "Type Mismatch" error is raised. If you want to be sure to execute an append operation regardless of the data types involved, use the & operator. Finally note that you can't store fixed-length strings in Variant variables.

Aggregate Data Types

The native data types we have examined so far have been simple. While useful in their own right, they can also serve as building blocks to form aggregate data types. In this section, we examine this concept more closely.

User-Defined Types

A user-defined type (UDT) is a compound data structure that holds several variables of simpler data types. Before you can use a UDT variable, you must first define its structure, using a Type directive in the declaration section of a module:

```
Private Type EmployeeUDT
  Name As String
  DepartmentID As Long
  Salary As Currency
End Type


Dim Emp As EmployeeUDT
Emp.Name = "Roscoe Powell"
Emp.DepartmentID = 123
```

Type structures can also contain substructures, for example:

Private Type LocationUDT
  Address As String
  City As String
  Zip As String
  State As String * 2
End Type
	Private Type EmployeeUDT
	  Name As String
	  DepartmentID As Long
	  Salary As Currency
	  Location As LocationUDT
	End Type

When you access such nested structures, you can resort to the With&ldots;End With clause to produce more readable code:

```
With Emp

    Print .Name

    Print .Salary

    With .Location

        Print .Address

        Print .City & " " & .Zip & " " & .State

    End With

End With
```

Visual Basic lets you copy one UDT to another UDT with the same structure using a regular assignment, as in the following code:

```
Dim emp1 As EmployeeUDT, emp2 As EmployeeUDT

...

emp2 = emp1
```

Arrays

Arrays are ordered sets of homogeneous items. Visual Basic

supports arrays made up of elementary data types. You can build one-dimensional arrays, two-dimensional arrays, and so on, up to 60 dimensions. (I never met a programmer who bumped into this limit in a real application, though.)

Static and dynamic arrays

Basically, you can create either static or dynamic arrays. Static arrays must include a fixed number of items, and this number must be known at compile time so that the compiler can set aside the necessary amount of memory. You create a static array using a Dim statement with a constant argument:

```
' This is a static array.

Dim Names(100) As String
```

Visual Basic starts indexing the array with 0. Therefore, the preceding array actually holds 101 items.

Most programs don't use static arrays because programmers rarely know at compile time how many items you need and also because static arrays can't be resized during execution. Both these issues are solved by dynamic arrays. You declare and create dynamic arrays in two distinct steps. In general, you declare the array to account for its visibility (for example, at the beginning of a module if you want to make it visible by all the procedures of the module) using a Dim command with an empty pair of brackets. Then you create the array when you actually need it, using a ReDim statement:

```
' An array defined in a BAS module (with Private scope)

Dim Customers() As String

...
```

```
Sub Main()

  ' Here you create the array.

  ReDim Customer(1000) As String

End Sub
```

If you're creating an array that's local to a procedure, you can do everything with a single ReDim statement:

```
Sub PrintReport()

  ' This array is visible only to the procedure.

  ReDim Customers(1000) As String

  ' ...

End Sub
```

If you don't specify the lower index of an array, Visual Basic assumes it to be 0, unless an Option Base 1 statement is placed at the beginning of the module. My suggestion is this: Never use an Option Base statement because it makes code reuse more difficult. (You can't cut and paste routines without worrying about the current Option Base.) If you want to explicitly use a lower index different from 0, use this syntax instead:

```
ReDim Customers(1 To 1000) As String
```

Dynamic arrays can be re-created at will, each time with a different number of items. When you re-create a dynamic array, its contents are reset to 0 (or to an empty string) and you lose the data it contains. If you want to resize an array without losing its contents, use the ReDim Preserve command:

```
ReDim Preserve Customers(2000) As String
```

When you're resizing an array, you can't change the number of its dimensions nor the type of the values it contains. Moreover, when you're using ReDim Preserve on a multidimensional array, you can resize only its last dimension:

```
ReDim Cells(1 To 100, 10) As Integer

...

ReDim Preserve Cells(1 To 100, 20) As Integer ' This works.

ReDim Preserve Cells(1 To 200, 20) As Integer ' This doesn't.
```

Finally, you can destroy an array using the Erase statement. If the array is dynamic, Visual Basic releases the memory allocated for its elements (and you can't read or write them any longer); if the array is static, its elements are set to 0 or to empty strings.

You can use the LBound and UBound functions to retrieve the lower and upper indices. If the array has two or more dimensions, you need to pass a second argument to these functions to specify the dimension you need:

```
Print LBound(Cells, 1) ' Displays 1, lower index of 1st dimension

Print LBound(Cells) ' Same as above

Print UBound(Cells, 2) ' Displays 20, upper index of 2nd dimension

' Evaluate total number of elements.
```

NumEls = (UBound(Cells) _ LBound(Cells) + 1) * _

(UBound(Cells, 2) _ LBound(Cells, 2) + 1)

Difference Between Static Array and Dynamic Array

Static Array	Dynamic Array
It must include fixed Number of items	It does not include fixed number of items
Static Arrays cannot be resized during COmpile time.	Dynamic Arrays can be resized after its Last dimension.
The Static array element set to 0 or to empty string.	Dynamic Arraya can be destroyed uding Erase Statement and the memory is released.
Static array can be initialized not erased.	After dsetroying dynamic Arrays you cannot read or write them.

Arrays within UDTs

UDT structures can include both static and dynamic arrays. Here's a sample structure that contains both types:

```
Type MyUDT
  StaticArr(100) As Long
  DynamicArr() As Long
```

```
End Type

...

Dim udt As MyUDT

' You must DIMension the dynamic array before using it.

ReDim udt.DynamicArr(100) As Long

' You don't have to do that with static arrays.

udt.StaticArr(1) = 1234
```

The memory needed by a static array is allocated within the UDT structure; for example, the StaticArr array in the preceding code snippet takes exactly 400 bytes. Conversely, a dynamic array in a UDT takes only 4 bytes, which form a pointer to the memory area where the actual data is stored. Dynamic arrays are advantageous when each individual UDT variable might host a different number of array items. As with all dynamic arrays, if you don't dimension a dynamic array within a UDT before accessing its items, you get an error 9-"Subscript out of range."

Collection

Note: To know about collection click here.

Difference between Array and Collection

	Array	Collection
1	It need to be predimensioned before using it.	It doesn't require Predimensional before using it.
2	We can't not generate new element in between existing elements.	New element can be inserted whenever required.
3	Array is a collection of homogeneous data	Collection can store nonhomogeneous data.
4	Index represents data Item in Array which is numeric data only.	Key represents data Item in Collection.key can be string value.
5	We can modify any array element.	We can not modify Collection Item.
6	Fast in execution.	Slower in execution

☐Variables

Variables are used by Visual Basic to hold information needed by your application.

☐Rules used in naming variables:

- **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)**

☐Variable Declaration

There are three ways for a variable to be typed (declared):

1. Default

2. Implicit

3. Explicit

- **If variables are not implicitly or explicitly typed, they are assigned the variant type by default. The variant data type is a special type used by Visual Basic that can contain numeric, string, or date data.**
- **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

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 MyInt as Integer

  Dim MyDouble as Double

  Dim MyString, 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 MyInt 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 MyInt 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 MyInt 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 MyInt is defined as Global in a module and declared local in a routine MyRoutine, while in MyRoutine, the local value of MyInt is accessed. Outside MyRoutine, the global value of MyInt is accessed.**

Scope of Variables

Global Variables

In Visual Basic jargon, global variables are those variables declared using the Public keyword in BAS modules. Conceptually, these variables are the simplest of the group because they survive for the life of the application and their scope is the entire application. (In

other words, they can be read and modified from anywhere in the current program.) The following code snippet shows the declaration of a global variable:

```
' In a BAS module

Public InvoiceCount as Long ' This is a global variable.
```

Visual Basic 6 still supports the Global keyword for backward compatibility with Visual Basic 3 and previous versions, but Microsoft doesn't encourage its use.

In general, it's a bad programming practice to use too many global variables. If possible, you should limit yourself to using module-level or local variables because they allow easier code reuse. If your modules and individual routines rely on global variables to communicate with each other, you can't reuse such code without also copying the definitions of the involved global variables. In practice, however, it's often impossible to build a nontrivial application without using global variables, so my suggestion is this: Use them sparingly and choose for them names that make their scope evident (for example, using a g_ or glo prefix). Even more important, add clear comments stating which global variables are used or modified in each procedure:

```
' NOTE: this procedure depends on the following global variables:

' g_InvoiceCount : number of invoices (read and modified)

' g_UserName : name of current user (read only)

Sub CreateNewInvoice()

  ...
```

```
End Sub
```

An alternative approach, which I often find useful, is to define a special GlobalUDT structure that gathers all the global variables of the application and to declare one single global variable of type GlobalUDT in one BAS module:

```
' In a BAS module

Public Type GlobalUDT

  InvoiceCount As Long

  UserName As String

  ....

End Type

Public glo As GlobalUDT
```

You can access these global variables using a very clear, unambiguous syntax:

```
' From anywhere in the application

glo.InvoiceCount = glo.InvoiceCount + 1
```

This technique has a number of advantages. First, the scope of the variable is evident by its name. Then if you don't remember the name of your variable, you can just type the three characters glo, and then type the dot and let Microsoft IntelliSense show you the list of all the components. In most cases, you just need to type a few characters and let Visual Basic complete the name for you. It's a tremendous time saver. The third advantage is that you can easily save all your

global variables to a data file:

```
' The same routine can save and load global data in GLO.

Sub SaveLoadGlobalData(filename As String, Save As Boolean)

  Dim filenum As Integer, isOpen As Boolean

  On Error Goto Error_Handler

  filenum = FreeFile

  Open filename For Binary As filenum

  isOpen = True

  If Save Then

  Put #filenum, , glo

  Else

  Get #filenum, , glo

  End If

Error_Handler:

  If isOpen Then Close #filenum

End Sub
```

The beauty of this approach is that you can add and remove global variables-actually, components of the GlobalUDT structure-without modifying the SaveLoadGlobalData routine. (Of course, you can't

correctly reload data stored with a different version of GlobalUDT.)

Module-Level Variables

If you declare a variable using a Private or a Dim statement in the declaration section of a module-a standard BAS module, a form module, a class module, and so on-you're creating a private module-level variable. Such variables are visible only from within the module they belong to and can't be accessed from the outside. In general, these variables are useful for sharing data among procedures in the same module:

```
' In the declarative section of any module

Private LoginTime As Date ' A private module-level variable

Dim LoginPassword As String ' Another private module-level
variable
```

You can also use the Public attribute for module-level variables, for all module types except BAS modules. (Public variables in BAS modules are global variables.) In this case, you're creating a strange beast: a Public module-level variable that can be accessed by all procedures in the module to share data and that also can be accessed from outside the module. In this case, however, it's more appropriate to describe such a variable as a property:

```
' In the declarative section of Form1 module

Public CustomerName As String ' A Public property
```

You can access a module property as a regular variable from inside the module and as a custom property from the outside:

```
' From outside Form1 module...

Form1.CustomerName = "John Smith"
```

The lifetime of a module-level variable coincides with the lifetime of the module itself. Private variables in standard BAS modules live for the entire life of the application, even if they can be accessed only while Visual Basic is executing code in that module. Variables in form and class modules exist only when that module is loaded in memory. In other words, while a form is active (but not necessarily visible to the user) all its variables take some memory, and this memory is released only when the form is completely unloaded from memory. The next time the form is re-created, Visual Basic reallocates memory for all variables and resets them to their default values (0 for numeric values, "" for strings, Nothing for object variables).

Dynamic Local Variables

Dynamic local variables are defined within a procedure; their scope is the procedure itself, and their lifetime coincides with that of the procedure:

```
Sub PrintInvoice()

  Dim text As String ' This is a dynamic local variable.

  ...

End Sub
```

Each time the procedure is executed, a local dynamic variable is re-created and initialized to its default value (0, an empty string, or

Nothing). When the procedure is exited, the memory on the stack allocated by Visual Basic for the variable is released. Local variables make it possible to reuse code at the procedure level. If a procedure references only its parameters and its local variables (it relies on neither global nor module-level variables), it can be cut from one application and pasted into another without any dependency problem.

Static Local Variables

Static local variables are a hybrid because they have the scope of local variables and the lifetime of module-level variables. Their value is preserved between calls to the procedure they belong to until their module is unloaded (or until the application ends, as is the case for procedures inside standard BAS modules). These variables are declared inside a procedure using the Static keyword:

```
Sub PrintInvoice()

  Static InProgress As Boolean ' This is a Static local variable.

  ...

End Sub
```

Alternatively, you can declare the entire procedure to be Static, in which case all variables declared inside it are considered to be Static:

```
Static Sub PrintInvoice()

  Dim InProgress As Boolean ' This is a Static local variable.

  ...
```

```
End Sub
```

Static local variables are similar to private module-level variables, to the extent that you can move a Static declaration from inside a procedure to the declaration section of the module (you only need to convert Static to Dim, because Static isn't allowed outside procedures), and the procedure will continue to work as before. In general, you can't always do the opposite: Changing a module-level variable into a Static procedure-level variable works if that variable is referenced only inside that procedure. In a sense, a Static local variable is a module-level variable that doesn't need to be shared with other procedures. By keeping the variable declaration inside the procedure boundaries, you can reuse the procedure's code more easily.

Static variables are often useful in preventing the procedure from being accidentally reentered. This is frequently necessary for event procedures, as when, for example, you don't want to process user clicks of the same button until the previous click has been served, as shown in the code below.

```
Private Sub cmdSearch_Click()

  Static InProgress As Boolean

  ' Exit if there is a call in progress.

  If InProgress Then MsgBox "Sorry, try again later": Exit Sub

  InProgress = True

  ' Do your search here.

  ...
```

```
 ' Then reenable calls before exiting.

 InProgress = False

End Sub
```

☐VB Constants Declaration

Constants, like variables, are named storage locations in memory with local, form, module, or global scope. Visual Basic prevents the value of a constant from being changed during program execution. Visual Basic uses constants more efficiently than variables, so if you plan to use a value that never changes, you should create a constant. To create a constant, use the Const statement with the following syntax:

[Public | Private] Const constname [As type] = expression

You declare constants by following the same rules as for variables for declaring local, module, or global data. Constants may only be declared as public within the General Declarations section of a standard module. For more information, see Declaring Module, Form, and Public Variables in this chapter.

The following example code declares and uses a public constant:

```
'General Declarations of a standard module

Public Const PI As Double = 3.1415

'Code within a procedure

dblArea = PI * dblRadius ^ 2

dblCircum = 2 * PI * dblRadius
```

☐VB Built-in Constants

Visual Basic provides predefined constants that are extremely useful in coding an application.

Alignment Constants

□Align Property

Constant	Value	Description
vbAlignNone	0	Size and location set at design time or in code.
vbAlignTop	1	Picture box at top of form.
vbAlignBottom	2	Picture box at bottom of form.
vbAlignLeft	3	Picture box at left of form.
vbAlignRight	4	Picture box at right of form.

□Alignment Property

Constant	Value	Description
vbLeftJustify	0	Left align
vbRightJustify	1	Right align
vbCenter	2	Center

Border Property Constants

BorderStyle Property (Form)

Constant	Value	Description
vbBSNone	0	No border
vbFixedSingle	1	Fixed single
vbSizable	2	Sizable (forms only)
vbFixedDouble	3	Fixed double (forms only)

BorderStyle Property (Shape and Line)

Constant	Value	Description
vbTransparent	0	Transparent.
vbBSSolid	1	Solid.
vbBSDash	2	Dash.
vbBSDot	3	Dot.
vbBSDashDot	4	Dash-dot.

vbBSDashDotDot	5	Dash-dot-dot.
vbBSInsideSolid	6	Inside solid.

Color Constants

□Colors

Constant	Value	Description
vbBlack	0x0	Black
vbRed	0xFF	Red
vbGreen	0xFF00	Green
vbYellow	0xFFFF	Yellow
vbBlue	0xFF0000	Blue
vbMagenta	0xFF00Ff	Magenta
vbCyan	0xFFFF00	Cyan
vbWhite	0xFFFFFF	White

Control Constants

☐ComboBox Control

Constant	Value	Description
vbComboDropdown	0	DropdownCombo
vbComboSimple	1	Simple Combo
vbComboDropdown List	2	DropdownList

☐ListBox Control

Constant	Value	Description
vbMultiSelectNone	0	None
vbMultiSelectSimple	1	Simple
vbMultiSelectExtended	2	Extended

☐ScrollBar Control

Constant	Value	Description
vbSBNone	0	None

vbHorizontal	1	Horizontal
vbVertical	2	Vertical
vbBoth	3	Both

□Shape Control

Constant	Value	Description
vbShapeRectangle	0	Rectangle
vbShapeSquare	1	Square
vbShapeOval	2	Oval
vbShapeCircle	3	vbShapeCircle
vbShapeRoundedRectangle	4	Rounded rectangle
vbShapeRoundedSquare	5	Rounded square.

Data Control Constants

□Options Property Constants

Constant	Value	Description
vbDataDenyWrite	1	Other users can't change records in recordset.
vbDataDenyRead	2	Other users can't read records in recordset.
vbDataReadOnly	4	No user can change records in recordset.
vbDataAppendOnly	8	New records can be added to the recordset, but existing records can't be read.
vbDataInconsistent	16	Updates can apply to all fields of the recordset.
vbDataConsistent	32	Updates apply only to those fields that will not affect other records in the recordset.
vbDataSQLPassThrough	64	Sends an SQL statement to an ODBC database.

Beginning-of-File Constants

Constant	Value	Description

vbMoveFirst	0	Move to first record.
vbBOF	1	Move to beginning of file.

□End-of-File Constants

Constant	Value	Description
vbMovelLat	0	Move to last record
vbEOF	1	Move to end of file
vbAddNew	2	Add new record to end of file.

□Recordset-Type Constants

Constant	Value	Description
vbRSTypeTable	0	Table-type recordset
vbRSTypeTable	1	Dynaset-type recordset
vbRSTypeSnapShot	2	Snapshot-type recordset

Date Constants

Return Values

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Dir, GetAttr, and SetAttr Constants

Constant	Value	Description
vbNormal	0	Normal(default for Dir and SetAttr)
vbReadOnly	1	Readonly
vbHidden	2	Hidden

vbSystem	4	System file
vbVolume	8	Volume label
vbDirectory	16	Directory
vbArchive	32	File has changed since last backup

Form Constants

Show Parameters

Constant	Value	Description
vbModal	1	Modal from
vbModeless	0	Modelessform

Arrange Method for MDI Forms

Constant	Value	Description
vbCascade	0	Cascade all nonminimized MDI child froms
vbTileHorizontal	1	Horizontally tile all nonminimized MDI child froms

vbTileVertical	2	Vertically tile all nonminimized MDI child forms
vbArrangeIcons	3	Arrange icons for minimized MDI child forms.

WindowState Property

Constant	**Values**	**Description**
vbNormal	0	Normal
vbMinimized	1	Minimized
vbMaximized	2	Maximized

Miscellaneous Constants

ZOrder Method

Constant	Value	Description
vbBringToFront	0	Bring to front
vbSendToBack	1	Send to back

Shell Constants

StrConv Constants

Constant	Value	Description
vbUpper	1	Uppercases the string
vbLower	2	Lowercases the string
vbProperCase	3	Uppercases first letter of every word in string.

Statements

1. Branch statement

The main branch statement is the If...Else...Else If...End If block. Visual Basic supports several flavors of this statement, including single-line and multiline versions:

```
' Single line version, without Else clause

If x > 0 Then y = x

' Single line version, with Else clause

If x > 0 Then y = x Else y = 0

' Single line, but with multiple statements separated by colons

If x > 0 Then y = x: x = 0 Else y = 0

' Multiline version of the above code (more readable)

If x > 0 Then

  y = x

  x = 0

Else

  y = 0

End If

' An example of If..ElseIf..Else block
```

```
If x > 0 Then
  y = x
ElseIf x < 0 Then
  y = x * x
Else ' X is surely 0, no need to actually test it.
  x = -1
End If
```

You should be aware that any nonzero value after the If keyword is considered to be True and therefore fires the execution of the Then block:

Note: The following lines are equivalent.

```
If value <> 0 Then Print "Non Zero"
If value Then Print "Non Zero"
```

The following examples show how you can often write more concise and efficient code by rewriting a Boolean expression:

```
' If two numbers are both zero, you can apply the OR operator
' to their bits and you still have zero.
If x = 0 And y = 0 Then ...
If (x Or y) = 0 Then ...

' If either value is <>0, you can apply the OR operator

' to their bits and you surely have a nonzero value.

If x <> 0 Or y <> 0 Then ...

If (x Or y) Then ...

If Not (x = y) Then ... ' The same as x<>y

If Not x Then ... ' The same as x<>-1, don't use instead of x=0
```

- 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):

```
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 code 8; symbolic constant vbKeyBack) to pass through the key press event. Else, you will not be able to edit the text box properly.

2. Select case statement

The Select Case statement is less versatile than the If block in that it can test only one expression against a list of values:

```
Select Case Text1.text
  Case "0" To "9"
  ' It's a digit.
  Case "A" To "Z", "a" To "z"
  ' It's a letter.
  Case ".", ",", " ", ";", ":", "?"
  ' It's a punctuation symbol or a space.
```

```
    Case Else

    ' It's something else.

  End Select
```

Note 1:

```
' This series of subexpressions connected by the AND
operator:

If x > 0 And Sqr(y) > x And Log(x) < z Then z = 0

' can be rewritten as:

Select Case False

  Case x > 0, Sqr(y) > x, Log(x) < z

  ' Do nothing if any of the above meets the condition,

  ' that is, is False.

  Case Else

  ' This is executed only if all the above are True.

  z = 0

End Select
```

Note 2:

```
' This series of subexpressions connected by the OR
operator:

If x = 0 Or y < x ^ 2 Or x * y = 100 Then z = 0

' can be rewritten as:

Select Case True

  Case x = 0, y < x ^ 2, x * y = 100

  ' This is executed as soon as one of the above is found

  ' to be True.

  z = 0

End Select
```

3. 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:

```
xyz :
-----
-----
GoTo xyz
```

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

4. Other Functions

- **A few VBA functions are closely related to control flow, even if by themselves they don't alter the execution flow. The IIf function, for example, can often replace an If...Else...End If block, as in the following code:**

```
' These lines are equivalent.

If x > 0 Then y = 10 Else y = 20

y = IIf(x > 0, 10, 20)
```

- **The Switch function accepts a list of (condition, value) pairs**

and returns the first value that corresponds to a condition that evaluates as True. See, for example, how you can use this function to replace this Select Case block:

```
Select Case x

  Case Is <= 10: y = 1

  Case 11 To 100: y = 2

  Case 101 To 1000: y = 3

  Case Else: y = 4

End Select



'Same effect in just one line.

' The last "True" expression replaces the "Else" clause.

y = Switch(x <= 10, 1, x <= 100, 2, x <= 1000, 3, True, 4)
```

CAUTION

While the IIf, Choose, and Switch functions are sometimes useful for reducing the amount of code you have to write, you should be aware that they're always slower than the If or Select Case structure that they're meant to replace. For this

reason, you should never use them in time-critical loops.

Loops

- Looping is done with the Do/Loop format. Loops are used for operations are to be repeated some number of times. The loop repeats until some specified condition at the beginning or end of the loop is met.

- Do While/Loop Example:

```
Counter = 1
Do While Counter <= 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 the 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.

> 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 or For/Next loop.

Counting is accomplished using the For/Next loop.

```
For counter = startvalue To endvalue [Step
increment]
  ' Statements to be executed in the loop...
Next
```

Example

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

In this example, the variable J initializes at 1 and, with each iteration of the For/Next loop, is incremented by 2 (Step). This looping continues until J 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.

TIP

Always use an Integer or Long variable as the controlling variable of a For...Next loop because they're faster than a Single or a Double controlling variable, by a factor of 10 or more. If you need to increment a floating-point quantity, the most efficient technique is

explained in the next example.

This technique permits you to execute a block of statements with different values for a controlling variable, which don't need to be in sequence:

```
' Test if Number can be divided by any of the first 10 prime numbers.

Dim var As Variant, NotPrime As Boolean

For Each var In Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)

  If (Number Mod var) = 0 Then NotPrime = True: Exit For

Next
```

The values don't even have to be numeric:

```
' Test if SourceString contains the strings "one", "two", "three", etc.

Dim var2 As Variant, MatchFound As Boolean

For Each var2 In Array("one", "two", "three", "four", "five")

  If InStr(1, SourceString, var2, vbTextCompare) Then

  MatchFound = True: Exit For

  End If

Next
```

☐Visual Basic Operators

- **Arithmetic operators:**

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

^	Exponentiation
*** /**	Multiplication and division
\	Integer division (truncates)
Mod	Modulus
+ -	Addition and subutraction

Parentheses around expressions can change precedence.

- **Concatentate Operator:**

To concatentate two strings, use the & symbol or the + symbol:

lblTime.Caption = "The current time is" & Format(Now, "hh:mm")

txtSample.Text = "Hook this " + "to this"

- **Comparison Operators:**

There are six comparison operators in Visual Basic:

>	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)**

- **Logical Operators:**

We will use three logical operators

Not	Logical not
And	Logical and
Or	Logical or

The Not operator simply negates an operand.

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.

Operator evaluation precedence

1. **First: Arithmetic**
2. **Second: Comparison**
3. **Last: Logical Operator**

At first glance, it might seem that Visual Basic 6 supports countless properties for various objects. Fortunately, there's a set of properties many objects of different classes share. In this section, we'll examine these common properties.

◻The Left, Top, Width, and Height Properties

By default, these properties are measured in twips, a unit that lets you create resolution-independent user interfaces, but you can switch to another unit, for example, pixels or inches, by setting the container's ScaleMode property. But you can't change the unit used for forms because they have no container: Left, Top, Width, and Height properties for forms are always measured in twips.

```
' Double a form's width, and move it to the

' upper left corner of the screen.

Form1.Width = Form1.Width * 2

Form1.Left = 0

Form1.Top = 0
```

CAUTION

Controls don't necessarily have to support all four properties in a uniform manner. For example, ComboBox controls' Height property can be read but not written to, both at design time and run time.

The ForeColor and BackColor Properties

The effect of these properties depends on other properties: for example,

- **Setting the BackColor property of a Label control has no effect if you set the BackStyle property of that Label to 0-Transparent.**

- **CommandButton controls are peculiar in that they expose a BackColor property but not a ForeColor property, and the background color is active only if you also set the Style property to 1-Graphical.**

My first suggestion is always use a standard color value unless you have a very good reason to use a custom color.

```
' Make Label1 appear in a selected state.

Label1.ForeColor = vbHighlightText

Label1.BackColor = vbHighlight
```

When you're assigning a custom color, you can use one of the symbolic constants that Visual Basic defines for the most common colors (vbBlack, vbBlue, vbCyan, vbGreen, vbMagenta, vbRed, vbWhite, and vbYellow), or you can use a numeric decimal or hexadecimal constant:

```
' These statements are equivalent.

Text1.BackColor = vbCyan

Text1.BackColor = 16776960

Text1.BackColor = &HFFFF00
```

You can also use an RGB function to build a color value composed of its red, green, and blue components. Finally, to ease the porting of existing QuickBasic applications, Visual Basic supports the QBColor function:

```
' These statements are equivalent to the ones above.

Text1.BackColor = RGB(0, 255, 255) ' red, green, blue values

Text1.BackColor = QBColor(11)
```

The Font Property

Font is a compound object, and you must assign its properties separately. Font objects expose the Name, Size, Bold, Italic, Underline, and Strikethrough properties.

```
Text1.Font.Name = "Tahoma"

Text1.Font.Size = 12

Text1.Font.Bold = True

Text1.Font.Underline = True
```

TIP

You can use the Set command to assign whole Font objects to controls.

' Assign to Text2 the same font as used by Text1.

Set Text2.Font = Text1.Font

The Caption and Text Properties

The Caption property is special in that it can include an ampersand (&) character to associate a hot key with the control. The Text property, when present, is always the default property for the control, which means that it can be omitted in code:

' These statements are equivalent.

Text2.Text = Text1.Text

Text2 = Text1

NOTE

In general, if a control exposes the Text property it also supports the SelText, SelStart, and SelLength properties, which return information about the portion of text that's currently selected in the control.

The Enabled and Visible Properties

For instance, a Form is a container for its controls and a Frame control can be a container for a group of OptionButton controls-setting its Visible or Enabled properties indirectly affects the state of its contained objects. This feature can often be exploited to reduce the amount of code you write to enable or disable a group of related controls.

```
' Enable or disable the Text1 control when

' the user clicks on the Check1 CheckBox control.

Private Sub Check1_Click()

  Text1.Enabled = (Check1.Value = vbChecked)

End Sub
```

The TabStop and TabIndex Properties

If a control is able to receive the input focus, it exposes the TabStop property. Most intrinsic controls support this property, including TextBox, OptionButton, CheckBox, CommandButton, OLE, ComboBox, both types of scroll bars, the ListBox control, and all its variations. The default value for this property is True, but you can set it to False either at design time or run time.If a control supports the TabStop property, it also supports the TabIndex property, which affects the Tab order sequence-that is, the sequence in which the controls are visited when the user presses the Tab key repeatedly.

```
' Let the user press the Alt+N hot key

' to move the input focus on the Text1 control.

Label1.Caption = "&Name"

Text1.TabIndex = Label1.TabIndex + 1
```

The MousePointer and MouseIcon Properties

If you want to show an hourglass cursor, wherever the user moves the mouse, use this code:

```
' A lengthy routine

Screen.MousePointer = vbHourglass

...

' Do your stuff here

...

' but remember to restore default pointer.

Screen.MousePointer = vbDefault



Here's another example:

' Show a crosshair cursor when the mouse is over the Picture1

' control and an hourglass elsewhere on the parent form.

Picture1.MousePointer = vbCrosshair

MousePointer = vbHourglass
```

the MouseIcon property is used to display a custom, user-defined mouse cursor. In this case, you must set the MousePointer to the special value 99-vbCustom and then assign an icon to the MouseIcon property:

The Tag Property

All controls support the Tag property, without exception. This is true even for ActiveX controls, including any third-party controls. How can I be so certain that all controls support this property? The reason is that the property is provided by Visual Basic itself, not by the control. Tag isn't the only property provided by Visual Basic to any control: Index, Visible, TabStop, TabIndex, ToolTipText, HelpContextID, and WhatsThisHelpID properties all belong to the same category. These properties are collectively known as extender properties. Note that a few extender properties are available only under certain conditions. For example, TabStop is present only if the control can actually receive the focus. The Tag property is distinctive because it's guaranteed to be always available, and you can reference it in code without any risk of raising a run-time error.

The Tag property has no particular meaning to Visual Basic: It's simply a container for any data related to the control that you want to store. For example, you might use it to store the initial value displayed in a control so that you can easily restore it if the user wants to undo his or her changes.

Just as there are many properties that most objects share, they also have many methods in common. In this section, we examine these methods.

The Move Method

If a control supports Left, Top, Width, and Height properties, it also supports the Move method, through which you can change some or all four properties in a single operation. The following example changes three properties: Left, Top, and Width.

```
' Double a form's width, and move it to the upper left corner of the screen.

' Syntax is: Control.Move Left, Top, Width, Height.

Form1.Move 0, 0, Form1.Width * 2
```

TIP

The Move method should always be preferred to individual property assignment for at least two reasons: This operation is two to three times faster than four distinct assignments, and if you're modifying the Width and Height properties of a form, each individual property assignments would fire a separate Resize event, thus adding a lot of overhead to your code.

◻The Refresh Method

The Refresh method causes the control to be redrawn. But you can explicitly invoke this method when you modify a control's property and you want the user interface to be immediately updated:

```
For n = 1000 To 1 Step -1

  Label1.Caption = CStr(i)

  Label1.Refresh ' Update the label immediately.

Next
```

CAUTION

If you want to update all the controls on a form but you don't want the end user to interact with the program, just execute the Refresh method of the parent form.

◻The SetFocus Method

```
' Move the focus to Text1.

If Text1.Visible And Text1.Enabled Then

  Text1.SetFocus

End If
```

And here's the code for the other possible approach, using the On Error statement:

```
' Move the focus to Text1.

On Error Resume Next

Text1.SetFocus
```

Here's another possible solution:

```
Private Sub Form_Load()

  Text1.TabIndex = 0

End Sub
```

The ZOrder Method

The ZOrder method affects the visibility of the control with respect to other overlapping controls. You just execute this method without

any argument if you want to position the control in front of other controls; or you can pass 1 as an argument to move the control behind other controls:

```
' Move a control behind any other control on the form.

Text1.ZOrder 1

Text1.ZOrder ' Move it in front.
```

Note that you can set the relative z-order of controls at design time using the commands in the Order submenu of the Format menu, and you can also use the Ctrl+J key combination to bring the selected control to the front or the Ctrl+K key combination to move it behind other controls.

The actual behavior of the ZOrder method depends on whether the control is standard or lightweight. In fact, lightweight controls can never appear in front of standard controls. In other words, the two types of controls-standard and lightweight-are located on distinct z-order layers, with the layer of standard controls in front of the layer of lightweight controls. This means that the ZOrder method can change the relative z-order of a control only within the layer it belongs to. For example, you can't place a Label (lightweight) control in front of a TextBox (standard) control. However, if the standard control can behave like a container control-a PictureBox or a Frame control, for example-you can make a lightweight control appear in front of the standard control if you place the lightweight control inside that container control, as you can see in Figure 2-5.

The ZOrder method also applies to forms. You can send a form behind all other forms in the same Visual Basic application, or you can bring it in front of them. You can't use this method, however, to

control the relative position of your forms with respect to windows belonging to other applications.

In addition to common properties and methods, Visual Basic 6 forms and controls support common events. In this section, we'll describe these events in some detail.

The Click and DblClick Events

Whenever a CheckBox or an OptionButton control's Value property changes through code, Visual Basic fires a Click event, exactly as if the user had clicked on it.

ListBox and ComboBox controls also fire Click events whenever their ListIndex properties change.

TIP

Also notice that when you double-click on a control, it receives both the Click and the DblClick events. This makes it difficult to distinguish single clicks from double-clicks. While you shouldn't assign separate effects to click and double-click actions on the same control, here's a simple method to work around the problem of finding out what the user actually did:

```
' A module-level variable
Dim isClick As Boolean


Private Sub Form_Click()
  Dim t As Single
  isClick = True
  ' Wait for the second click for half a second.
  t = Timer
  Do
  DoEvents
  ' If the DblClick procedure canceled this event,
  ' bail out.
  If Not isClick Then Exit Sub
  ' The next test accounts for clicks just before midnight.
  Loop Until Timer > t + .5 Or Timer < t
  ' Do your single-click processing here.
  ...
End Sub
```

```
Private Sub Form_DblClick()
  ' Cancel any pending click.
  isClick = False
  ' Do your double-click processing here.
  ...
End Sub
```

The Change Event

Whenever the contents of a control change, Visual Basic fires a Change event.

TextBox and ComboBox controls raise a Change event when the user types something in the editable area of the control. (But be careful, the ComboBox control raises a Click event when the user selects an item from the list portion rather than types in a box.)

Both scroll bar controls raise the Change event when the user clicks on either arrows or moves the scroll boxes.

The Change event is also supported by the PictureBox, DriveListBox, and DirListBox controls.

◻The GotFocus and LostFocus Events

These events are conceptually very simple: GotFocus fires when a control receives the input focus, and LostFocus fires when the input focus leaves it and passes to another control. Finally, note that forms support both GotFocus and LostFocus events, but these events are raised only when the form doesn't contain any control that can receive the input focus, either because all of the controls are invisible or the TabStop property for each of them is set to False.

◻The KeyPress, KeyDown, and KeyUp Events

These events fire whenever the end user presses a key while a control has the input focus. The exact sequence is as follows:

KeyDown (the users presses the key),

KeyPress (Visual Basic translates the key into an ANSI numeric code), and

KeyUp (the user releases the key).

Only keys that correspond to control keys (Ctrl+x, BackSpace, Enter, and Escape) and printable characters activate the KeyPress event.

For all other keys-including arrow keys, function keys, Alt+x key combinations, and so on-this event doesn't fire and only the KeyDown and KeyUp events are raised.

The KeyPress event is the simplest of the three. It's passed the ANSI code of the key that has been pressed by the user, so you often need to convert it to a string using the Chr$() function:

```
Private Sub Text1_KeyPress(KeyAscii As Integer)

  ' Convert all keys to uppercase, and reject blanks.

  KeyAscii = Asc(UCase$(Chr$(KeyAscii))

  If KeyAscii = Asc(" ") Then KeyAscii = 0

End Sub
```

The KeyDown and KeyUp events receive two parameters, KeyCode and Shift. The former is the code of the pressed key, the latter is an Integer value that reports the state of the Ctrl, Shift, and Alt keys; because this value is bit-coded, you have to use the AND operator to extract the relevant information:

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As
Integer)

  If Shift And vbShiftMask Then

  ' Shift key pressed

  End If

  If Shift And vbCtrlMask Then

  ' Ctrl key pressed
```

```
    End If
    If Shift And vbAltMask Then
    ' Alt key pressed
    End If
    ' ...
End Sub
```

The KeyCode parameter tells which physical key has been pressed, and it's therefore different from the KeyAscii parameter received by the KeyPress event. You usually test this value using a symbolic constant, as in the following code:

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
    ' If user presses Ctrl+F2, replace the contents
    ' of the control with the current date.
    If KeyCode = vbKeyF2 And Shift = vbCtrlMask Then
    Text1.Text = Date$
    End If
End Sub
```

In contrast to what you can do with the KeyPress event, you can't

alter the program's behavior if you assign a different value to the KeyCode parameter.

For example, suppose that you want to offer your users the ability to clear the current field by pressing the F7 key. You don't want to write the same piece of code in the KeyDown event procedure for each and every control on your form, and fortunately you don't have to. In fact, you only have to, set the form's KeyPreview property to True (either at design time or at run time, in the Form_Load procedure, for example) and then write this code:

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)

  If KeyCode = vbKeyF7 Then

  ' An error handler is necessary because we can't be sure

  ' that the active control actually supports the Text

  ' property.

  On Error Resume Next

  ActiveControl.Text = ""

  End If

End Sub
```

If the form's KeyPreview property is set to True, the Form object receives all keyboard-related events before they're sent to the control that currently has the input focus. Use the form's

ActiveControl property if you need to act on the control with the input focus, as in the previous code snippet.

The MouseDown, MouseUp, and MouseMove Events

While writing code for mouse events, you should be aware of a few implementation details as well as some pitfalls in using these events. Keep in mind the following points:

When you press a mouse button over a form or a control and then move the mouse outside its client area while keeping the button pressed, the original control continues to receive mouse events. In this case, the mouse is said to be captured by the control: the capture state terminates only when you release the mouse button. All the MouseMove and MouseUp events fired in the meantime might receive negative values for the x and y parameters or values that exceed the object's width or height, respectively.

MouseDown and MouseUp events are raised any time the user presses or releases a button. For example, if the user presses the left button and then the right button (without releasing the left button), the control receives two MouseDown events and eventually two MouseUp events.

The Button parameter passed to MouseDown and MouseUp events reports which button has just been pressed and released, respectively. Conversely, the MouseMove event receives the current state of all (two or three) mouse buttons.

When the user releases the only button being pressed, Visual Basic

fires a MouseUp event and then a MouseMove event, even if the mouse hasn't moved. This detail is what makes the previous code example work correctly after a button release: The current status is updated by the extra MouseMove event, not by the MouseUp event, as you probably expected. Note, however, that this additional MouseMove event doesn't fire when you press two buttons and then release only one of them.

It's interesting to see how MouseDown, MouseUp, and MouseMove events relate to Click and DblClick events:

A Click event occurs after a MouseDown &ldots; MouseUp sequence and before the extra MouseMove event.
When the user double-clicks on a control, the complete event sequence is as follows: MouseDown, MouseUp, Click, MouseMove, DblClick, MouseUp, MouseMove. Note that the second MouseDown event isn't generated.
If the control is clicked and then the mouse is moved outside its client area, the Click event is never raised. However, if you double-click a control and then you move the mouse outside its client area, the complete event sequence occurs. This behavior reflects how controls work under Windows and shouldn't be considered a bug.

□Difference between Methods and Events

Method	Event
They are inbuilt procerdure in Visual basic.	They are user written procedure in the Visual Basic.

By taking object on a from we can get method list while writing the form code.	The events are fired when the user passes any action on the object through code.
A method requries an object tp provide them a context.	When we select a object it events are automatically added to form Module.
User cannot write their own methods for the controls.	Users can write their own code in an event to force a control to react precisely the way you want it to.
Combo1.Additem	Combo1.Dbclick()
e.g Line , Circle, psets, points are methods.	e.g Paint is a graphic events.

◻The Show Method

The Show method displays a Form object. If the form is not already loaded, the Show method will automatically load it for you. To use the Show method, use the following syntax:

object.Show style, ownerform

The style parameter is an integer value that determines if the form being shown is modal or modeless. A modal form requires the user to take some action before the focus can switch to another form within an application. A modeless form does not require a response from the user before the focus can be switched to another form within an application. The ownerform parameter specifies the component which "owns" the form being shown. Both the style and ownerform parameters are optional.

The following example code uses the Show method:

frmCustomerInfo.Show

◻The Load Statement

The Load statement loads a form into memory but does not display it. To use the Load statement, use the following syntax:

Load object

The following example code uses the Load statement:

Load frmCustomerInfo

Note

You don’t need to use the Load statement with forms unless you want to load a form without displaying it. Any reference to a form automatically loads the form if the form is not already loaded. Once the form is loaded, its properties and controls can be altered by the application, whether or not the form is actually visible.

TIP

You can use both the Show method and the Load statement to load a form into memory; however, the Show method displays the form; while the Load statement does not.

The Hide Method

The Hide method will cause the form to be invisible. The form remains in memory.

To use the Hide method, use the following syntax:

object.Hide

The following example code invokes the Hide method:

frmMain.Hide

Note

If the form isn’t loaded when the Hide method is invoked, the Hide method loads the form but doesn’t display it.

The Unload Statement

Unloading a form removes the form from the display and releases its memory. You can also unload a form, then load it again to reset its properties to their original values. The Unload statement:

- **Unloads a form or control from memory.**
- **Releases the display component of the form from memory.**
- **Resets the form and control properties to their original values. If any changes were made to the form, either by the program or by the user, these changes are lost.**
- **Invokes the Unload event.**
- **Terminates execution of the application if the unloaded form is the only form in the application.**

To use the Unload statement, use the following syntax:

```
Unload object
```

The following example code invokes the Unload statement:

```
Unload frmMain
```

Note

- **You can use the Me keyword to reference the current form. For example, Unload Me unloads the current form from memory.**

- **An application does not end until all forms are unloaded from memory. The End statement removes all forms from memory and ends the application.**

☐The Unload Event

Use an Unload event procedure to verify that the form should be unloaded or to specify actions you want to take place when the form is unloaded. Any form-level validation code needed for closing the form or saving its data to a file can also be included. The Unload event occurs when:

§ **The form is unloaded using the Unload statement.**

§ **The form is closed by the user either clicking the Close command on the application menu, or clicking the Close button on the application title bar.**

Note

- **The Unload event does not occur if the form is removed from memory by the End statement.**

- If you plan to use a form repeatedly, it's faster to hide and show the form, rather than load and unload it.

Example of Form Display Capabilities

The following example code controls the startup of a program. It loads and displays a splash screen, loads the first form, hides the splash screen, and then displays the first form.

```
Sub Main( )

    frmSplashScreen.Show

    Load frmFirstForm

    frmSplashScreen.Hide

    frmFirstForm.Show

End Sub
```

You can end execution of your program by unloading the last form in your application or by using the End statement. The End statement terminates execution of your application and unloads all forms from memory.

```
Sub Form_Unload()

    [Final code statements]

    End

End Sub
```

Forms support a number of events that fire at various times throughout the life of the form and the life of the application. These events include:

☐Initialize

The Initialize event occurs when an application creates an instance of a form before the form is loaded or displayed. The code you place in the Form_Initialize event procedure is therefore the first code that gets executed when a form is created. In the initialization state, the form exists as an object, but it has no window. The Initialize event only occurs the first time a form is loaded, unless the form is set to Nothing, as shown in the following example code:

```
Set frmCustomerInfo = Nothing
```

☐Load

The Load event fires each time a form loads. This happens when the Load method is used, the Show method is used, or a control is referenced on a form that has not yet been loaded.

☐Activate

The Activate event occurs when the form receives focus. As a user changes from one form to another in a modeless environment, the Activate event will fire.

GotFocus

A form receives a GotFocus event only if there are no controls on the form capable of receiving the focus. Typically, you use a GotFocus event procedure to specify the actions that occur when a control or form first receives focus.

Note

If you add code to the form's Activate event, the GotFocus event will not fire.

LostFocus

Like the GotFocus event, the LostFocus event will only fire if there are no controls on the form that are capable of losing focus.

Deactivate

The Deactivate event occurs when the form loses focus.

Note

If you add code to the LostFocus event, the Deactivate event will not fire.

☐QueryUnload

The QueryUnload event fires when the form receives a command to unload. This event provides developers with the ability to evaluate how the form received the request to unload and cancel the unload process. The QueryUnload event will fire before the Unload event. If the QueryUnload event cancels the unload request, the Unload event will not fire.

☐Unload

The Unload event fires each time a form is unloaded from memory. This occurs when an application ends, using the End statement, or the form is explicitly unloaded with the Unload method.

☐Terminate

The only way to release all memory and resources is to unload the form and then set all references to Nothing. Your form receives its Terminate event just before it is destroyed (set to Nothing) or the application ends. However, if the End statement is used, the Terminate event will not fire.

☐Order of Execution

The events listed above execute in a specific order. It is important to understand this order when deciding where to place code that initializes controls, prompts the user to save changes, or closes database connections.

The following list describes the general flow of event execution:

Initialize	(fires once when the form is first referenced)
Load	(fires every time the form loads)
Activate	(fires whenever the form is activated from within the program)
GotFocus	(fires only if no controls on the form can receive focus)
LostFocus	(fires when focus changes to another form)
Deactivate	(fires when the form in no longer active within the program)
QueryUnload	(fires when the form receives an unload command)
Unload	(fires after the QueryUnload event)
Terminate	(fires when the form is set to Nothing or the program ends)

As with any other objects, you can set form properties at design time in the Properties window, or at run time by writing code.

☐Name

The default name for a new form is “Form” plus a unique integer. For example, the first new Form object is Form1, the second is Form2, and so on.

Because a form’s name is used to reference it in code, it is important to set the Name property early in the development process, preferably when you create the form. A form's Name property must start with a letter; the name can include numbers and underline (_) characters, but it cannot include punctuation or spaces.

Note Forms cannot have the same name as another public object, such as Clipboard, Screen, or App. Although the Name property setting can be a keyword, property name, or the name of another object, this can create conflicts in your code.

☐Icon

The Icon property specifies the icon that appears when a form is minimized. In Windows, the icon also appears in the title bar.

You set the Icon property at design time. Visual Basic supplies a large library of icons for your applications, but you can use any file with the extension .ico.

☐WindowState

The WindowState property determines how the form will appear when displayed (normal, minimized, or maximized). You set the

WindowState property at run time. Before a form is displayed, the WindowState property is always set to vbNormal, regardless of its initial setting.

To set the WindowState property, use the following syntax:

object.WindowState = value

The following table lists possible values for the WindowState property.

Constant	Setting	Description
VbNormal	0	(Default) Normal size
VbMinimized	1	Minimized to an icon
VbMaximized	2	Enlarged to maximum size

The following example code maximizes frmCalculator:

frmCalculator.WindowState = vbMaximized

☐BorderStyle

The BorderStyle property controls the appearance of the form's border. This property also determines whether the user can resize, minimize, or maximize the form.

The following table lists the BorderStyle property settings for a Form object.

Constant	Setting	Description
vbBSNone	0	None (no border or border-related elements).
vbFixedSingle	1	Can include Control menu box, title bar, Maximize button, and Minimize button. Resizable only using Maximize and Minimize buttons.
vbSizable	2	Default) Resizable using any of the optional border elements listed for setting 1.
vbFixedDialog	3	Can include Control menu box and title bar; cannot include Maximize or Minimize buttons. Not resizable.
vbFixedToolWindow	4	Displays a nonsizable window with a Close button and title bar text in a reduced font size. The form does not appear in the Windows 98 taskbar.
vbSizableToolWindow	5	Displays a sizable tool window with a Close button and title bar text in a reduced font size. The form does not appear in the Windows 98 taskbar.

MaxButton and MinButton

The MaxButton and MinButton properties determine whether standard Windows Maximize and Minimize buttons are displayed in

the form's title bar. Setting these properties has no effect unless the BorderStyle property is set to 1, 2, or 3 (vbFixedSingle, vbSizable, or vbFixedDialog).

ControlBox

The ControlBox property determines whether a standard Windows control box appears on a form. When you set the ControlBox property to True, you must also set the BorderStyle property to 1, 2, or 3 (vbFixedSingle, vbSizable, or vbFixedDouble) to display the control box.

☐Type of Cotrols

Standard controls, ActiveX controls, and insertable objects. These categories are described in the table below.

Type	Description
Standard controls	These controls are contained in Visual Basic. Examples include the CommandButton and TextBox controls. Also called intrinsic controls, standard controls are always available from the Toolbox and are the main focus of this chapter. These controls are covered in the topic of Standard Controls in this chapter.
ActiveX controls	These controls are separate files with the .ocx extension. These controls can be added to the Toolbox. These controls are explained in details in chapter 9. (Advance Active X control)
Insertable objects	These are typically OLE objects such as a Microsoft Excel Worksheet object. Insertable objects can be added to the Toolbox and are covered briefly in the topic Insertable Objects in this chapter and is discussed in detail in Chapter 12. Win API, OLE, DHTML.....

◻Text Box

Special Properties	Value
Locked	=True / False i.e. If Text1.Locked = True then user can not edit the text of text box
MaxLength	Numeric value represents maximum char length allowed user to type in the control
MultiLine	=True / False i.e. if Text1.Multiline = True then user text can navigate to next line in text box.
PasswordChar	Char represents face of text typed in text box. This is useful for showing Password in user field.
ScrollBar	0 - None, 1 - Horizontal, 2 - Vertical, 4 - Both. The value it self indicates possible scrollbars in text box. This property works only in environment where Multiline property set to true for text box control.
Special Events	Details
Change ( )	Fires when value of text box control get change by user
GotFocus ( )	Fires when text box get cursor into it

Keypress (Keyascii as integer)	Fires when user press any key to type in the text box, it navigate with Ascii value of key being pressed
MouseMove (Button as integer, Shift as integer, x as single, y as single)	Fires when user moves mouse over the text box.
LostFocus ( )	Fires when cursor comes out of text box.

☐Label

Special Properties	Value
AutoSize	= True / False i.e. Determines whether a control is automatically resized to display its entire contents.
BackStyle	0 - Transparent, 1 - Opaque i.e. indicates background style for the label
RightToLeft	= True / False i.e. Determines text display direction
UseMnemonicn	= True / False i.e. sets a value that specifies whether an "&" in a caption of label defines access key

WordWrap	= True / False i.e. Returns / sets a value that determine whether a control expand to fit the text in its caption.

❑Difference between Text Box and Label

Text Box	Label
"text" is default property	"caption" is default property
Control.text returns value of the control	Control.caption returns value of the control
can not expand to the size of content automatically	can expand to the size of context automatically using AutoSize property
it is userful to get value from user	it is useful to show information for user
provides Key Board Events	does not provides Key Board Events

☐Picture Box

Special Properties	Value
AutoRedraw	= True / False if True then Enables automatic repainting of PictureBox control. Graphics and text are written to the screen and to an image stored in memory. The object doesn't receive Paint events; it's repainted when necessary, using the image stored in memory.
AutoSize	= True/ False if True then Automatically resizes control to the size of picture.
DrawMode	= Number (1 to 16) Returns or sets a value that determines the appearance of output from graphics method or the appearance of a Shape or Line control. Default : <Object>.DrawMode = 13 - Copypen

DrawStyle	= Number (0 to 6) Returns or sets a value that determines the line style for output from graphics methods like Circle, Line etc. Default : <Object>.DrawStyle = 0 - Solid
DrawWidth	=Number sets the line width for output from graphics methods.
FillColor	= Value (Color value) Sets the color used to fill in shapes, circles and boxes created with the Circle and Line graphics methods. By default, FillColor is set to 0 (Black) Here, any color function or color contast can be used like vbRed, vbBlack or QBColor(0 to 15) or RGB(R, G, B)
FillStyle	= Number Sets the pattern used to fill Shape controls as well as circles and boxes created with the Circle and Line graphics methods. When FillStyle is set to 1 (Transparent), the FillColor property is ignored. Value can be 0 to 7

Picture	= and picture file sets picture file for the picture box.
ScaleMode	sets a value indicating the unit of measurement for coordinates of an picture box object when using graphics methods or when positioning controls. ScaleHeight, ScaleWidth are related properties to scalemode properties.
Special Events	Details
Paint ( )	Fires when value of text box control get change by user
Resize ( )	Fires when text box get cursor into it
MouseMove (Button as integer, Shift as integer, x as single, y as single)	Fires when user moves mouse over the Picture box. This events is most important to get graphics method and work with graphics in picture box. Co-ordinates or argument remains same for all the objects or controls.

Paint ( )	A Paint event procedure is useful if you have output from graphics methods in your code. With a Paint procedure, you can ensure that such output is repainted when necessary. The Paint event is invoked when the Refresh method is used. If the AutoRedraw property is set to True, repainting or redrawing is automatic, so no Paint events are necessary.
Resize ( )	fires when size of an object changes.
Special Methods	Details
Circle	Draw circle on the object Circle (X, Y) , radius
Cls	Clear the picture box. i. e. erases graphics used on the picture box
Line	Draws a line on the picture box Line (X1, Y1) - (X2, Y2) , Color, BF B stand for Box, F stand for Fill
PaintPicture	Draws the contents of a graphics file like *.bmp on the picture box object.PaintPicture picture, x1, y1

◻Image

Special Properties	Value
Picture	= and picture file sets picture file for the picture box.
Stretch	= True / False If Stretch is set to True, resizing the control also resizes the graphic it contains.
Special Events	Details
MouseMove (Button as integer, Shift as integer, x as single, y as single)	Fires when user moves mouse over the Image control. Co-ordinates or argument remains same for all the objects or controls.

◻Difference Between Picture Box and Image

Image	Picture Box

They don't support graphical methods or the AutoRedraw and the ClipControls properties.	They support graphical methods or the AutoRedraw and the ClipControls properties.
They can't work as containers, just to hint at their biggest limitations.	They can work as controls container. That why it is called forms as form.
They load faster and consume less memory and system resources.	They are heavy controls as it also consume more memory and system resources.
Image controls can load bitmaps and JPEG and GIF images.	Picture Box support variety of file formats like, BMP, DIB, WMF, EMF, GIF, JPEG, ICO, and CUR files. To know about full form for these extensions click me.
Image controls don't expose the AutoSize property because by default they resize to display the contained image.	PictureBox controls can have AutoSize property by with they resize them selves. That can be set to True / False.
Image controls support a Stretch property that, if True, resizes the image (distorting it if necessary) to fit the control. In a sense, the Stretch property somewhat remedies the lack of the PaintPicture method for this control.	Picture Box controls don't expose the Stretch property but they have PaintPicture method and are much more useful to generate vector graphics.

LoadPicture

```
Picture1.Picture = LoadPicture("c:\windows\setup.bmp")
```

will load picture to the picture at run time. This function also work with image control.

To unload picture from control use this line

```
Picture1.Picture = LoadPicture("")
```

OR

```
Set Picture1.Picture = Nothing
```

ZO0oMm......

you can zoom in to or reduce an image by loading it in an Image control and then setting its Stretch property to True to change its width and height:

```
' Load a bitmap.

Image1.Stretch = False

Image1.Picture = LoadPicture("c:\windows\setup.bmp")

' Reduce it by a factor of two.

Image1.Stretch = True

Image1.Move 0, 0, Image1.Width / 2, Image1.Width / 2
```

Command Button

Special Properties	Value
Cancel	= True / False Only one CommandButton control on a form can be the Cancel button. When the Cancel property is set to True for one CommandButton, it's automatically set to False for all other CommandButton controls on the form. When a CommandButton control's Cancel property setting is True and the form is the active form, the user can choose the CommandButton by clicking it, pressing the ESC key, or pressing ENTER when the button has the focus.
Caption	= String that represents name or user imformation on the command for the use purpose. adding an & character to associate a hot key with the control.

Default	Only one command button on a form can be the default command button. When Default is set to True for one command button, it's automatically set to False for all other command buttons on the form. When the command button's Default property setting is True and its parent form is active, the user can choose the command button (invoking its Click event) by pressing ENTER.
DisablePicture	The DisabledPicture property specifies a picture object to display when the control (such as a CommandButton) is disabled. The DisabledPicture property is ignored unless the Style property of the control is set to 1 (graphical).
DownPicture	The DownPicture property refers to a picture object that displays when the button is in the down state. The DownPicture property is ignored unless the Style property is set to 1 (graphical).
Picture	Returns or sets a graphic or picture to be displayed in a control.
Style	Returns or sets a value indicating the display type and behavior of the control. If you wish to use picture to be displayed on the command button you have to use Style property as 1 - Graphical. By default it is 0 - standard.

SPECIAL

The only relevant CommandButton's run-time property is Value, which sets or returns the state of the control (True if pressed, False otherwise). In most cases, you don't need to query this property because if you're inside a button's Click event you can be sure that the button is being activated. The Value property is useful only for programmatically clicking a button:

```
' This fires the button's Click event.

        Command1.Value = True
```

The CommandButton control supports the usual set of keyboard and mouse events (KeyDown, KeyPress, KeyUp, MouseDown, MouseMove, MouseUp, but not the DblClick event) and also the GotFocus and LostFocus events.

Check Box

Special Properties	Value
Caption	= String that represents name or user imformation on the Checkl Box for the user purpose.
DisablePicture	The DisabledPicture property specifies a picture object to display when the control (such as a Check Box) is disabled. The DisabledPicture property is ignored unless the Style property of the control is set to 1 (graphical).
DownPicture	The DownPicture property refers to a picture object that displays when the Check Box is in the down state. The DownPicture property is ignored unless the Style property is set to 1 (graphical).
Picture	Returns or sets a graphic or picture to be displayed in a control.
Style	Returns or sets a value indicating the display type and behavior of the control. If you wish to use picture to be displayed on the Check Box you have to use Style property as 1 - Graphical. By default it is 0 - standard.

Value	= 0 - UnChecked (or) = 1 - Checked (or) = 2 - Grayed

CheckBox controls are useful when you want to offer your users a yes or no, true or false choice. Anytime you click on this control, it toggles between the yes state and the no state. This control can also be grayed when the state of the CheckBox is unavailable, but you must manage that state through code.

When you place a CheckBox control on a form, all you have to do, usually, is set its Caption property to a descriptive string. You might sometimes want to move the little check box to the right of its caption, which you do by setting the Alignment property to 1-Right Justify, but in most cases the default setting is OK. If you want to display the control in a checked state, you set its Value property to 1-Checked right in the Properties window, and you set a grayed state with 2-Grayed.

The only important event for CheckBox controls is the Click event, which fires when either the user or the code changes the state of the control. In many cases, you don't need to write code to handle this event. Instead, you just query the control's Value property when your code needs to process user choices. You usually write code in a CheckBox control's Click event when it affects the state of other controls. For example, if the user clears a check box, you might need to disable one or more controls on the form and reenable them when the user clicks on the check box again. This is how you usually do it (here I grouped all the relevant controls in one frame named Frame1):

```
Private Sub Check1_Click()
```

```
        Frame1.Enabled = (Check1.Value = vbChecked)

    End Sub
```

Note that Value is the default property for CheckBox controls, so you can omit it in code. I suggest that you not do that, however, because it would reduce the readability of your code.

☐Option Button

Special Properties	Value
Caption	= String that represents name or user imformation on the option button for the user purpose.
DisablePicture	The DisabledPicture property specifies a picture object to display when the control is disabled. The DisabledPicture property is ignored unless the Style property of the control is set to 1 (graphical).
DownPicture	The DownPicture property refers to a picture object that displays when the option button is in the down state. The DownPicture property is ignored unless the Style property is set to 1 (graphical).

Picture	Returns or sets a graphic or picture to be displayed in a control.
Style	Returns or sets a value indicating the display type and behavior of the control. If you wish to use picture to be displayed on the option button you have to use Style property as 1 - Graphical. By default it is 0 - standard.
Value	= True / False if True then option button is checked else unchecked

OptionButton controls are also known as radio buttons because of their shape. You always use OptionButton controls in a group of two or more because their purpose is to offer a number of mutually exclusive choices. Anytime you click on a button in the group, it switches to a selected state and all the other controls in the group become unselected.

Preliminary operations for an OptionButton control are similar to those already described for CheckBox controls. You set an OptionButton control's Caption property to a meaningful string, and if you want you can change its Alignment property to make the control right aligned. If the control is the one in its group that's in the selected state, you also set its Valueproperty to True. (The OptionButton's Value property is a Boolean value because only two states are possible.) Value is the default property for this control.

At run time, you typically query the control's Value property to learn which button in its group has been selected. Let's say you have three OptionButton controls, named optWeekly, optMonthly, and optYearly. You can test which one has been selected by the user as follows:

```
If optWeekly.Value Then
    ' User prefers weekly frequency.
ElseIf optMonthly.Value Then
    ' User prefers monthly frequency.
ElseIf optYearly.Value Then
    ' User prefers yearly frequency.
End If
```

Strictly speaking, you can avoid the test for the last OptionButton control in its group because all choices are supposed to be mutually exclusive. But the approach I just showed you increases the code's readability.

A group of OptionButton controls is often hosted in a Frame control. This is necessary when there are other groups of OptionButton controls on the form. As far as Visual Basic is concerned, all the OptionButton controls on a form's surface belong to the same group of mutually exclusive selections, even if the controls are placed at the opposite corners of the window. The only way to tell Visual Basic which controls belong to which group is by gathering them inside a Frame control. Actually, you can group your controls within any control that can work as a container—PictureBox, for example—but Frame controls are often the most reasonable choice.

Difference Between Option Button and Check

Box

Check Box	**Option Button**
This control exposes three value for its state. That are named as 0 - UnChecked, 1 - Checked, 2 - Grayed.	This control exposes two possibility for its value property. That is True if selected and False if not.
This can be used to give multiple options to the user.	This can be used when user has only one option.
There is no need of grouping objects, infact one can use this to give information in proper manner.	more than one option button can be selected with the use of Grouping objects like Frame, or Picture Box.

Frame

Special Properties	Value
BorderStyle	= 0 - None = 1 - Fixed Single
Visible	= True / False If True then Frame will not be visible while in run mode.

Frame controls are similar to Label controls in that they can serve as captions for those controls that don't have their own. Moreover,

Frame controls can also (and often do) behave as containers and host other controls. In most cases, you only need to drop a Frame control on a form and set its Caption property. If you want to create a borderless frame, you can set its BorderStyle property to 0-None.

Controls that are contained in the Frame control are said to be child controls. Moving a control at design time over a Frame control—or over any other container, for that matter—doesn't automatically make that control a child of the Frame control. After you create a Frame control, you can create a child control by selecting the child control's icon in the Toolbox and drawing a new instance inside the Frame's border. Alternatively, to make an existing control a child of a Frame control, you must select the control, press Ctrl+X to cut it to the Clipboard, select the Frame control, and press Ctrl+V to paste the control inside the Frame. If you don't follow this procedure and you simply move the control over the Frame, the two controls remain completely independent of each other, even if the other control appears in front of the Frame control.

Frame controls, like all container controls, have two interesting features. If you move a Frame control, all the child controls go with it. If you make a container control disabled or invisible, all its child controls also become disabled or invisible. You can exploit these features to quickly change the state of a group of related controls.

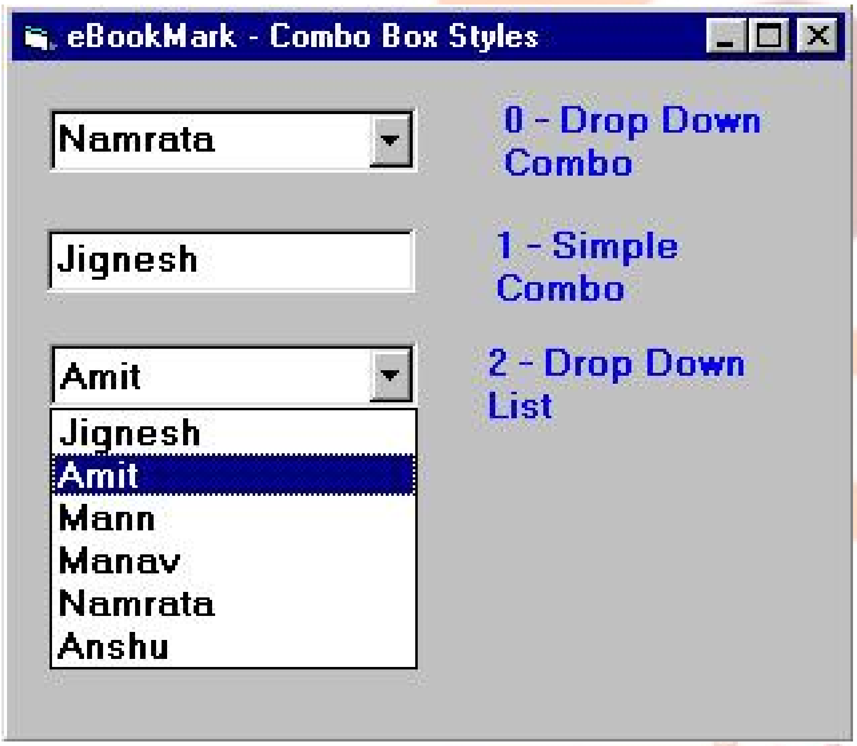
Combo Box

Special Properties	Value
ItemData	Returns or sets a specific number for each item in a ComboBox control.
List	Returns or sets the items contained in a control's list portion. The list is a string array in which each element is a list item. Available at design time for ComboBox controls through the Properties window.
Locked	For the ComboBox control, when Locked is set to True, the user cannot change any data, but can highlight data in the text box and copy it. This property does not affect programmatic access to the ComboBox.
Sorted	= True / False Returns a value indicating whether the elements of a control are automatically sorted alphabetically.

Style	= 0 - Dropdown Combo = 1 - Simple Combo = 2 - Dropdown list Returns or sets a value indicating the display type and behavior of the control. Read only at run time.
Text	Returns value available on the combo in selection
Special Events	Details
Scroll ( )	For a ComboBox control, this event occurs only when the scrollbars in the dropdown portion of the control are manipulated.
Special Methods	Details
AddItem	If you supply a valid value for index, item is placed at that position within the object. If index is omitted, item is added at the proper sorted position (if the Sorted property is set to True) or to the end of the list (if Sorted is set to False).

Clear	Clears the contents of a ComboBox. A ComboBox control bound to a Data control doesn't support the Clear method.
RemoveItem	Removes an item from a ComboBox control. A ComboBox that is bound to a Data control doesn't support the RemoveItem method.

FIGURE



You can create ComboBox controls that automatically sort their items using the Sorted property, you can add items at design time using the List item in the Properties window, and you can set a ComboBox control's IntegralHeight property as your user interface dictates. ComboBox controls don't support multiple columns and

multiple selections, so you don't have to deal with the Column, MultiSelect, Select, and SelCount properties and the ItemCheck event.

The ComboBox control is a sort of mixture between a ListBox and a TextBox control in that it also includes several properties and events that are more typical of the latter, such as the SelStart, SelLength, SelText, and Locked properties and the KeyDown, KeyPress, and KeyUp events.

The most characteristic ComboBox control property is Style, which lets you pick one among the three styles available, as you can see in Figure given above. When you set Style = 0-DropDown Combo, what you get is the classic combo; you can enter a value in the edit area or select one from the drop-down list. The setting Style = 1-Simple Combo is similar, but the list area is always visible so that in this case you really have a compounded TextBox plus ListBox control. By default, Visual Basic creates a control that's only tall enough to show the edit area, and you must resize it to make the list portion visible. Finally, Style = 2-Dropdown List suppresses the edit area and gives you only a drop-down list to choose from.

When you have a ComboBox control with Style = 0-Dropdown Combo or 2-Dropdown List, you can learn when the user is opening the list portion by trapping the DropDown event. For example, you can fill the list area just one instant before the user sees it (a sort of just-in-time data loading):

```
Private Sub Combo1_DropDown()

    Dim i As Integer

    ' Do it only once.
```

```
                If Combo1.ListCount = 0 Then
                    For i = 1 To 100
                        Combo3.AddItem "Item
                      #" & i
                    Next
                End If
        End Sub
```

The ComboBox control supports the Click and DblClick events, but they relate only to the list portion of the control. More precisely, you get a Click event when the user selects an item from the list, and you get a DblClick event only when an item in the list is double-clicked. The latter can occur only when Style = 1-Simple Combo, though, and you'll never get this event for other types of ComboBox controls.

ComboBox controls with Style = 1-Simple Combo possess an intriguing feature, called extended matching. As you type a string, Visual Basic scrolls the list portion so that the first visible item in the list area matches the characters in the edit area.

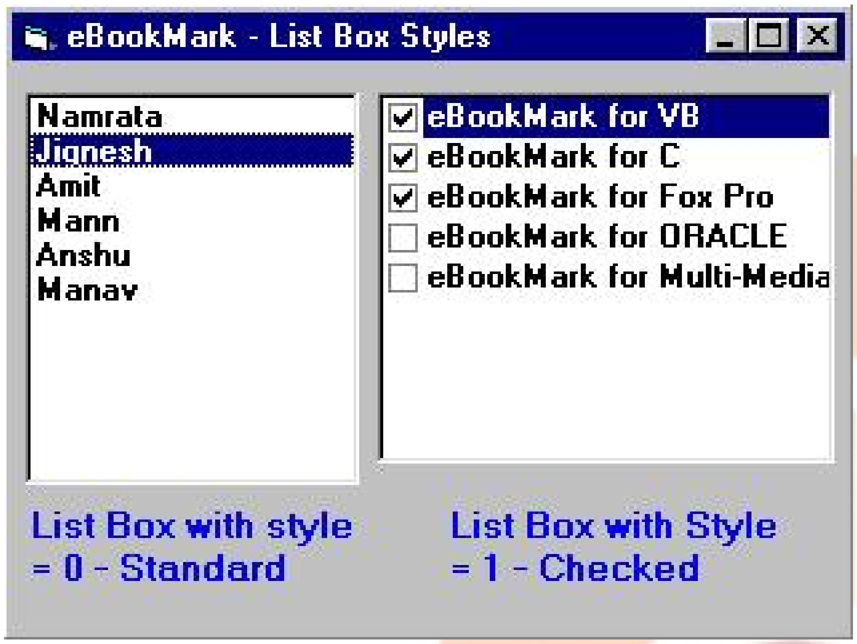
List Box

Special Properties	Value

ItemData	Returns or sets a specific number for each item in a List Box control.
List	Returns or sets the items contained in a control's list portion. The list is a string array in which each element is a list item. Available at design time for ListBox controls through the Properties window.
MultiSelect	= 0 - None = 1 - Simple = 2 - Extended Explain in details below
Sorted	= True / False Returns a value indicating whether the elements of a control are automatically sorted alphabetically.
Style	= 0 - Standard = 1 - Checked Returns or sets a value indicating the display type and behavior of the control. Read only at run time.
Special Events	Details

Scroll ( )	For a List Box control, this event occurs only when the scrollbars in the dropdown portion of the control are manipulated.
Special Methods	Details
AddItem	If you supply a valid value for index, item is placed at that position within the object. If index is omitted, item is added at the proper sorted position (if the Sorted property is set to True) or to the end of the list (if Sorted is set to False).
Clear	Clears the contents of a List Box. A List Box control bound to a Data control doesn't support the Clear method.
RemoveItem	Removes an item from a List Box control. A List Box that is bound to a Data control doesn't support the RemoveItem method.

FIGURE


eBookMark - List Box Styles
Namrata
Jignesh
Amit
Mann
Anshu
Manav
eBookMark for VB
eBookMark for C
eBookMark for Fox Pro
eBookMark for ORACLE
eBookMark for Multi-Media
List Box with style = 0 - Standard
List Box with Style = 1 - Checked

you set the Sorted attribute to True to create ListBox controls that automatically sort their items in alphabetical order. By acting on the Columns property, you create a different type of list box, with several columns and a horizontal scroll bar, you can't change the style of the ListBox control while the program is running.

If you know at design time which items must appear in the ListBox control, you can save some code and enter the items right in the Properties window, in the List property mini-editor

Both ListBox and ComboBox controls expose the AddItem method, which lets you add items when the program is executing. You usually use this method in the Form_Load event procedure:

```
Private Sub Form_Load()

  List1.AddItem "First"

  List1.AddItem "Second"

  List1.AddItem "Third"
```

End Sub

TIP

If you want to load many items in a list box but don't want to create an array, you can resort to Visual Basic's Choose function, as follows:

```
For i = 1 To 5
  List1.AddItem Choose(i, "America", "Europe", "Asia", "Africa", "Australia")
Next
```

Sometimes you need to add an item in a given position, which you do by passing a second argument to the AddItem method. (Note that indexes are zero-based.)

```
' Add at the very beginning of the list.
List1.AddItem "Zero", 0
```

Removing items is easy with the RemoveItem or Clear methods:

```
' Remove the first item in the list.
List1.RemoveItem 0
' Quickly remove all items (no need for a For...Next loop).
List1.Clear
```

The most obvious operation to be performed at run time on a filled ListBox control is to determine which item has been selected by the user. The ListIndex property returns the index of the selected item (zero-based), while the Text property returns the actual string stored in the ListBox. The ListIndex property returns -1 if the user hasn't selected any element yet, so you should test for this condition first:

```
If List1.ListIndex = -1 Then

  MsgBox "No items selected"

Else

  MsgBox "User selected " & List1.Text & " (#" & List1.ListIndex & ")"

End If
```

The ListCount property returns the number of items in the control. You can use it with the List property to enumerate them:

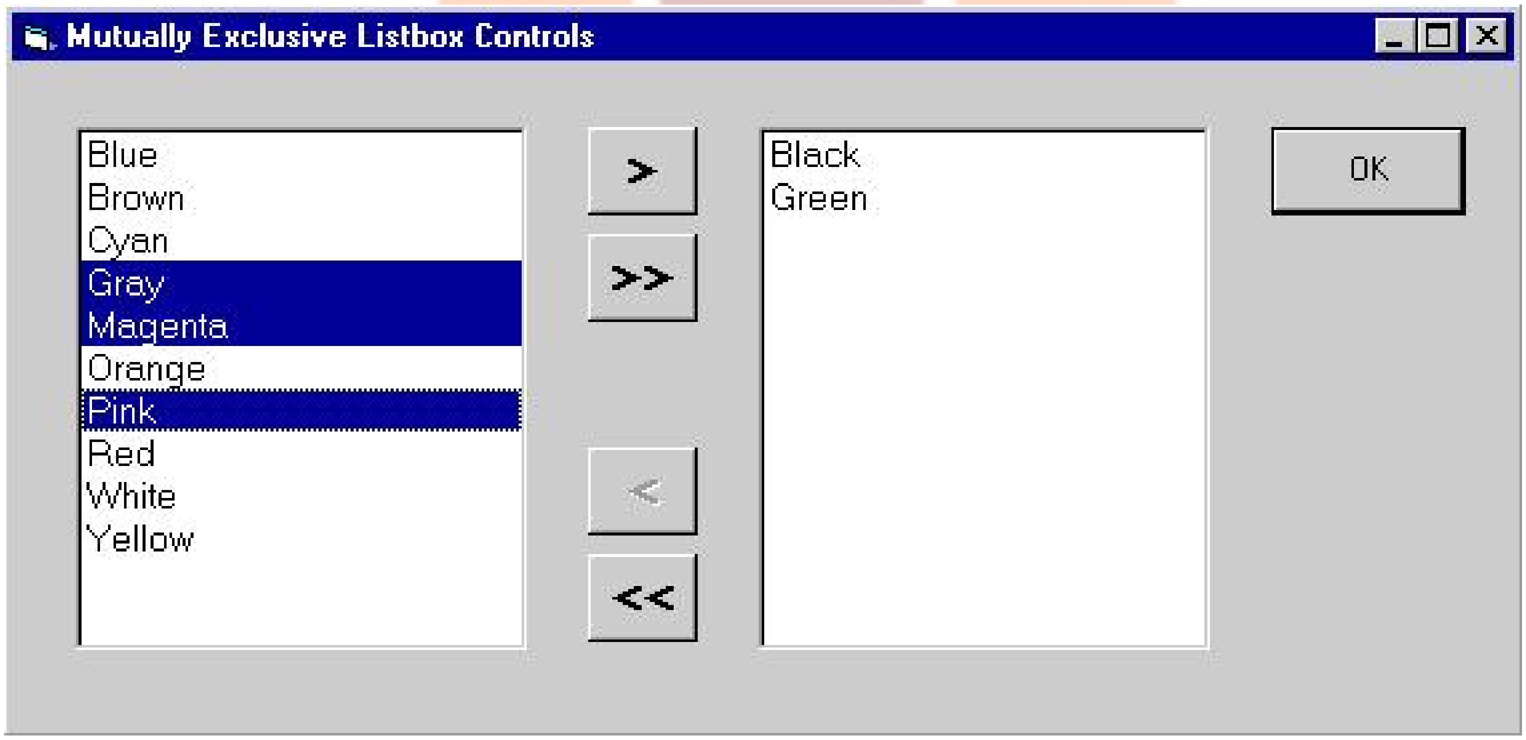
```
For i = 0 To List1.ListCount -1

  Print "Item #" & i & " = " & List1.List(i)

Next
```

❐Difference Between Combo Box and List Box

Combo Box	List Box
ComboBox controls don't support multiple columns and multiple selections	
Combo box don't have to deal with the Column, MultiSelect, Select, and SelCount properties and the ItemCheck event.	
Locked	
	Multiselect

❐Example

The logic behind your user interface might require that you monitor the DblClick event as well. As a general rule, double-clicking on a ListBox control's item should have the same effect as selecting the item and then clicking on a push button (often the default push button on the form). Take, for example, the mutually exclusive ListBox controls shown in Figure 3-8, a type of user interface that you see in many Windows applications. Implementing this structure in Visual Basic is straightforward:



```
Private Sub cmdMove_Click()

  ' Move one item from left to right.

  If lstLeft.ListIndex >= 0 Then

  lstRight.AddItem lstLeft.Text

  lstLeft.RemoveItem lstLeft.ListIndex
  End If
End Sub


Private Sub cmdMoveAll_Click()
  ' Move all items from left to right.
  Do While lstLeft.ListCount
  lstRight.AddItem lstLeft.List(0)
  lstLeft.RemoveItem 0
  Loop
End Sub


Private Sub cmdBack_Click()
  ' Move one item from right to left.
  If lstRight.ListIndex >= 0 Then
  lstLeft.AddItem lstRight.Text
  lstRight.RemoveItem lstRight.ListIndex
  End If
End Sub
```

```
Private Sub cmdBackAll_Click()
  ' Move all items from right to left.
  Do While lstRight.ListCount
  lstLeft.AddItem lstRight.List(0)
  lstRight.RemoveItem 0
  Loop
End Sub

Private Sub lstLeft_DblClick()
  ' Simulate a click on the Move button.
  cmdMove.Value = True
End Sub

Private Sub lstRight_DblClick()
  ' Simulate a click on the Back button.
  cmdBack.Value = True
End Sub
```

☐Scroll Bars

- Horizontal Scroll Bar
- Vertical Scroll Bar

We have two category in Visual Basic as far as scroll bar is concern. The difference between Vertical and Horizontal Scroll Bar is the lay out only.

Special Properties	Value
LargeChange	
Min	
Max	
SmallChange	
Value	
Special Events	Details
Scroll ( )	

few properties: Min and Max represent the valid range of values,

SmallChange is the variation in value you get when clicking on the scroll bar's arrows, and LargeChange is the variation you get when you click on either side of the scroll bar indicator.

The most important run-time property is Value, which always returns the relative position of the indicator on the scroll bar. By default, the Min value corresponds to the leftmost or upper end of the control:

```
' Move the indicator near the top (or left) arrow.

VScroll1.Value = VScroll1.Min

' Move the indicator near the bottom (or right) arrow.

VScroll1.Value = VScroll1.Max
```

There are two key events for scrollbar controls: the Change event fires when you click on the scroll bar arrows or when you drag the indicator; the Scroll event fires while you drag the indicator.

```
' Show the current scroll bar's value.

Private VScroll1_Change()

  Label1.Caption = VScroll1.Value

End Sub

Private VScroll1_Scroll()

  Label1.Caption = VScroll1.Value

End Sub
```

.they support both the TabIndex and TabStop properties. If you don't want the user to accidentally move the input focus on a scrollbar control when he or she presses the Tab key, you must explicitly set its TabStop property to False. When a scrollbar control has the focus, you can move the indicator using the Left, Right, Up, Down, PgUp, PgDn, Home, and End keys.

Example

RGB function example

☐Timer

Special Properties	Value
Enabled	
Interval	
Special Events	Details
Timer ( )	

A Timer control is invisible at run time, and its purpose is to send a periodic pulse to the current application. You can trap this pulse by writing code in the Timer's Timer event procedure and take advantage of it to execute a task in the background or to monitor a user's actions. This control exposes only two meaningful properties: Interval and Enabled. Interval stands for the number of milliseconds between subsequent pulses (Timer events), while Enabled lets you activate or deactivate events. When you place the Timer control on a form, its Interval is 0, which means no events. Therefore, remember to set this property to a suitable value in the Properties window or in the Form_Load event procedure:

```
Private Sub Form_Load()
  Timer1.Interval = 500 ' Fire two Timer events per second.
End Sub
```

CAUTION

You must be careful not to write a lot of code in the Timer event procedure because this code will be executed at every pulse and therefore can easily degrade your application's performance. Just as important, never execute a DoEvents statement inside a Timer event procedure because you might cause the procedure to be reentered, especially if the Interval property is set to a small value and there's a lot of code inside the procedure.

Timer controls are often useful for updating status information on a regular basis. For example, you might want to display on a status bar a short description of the control that currently has the input focus. You can achieve that by writing some code in the GotFocus event for all the controls on the form, but when you have dozens of controls this will require a lot of code (and time). Instead, at design time load a short description for each control in its Tag property, and then place a Timer control on the form with an Interval setting of 500. This isn't a time-critical task, so you can use an even larger value. Finally add two lines of code to the control's Timer event:

```
Private Sub Timer1_Timer()

  On Error Resume Next
```

```
 lblStatusBar.Caption = ActiveControl.Tag

End Sub
```

☐Drive List Box

Special Events	Details
Change ( )	
Scroll ( )	

☐Dir List Box

Special Events	Details
Change ( )	
Scroll ( )	

☐File List Box

Special Properties	Value
Archive	

Hidden	
MultiSelect	
Normal	
Pattern	
ReadOnly	
System	
Special Events	Details
PathChange ( )	
PatternChange ( )	
Scroll ( )	

the DriveListBox control is a combobox-like control that's automatically filled with your drive's letters and volume labels. The DirListBox is a special list box that displays a directory tree. The FileListBox control is a special-purpose ListBox control that displays all the files in a given directory,

These controls often work together on the same form; when the user selects a drive in a DriveListBox, the DirListBox control is updated to show the directory tree on that drive. When the user selects a path in the DirListBox control, the FileListBox control is filled with the list of files in that directory. These actions don't happen automatically, however—you must write code to get the job done.

After you place a DriveListBox and a DirListBox control on a form's surface, you usually don't have to set any of their properties; in fact, these controls don't expose any special property, not in the Properties window at least. The FileListBox control, on the other hand, exposes one property that you can set at design time—the Pattern property. This property indicates which files are to be shown in the list area: Its default value is *.* (all files), but you can enter whatever specification you need, and you can also enter multiple specifications using the semicolon as a separator. You can also set this property at run time, as in the following line of code:

```
File1.Pattern = "*.txt;*.doc;*.rtf"
```

After these preliminary steps, you're ready to set in motion the chain of events. When the user selects a new drive in the DriveListBox control, it fires a Change event and returns the drive letter (and volume label) in its Drive property. You trap this event and set the DirListBox control's Path property to point to the root directory of the selected drive:

```
Private Sub Drive1_Change()
```

```
  ' The Drive property also returns the volume label, so trim it.

  Dir1.Path = Left$(Drive1.Drive, 1) & ":\"

End Sub
```

When the user double-clicks on a directory name, the DirListBox control raises a Change event; you trap this event to set the FileListBox's Path property accordingly:

```
Private Sub Dir1_Change()

  File1.Path = Dir1.Path

End Sub
```

Finally, when the user clicks on a file in the FileListBox control, a Click event is fired , and you can query its Filename property to learn which file has been selected. Note how you build the complete path:

The DirListBox and FileListBox controls support most of the properties typical of the control they derive from—the ListBox control—including the ListCount and the ListIndex properties and the Scroll event. The FileListBox control supports multiple selection; hence you can set its MultiSelect property in the Properties window and query the SelCount and Selected properties at run time.

The FileListBox control also exposes a few custom Boolean properties, Normal, Archive, Hidden, ReadOnly, and System, which permit you to decide whether files with these attributes should be

listed. (By default, the control doesn't display hidden and system files.) This control also supports a couple of custom events, PathChange and PatternChange, that fire when the corresponding property is changed through code.

☐Line

Special Properties	Value
BorderColor	the color of the line
BorderStyle	
BorderWidth	
DrawMode	
X1	
Y1	
X2	
Y2	
Special Events	Details
It poses not events	

The Line control is a decorative control whose only purpose is let you draw one or more straight lines at design time, instead of displaying them using a Line graphical method at run time. This control exposes a few properties whose meaning should sound familiar to you by now: BorderColor (the color of the line),

BorderStyle (the same as a form's DrawStyle property), BorderWidth (the same as a form's DrawWidth property), and DrawMode. While the Line control is handy, remember that using a Line method at run time is usually better in terms of performance.

Here X1, Y1 are line starting co-ordinates and X2, Y2 are line termination co-ordinates. Line control can be used to generate graphic on the application. But same time graphic function such as Line (X1, Y1) - (X2, Y2) , Color, BF is much faster and better to generate line at run time.

◻Shape

Special Properties	Value
BackStyle	= 0 - Transaparent = 1 - Opaque
BorderColor	
BorderStyle	= 0 - Transparent = 1 - Solid
BorderWidth	
FillColor	
FillStyle	

Shape	display six basic shapes: Rectangle, Square, Oval, Circle, Rounded Rectangle, and Rounded Square
Special Events	Details
It poses not events	

In a sense, the Shape control is an extension of the Line control. It can display six basic shapes: Rectangle, Square, Oval, Circle, Rounded Rectangle, and Rounded Square. It supports all the Line control's properties and a few more: BorderStyle (0-Transparent, 1-Solid), FillColor, and FillStyle (the same as a form's properties with the same names). The same performance considerations can be pointed out for the Line control apply to the Shape control. It is better to generate Graphics on the form with the use of Graphic functions like Line, Circle, etc. They are faster and Much more different because they are Vector graphics.

To know more about Graphics Click me.

Dialog boxes

are a special type of form object that you can create in one of three ways:

- **Predefined dialog boxes can be created from code using the MsgBox or InputBox functions.**

- **Custom dialog boxes can be created using a standard form or by customizing an existing dialog box.**

- **Standard dialog boxes, such as Print and File Open, can be created using the CommonDialog control.**

In Windows-based applications, dialog boxes are used to prompt the user for data the application needs before continuing or to give information to the user.

☐Message Box

The MsgBox function uses the following syntax:

MsgBox(prompt [, buttons] [, title] [, helpFile, context])

The arguments for the MsgBox function are described in the following table.

Argument	**Description**
prompt	Text that contains message to the user.
buttons	Determines the number of and type of message box buttons as well as the type of symbol that appears on the message box.For more information about using MsgBox function constants, search for "MsgBox arguments" in MSDN Help.
title	The text that appears in the title bar of the message box.
helpFile	A string that identifies the Help file to use to provide context-sensitive Help for the dialog box. If helpfile is provided, context must also be provided.
context	Numeric expression that identifies the specific topic in the Help file that appears. If context is provided, helpfile must also be provided.

Note : Any arguments not enclosed in square brackets are required.

You must supply a value for these arguments. Arguments that are enclosed in brackets are optional. If values are not supplied for these arguments, Visual Basic will use the default value. The following instruction shows a simple message box that is generated by the MsgBox function.

MsgBox "This illustration shows a simple message box that is generated by the MsgBox function.",vbOkOnly + vbInformation, "eBookMark"

Often the MsgBox function is used to get a simple response from the user. You can assign the value returned by the MsgBox function to a variable, and then write code to handle the response.

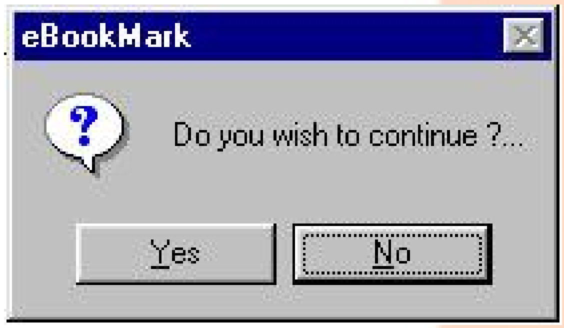


Fig. 2 can be treated with example as

```
Dim strMsg As String

Dim strTitle As String

Dim lngStyle As Long

Dim intResponse As Integer
```

```
strMsg = "Do you want to continue?"

lngStyle = vbYesNo + vbQuestion + vbDefaultButton2

strTitle = "MsgBox Demonstration"



intResponse = MsgBox(strMsg, lngStyle, strTitle)



If intResponse = vbYes Then

  'User chose Yes button

Else

  'User chose No button

End If
```

MsgBox ( ) constants

Constant	Value	Description
vbOkOnly	0	Display OK button only
vbOkCancel	1	Display OK and Cancel buttons

vbAbortRetryIgnore	2	Display Abort, Retry and Ignore buttons
vbYesNoCancel	3	Display Yes, No and Cancel buttons
vbYesNo	4	Display Yes and No Buttons
vbRetryCancel	5	Display Retry and Cancel Buttons
vbCritical	16	Display critical message icon
vbQuestion	32	Display Warning Query icon
vbExclamation	48	Display warning message icon
vbInformation	64	Display information Message icon
vbDefaultButton1	0	First Button is deafult
vbDefaultButton2	256	Second Button is deafult
vbDefaultButton3	512	Third Button is deafult
vbDefaultButton4	768	Fourth Button is deafult
vbApplicationModel	0	Application modal; the user must respond to the message box before continuing work in the current application

vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box
vbMsgBoxHelpButton	16384	Adds Help button to the message box
vbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right-aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right - to - left reading on Hebrew and Arabic system

MsgBox ( ) return values

Constant	Value	Description
vbOk	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIngore	5	Ignore

vbYes	6	Yes
vbNo	7	No

Input Box

Input boxes are an easy way for you to ask the user for input that can't be answered simply by clicking a button.

MyValue = InputBox ("This illustration shows Input box that is generated by the Input Box function.", "eBookMark", "Default Value")

below is a sintex for input box :

InputBox(prompt[, title] [, default] [, xPos] [, yPos] [, helpFile, context])

Note

For user input that requires no more than clicking a button, use the MsgBox function. When using the InputBox function, there is little control over the components of the dialog box. Only the text in the title bar, the command prompt displayed to the user, the position of the dialog box on the screen, and whether or not the dialog box displays a Help button can be changed. The MsgBox function provides more control over the appearance of the dialog box.

☐Custom dialog boxes

User can create own dialog boxes which can be used as and when required are called custom dialog boxes. Visual Basic navigates with some pre-designed dialog boxes which can be used as custom dialog boxes. These dialog boxes can be viewed in Template folder and even one can add more forms or other objects of own choice in the same folder. Here are some tips for developing a custom dialog box.

Control	**Property**	**Value (suggested)**
Form	Border Style	3 - Fixed Dialog
	ControlBox	False
	HelpContentId	0
	Moveable	False
	StartUpPosition	1 - Center Owner 2 - Center Screen
Command Button	Cancel	True
	Default	True
Text Box	PasswordChar	*

Here are some examples of custom dialog boxes which navigates with Visual Basic :

- ## About Screen

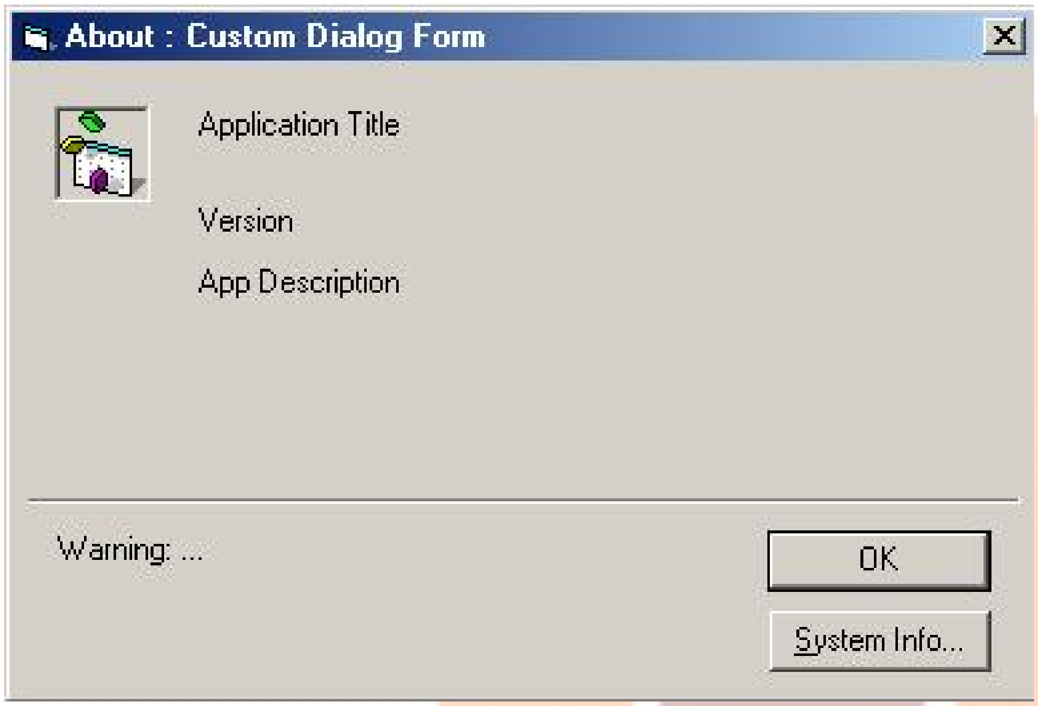
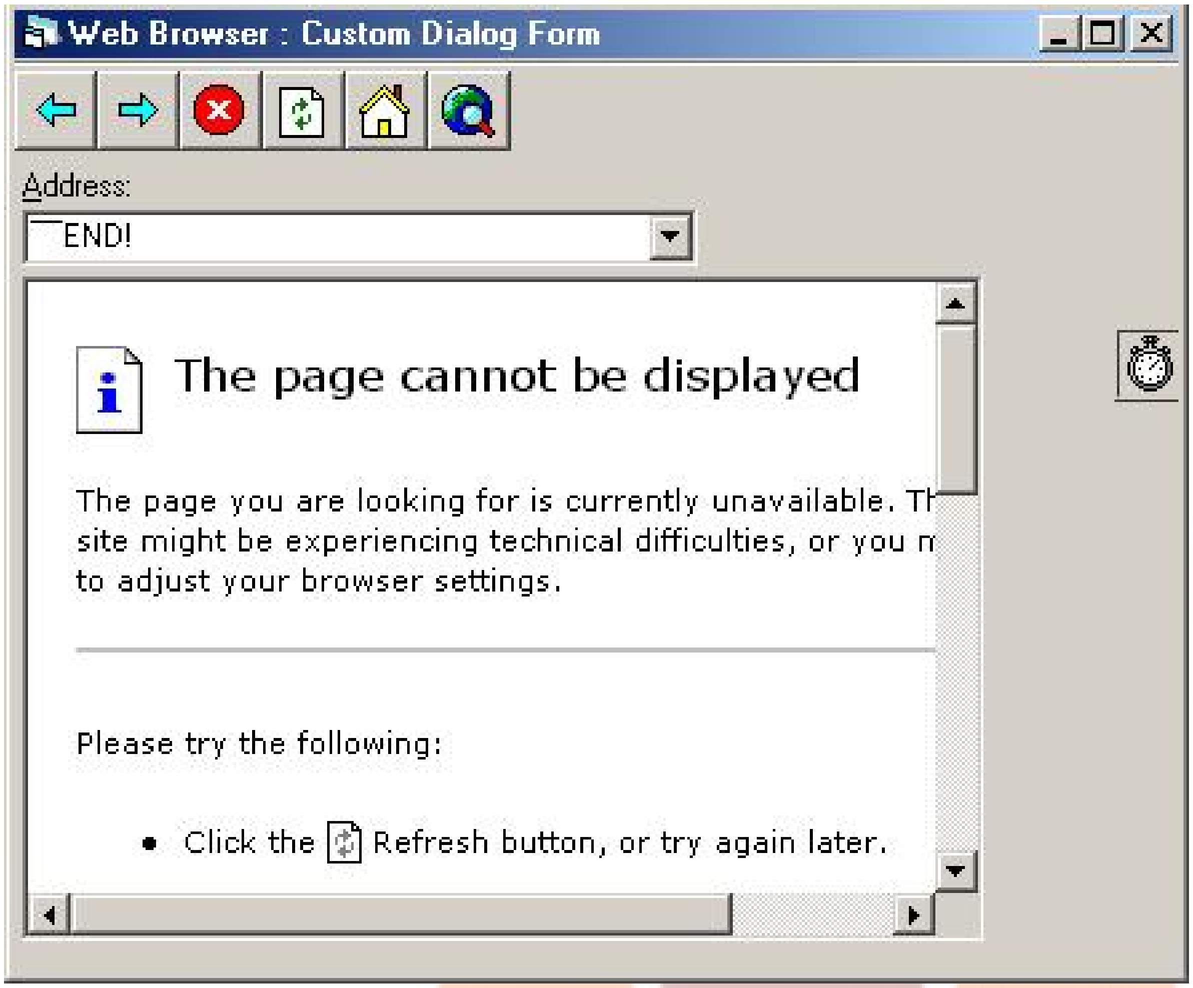


Fig. 1 shows about screen is useful for showing application related details.

- ## Web Browser



The page cannot be displayed

The page you are looking for is currently unavailable. Th
site might be experiencing technical difficulties, or you m
to adjust your browser settings.

Please try the following:

- Click the Refresh button, or try again later.

Fig. 2 represents Web browser and holds all the features of Internet Explorer.

- ## Dialog Screen

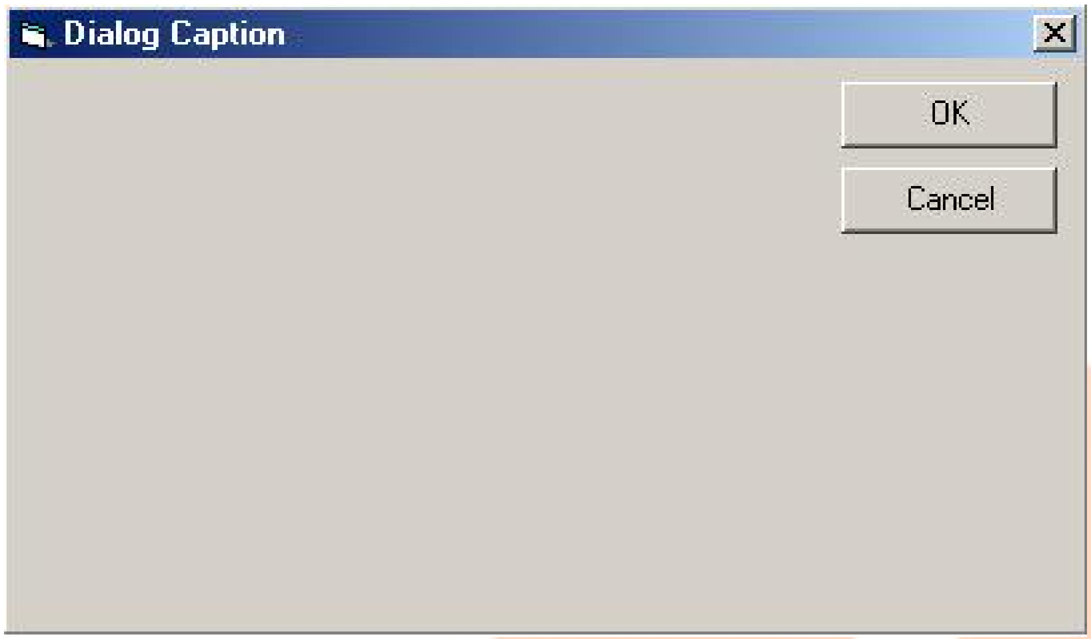


Fig. 3 can be used to generate custom dialog box

- ## Log in Dialog

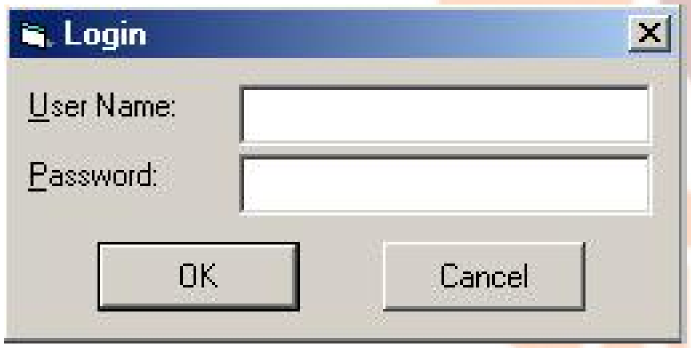


Fig. 4 is the same screen which people use for getting user identification in routine applications.

- ## Splash Screen

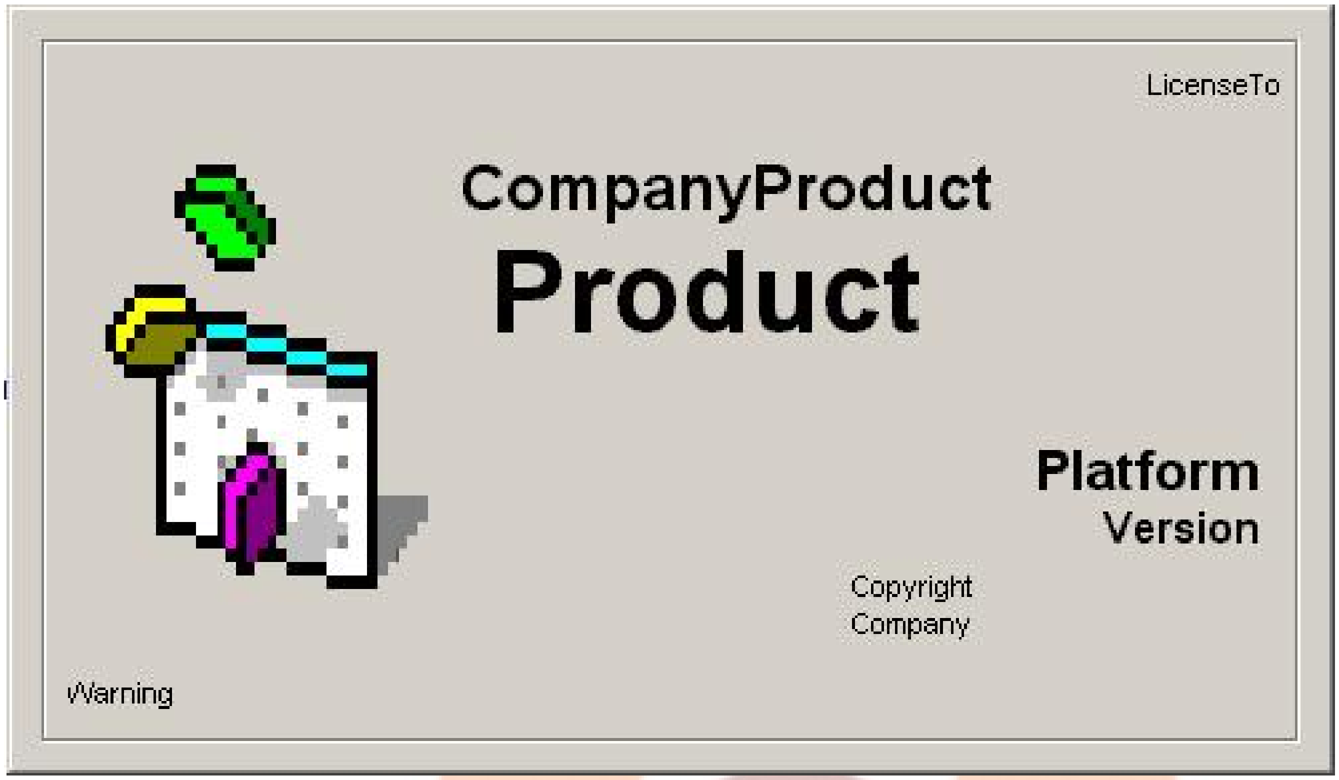


Fig. 5 is company and product representation screen and is called as Splash Screen.

- ## Tip of the Day

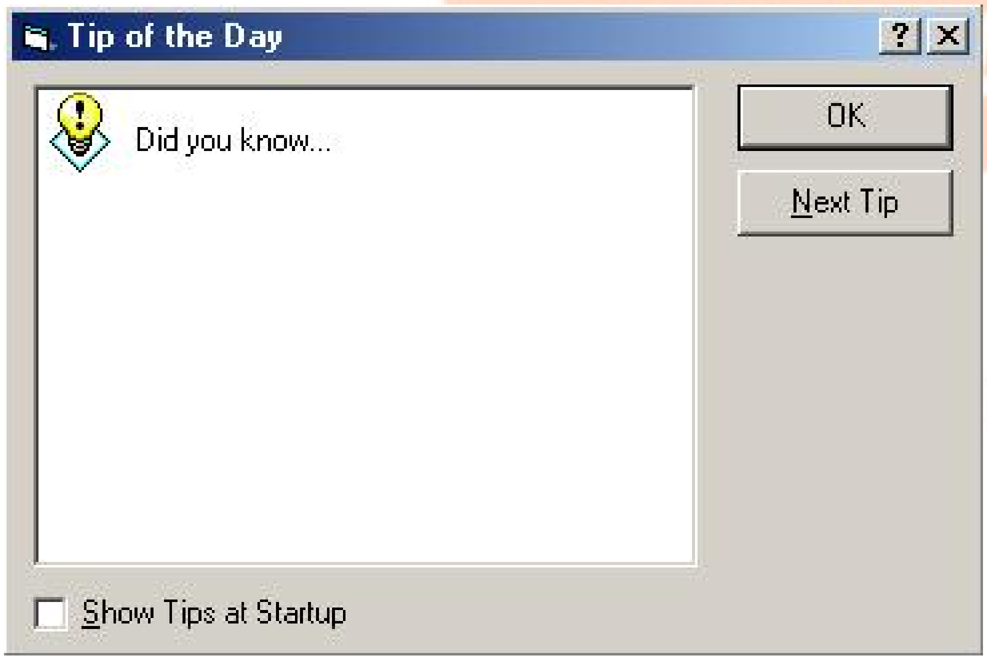
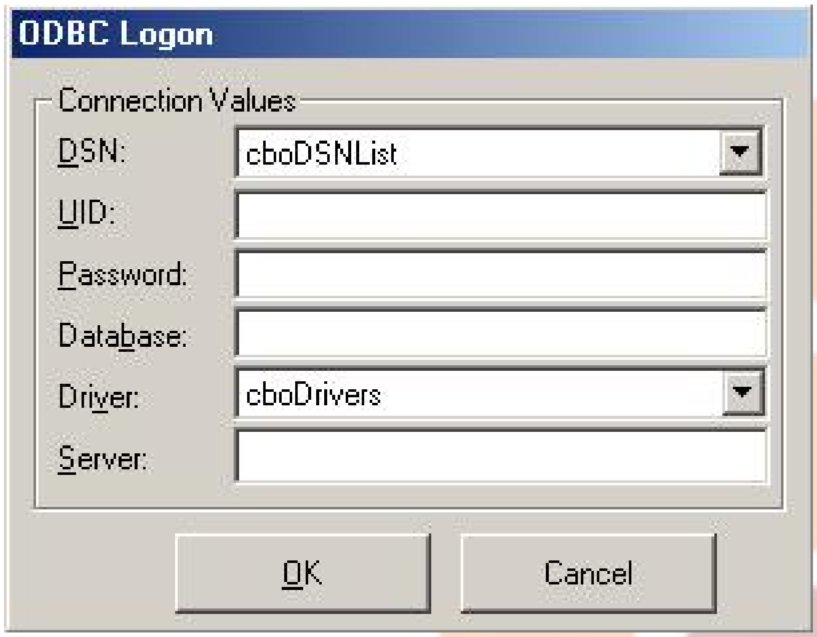


Fig. 6 shows Tips as and when application starts.

- ## ODBC Log in


ODBC Logon

Connection Values

DSN: cboDSNList

UID:

Password:

Database:

Driver: cboDrivers

Server:

Fig. 7 To connect ODBC data source on requires more code writting, if this screen is not there available in Templates.

- ## Options Dialog

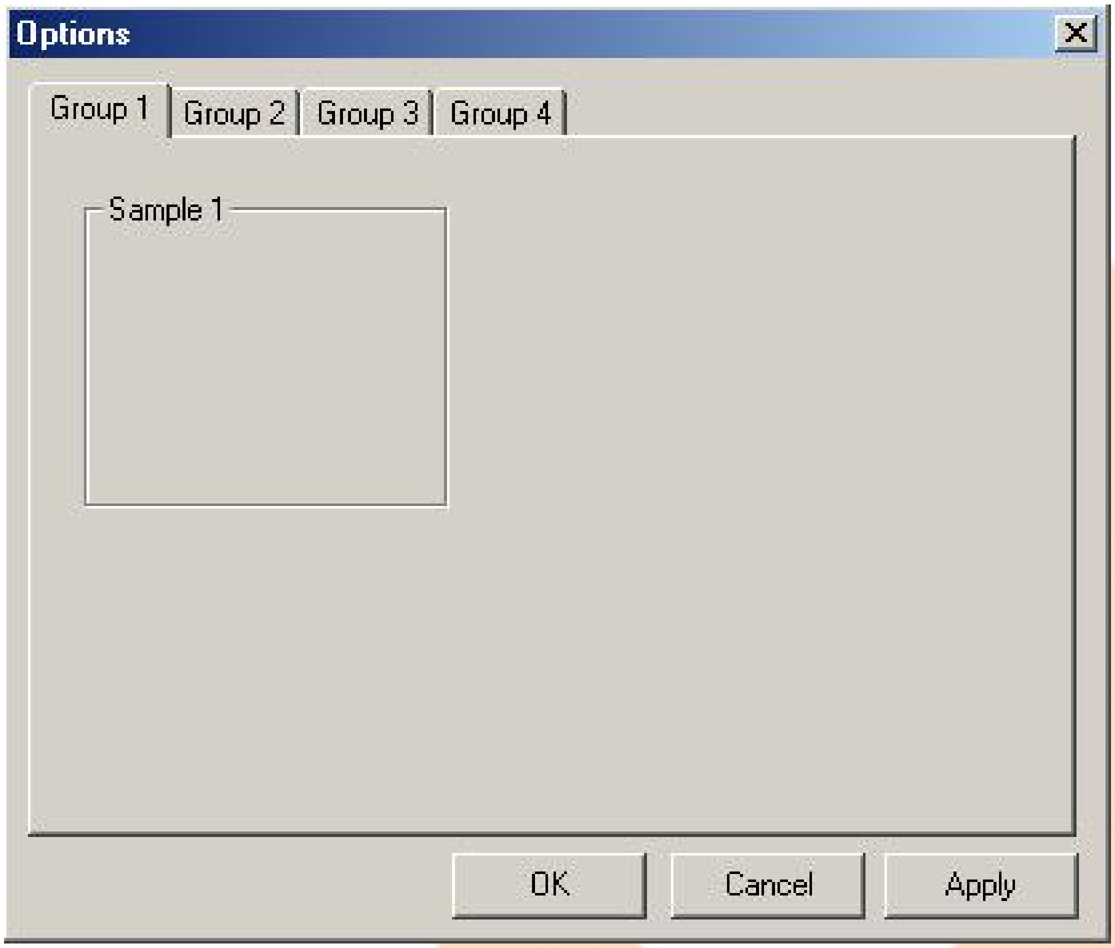
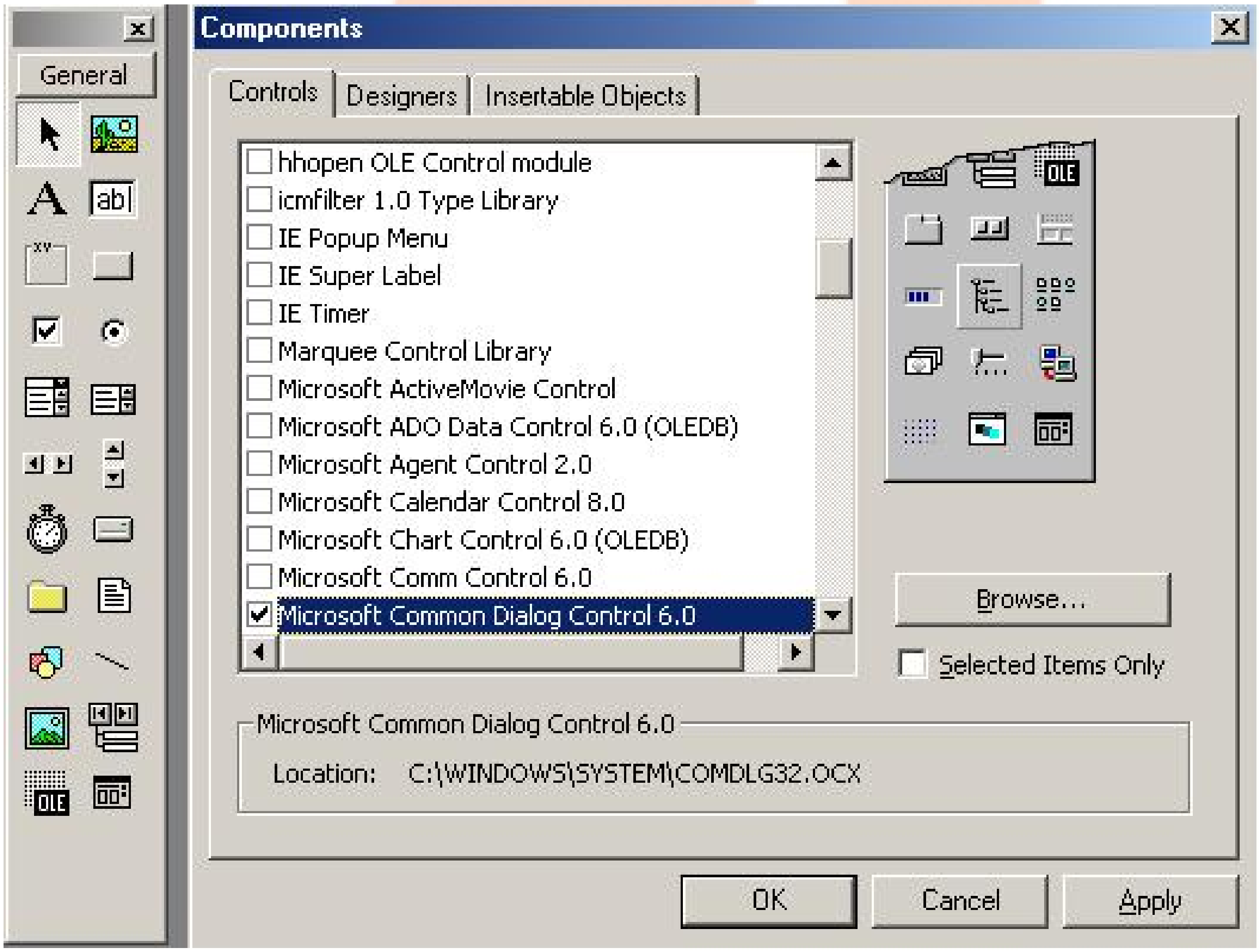


Fig. 8 is useful for represent multi control form in single form with grouped data collection tabs, is say option dialog box.

Standard dialog boxes

The CommonDialog control provides an easy and convenient way to invoke the Color, Font, Printer, FileOpen, and FileSave Windows common dialog boxes. This control exposes only properties and methods—no events. The control is invisible during the execution, so it doesn't support properties such as Left, Visible, or TabIndex. This control is embedded in the ComDlg32.ocx file, which has to be distributed with any Visual Basic application that uses it.

Following figure shows OCX control name and also shows control in Toolbar (Last Control)



- [ ] hhopen OLE Control module
- [ ] icmfilter 1.0 Type Library
- [ ] IE Popup Menu
- [ ] IE Super Label
- [ ] IE Timer
- [ ] Marquee Control Library
- [ ] Microsoft ActiveMovie Control
- [ ] Microsoft ADO Data Control 6.0 (OLEDB)
- [ ] Microsoft Agent Control 2.0
- [ ] Microsoft Calendar Control 8.0
- [ ] Microsoft Chart Control 6.0 (OLEDB)
- [ ] Microsoft Comm Control 6.0
- [✓] Microsoft Common Dialog Control 6.0

Microsoft Common Dialog Control 6.0

Location: C:\WINDOWS\SYSTEM\COMDLG32.OCX

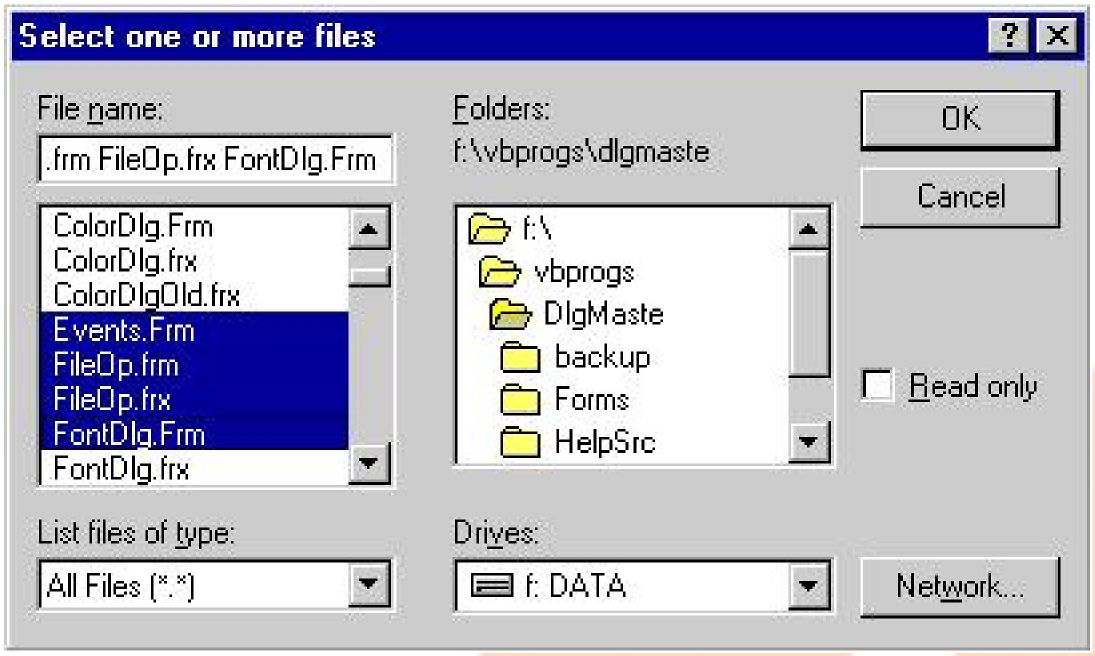
One of the few properties that can have the same meaning regardless of which common dialog box you're displaying is CancelError. If this property is True, an end user closing the dialog box using the Cancel key causes error 32755 (equal to the constant cdlCancel) to be raised in the calling program. The CommonDialog control includes intrinsic constants for all the errors that can be generated at run time. All common dialog boxes also share a few properties related to help support. You can display a Help button in the common dialog box and tell the CommonDialog control what page in what help file must be displayed when the user clicks the Help button. HelpFile is the complete name of the help file, HelpContext is the context ID of the requested page, and HelpCommand is the action that must be performed when the button is clicked. (It's usually assigned the value 1-cdlHelpContext.) Don't forget that to actually display the Help button, you must set a bit in the Flags property. The position of this bit varies with the particular common dialog box, for example:

```
' Show a Help button.

CommonDialog1.HelpFile = "F:\vbprogs\DlgMaste\Tdm.hlp"

CommonDialog1.HelpContext = 12

CommonDialog1.HelpCommand = cdlHelpContext

' The value for the Flags property depends on the dialog.

If ShowColorDialog Then
```

```
        CommonDialog1.Flags = cdlCCHelpButton

        CommonDialog1.ShowColor

    ElseIf ShowFontDialog Then

        CommonDialog1.Flags = cdlCFHelpButton

        CommonDialog1.ShowFont

    Else

        ' And so on

    End If
```

The CommonDialog control exposes six methods: ShowColor, ShowFont, ShowPrinter, ShowOpen, ShowSave, and ShowHelp. Each method displays a different common dialog box, as explained in the following sections.

Open Dialog Box



Following code shows how to display open dialog box with multiple file(s) open option

```
  CD.Filter = "All files (*.*)|*.*|" & Filter

  CD.FilterIndex = 1

  CD.Flags = cdlOFNAllowMultiselect Or cdlOFNFileMustExist
Or _

  cdlOFNExplorer

  CD.DialogTitle = "Select one or more files"

  CD.Filename = ""
```

```
' Exit if user presses Cancel.

CD.CancelError = True

CD.ShowOpen
```

☐Save Dialog Box

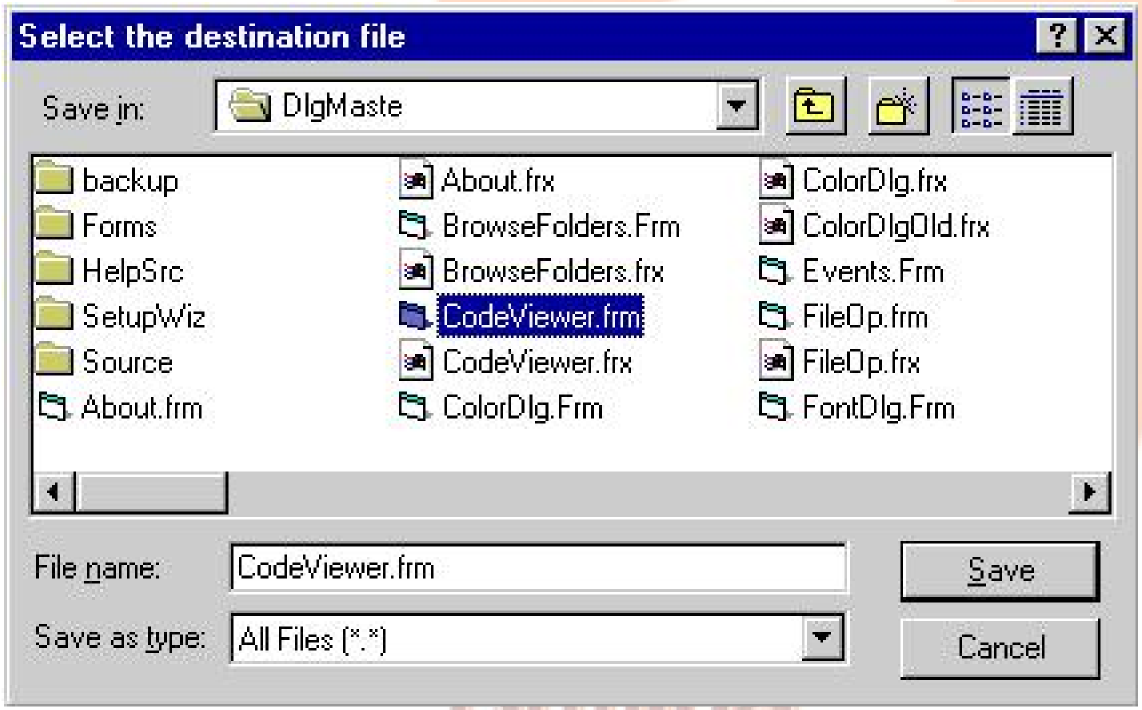
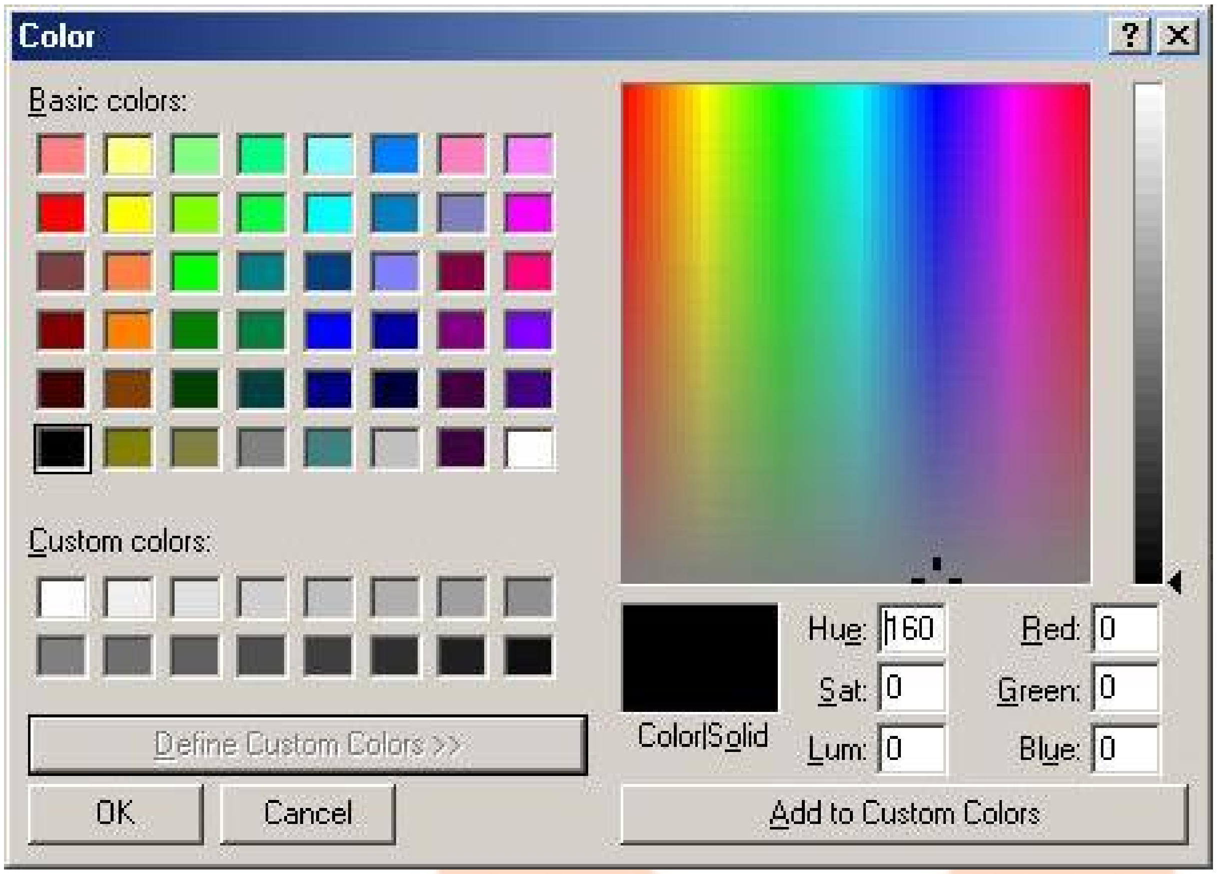


Figure shows save dialog box. Workes almost same like open dialog box.

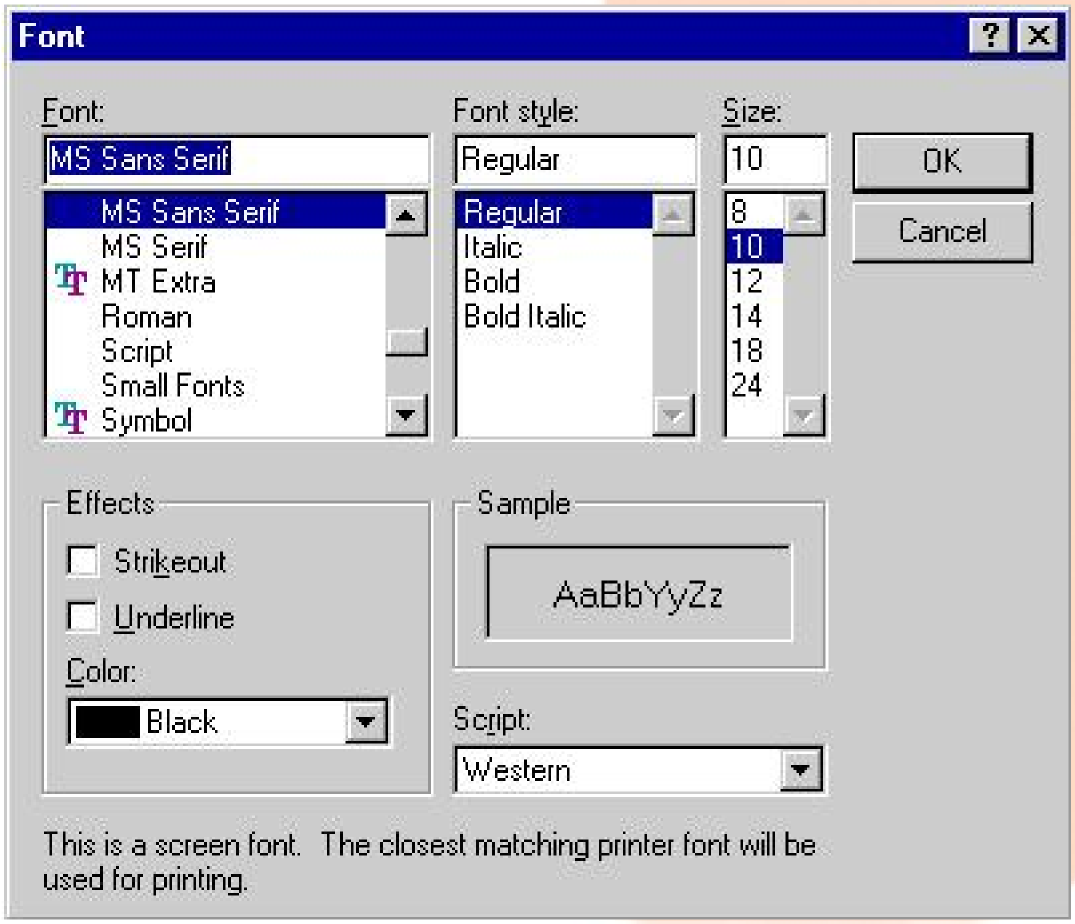
☐Color Dialog Box



Following code shows how one can apply color selection to the backbround of form with the use of color dialog box.

```
Private Sub cmdColor_Click()

  CD1.ShowColor '---shows color dialog box

  Form1.BackColor = CD1.Color '---sets user selected color to
form back

End Sub
```

Font Dialog Box



This is a screen font. The closest matching printer font will be used for printing.

Code mentioned below shows that font can be manipulate with the use of font dialog box.

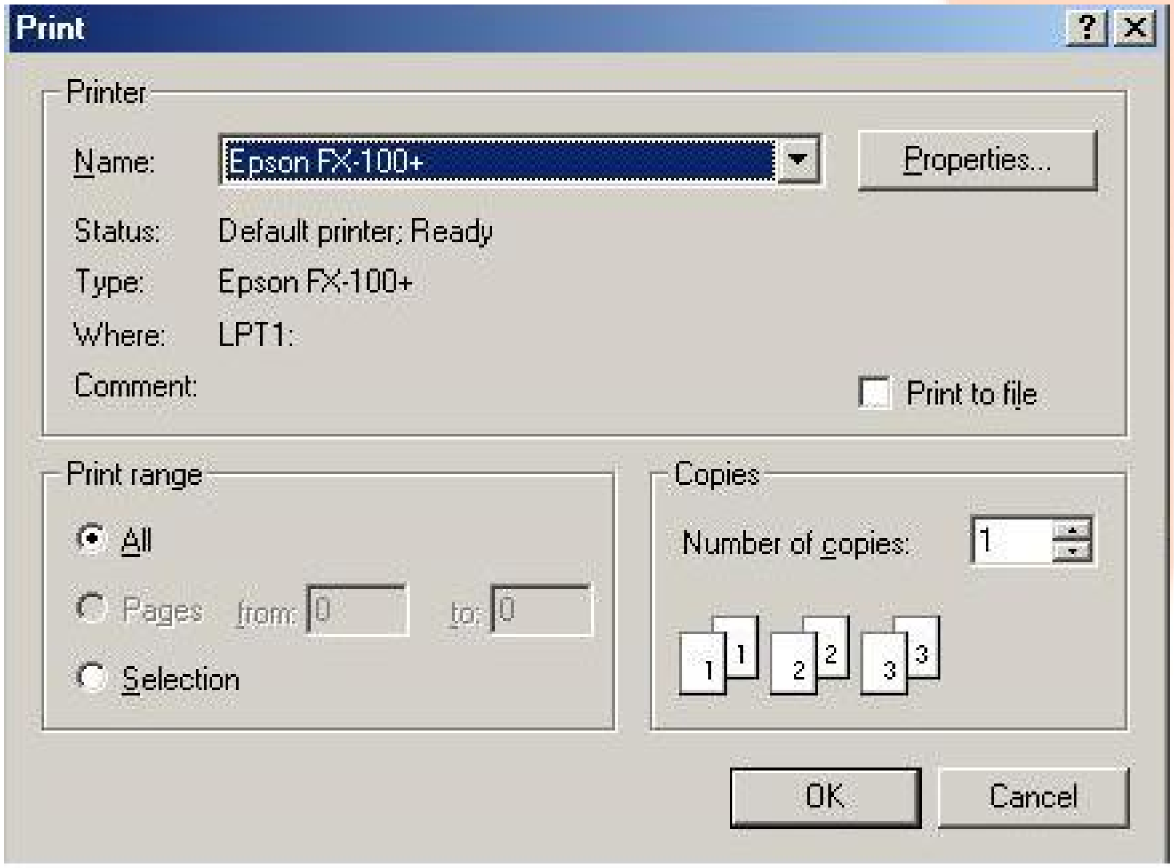
```
Private Sub cmdFont_Click()

       On Error GoTo ErrorHandler

         CD1.CancelError = True

         CD1.ShowFont '---shows font dialog box

  '---now set form properties as represented by user in font dialog box

         Form1.FontName = CD1.FontName

         Form1.FontSize = CD1.FontSize

         Form1.FontBold = CD1.FontBold

         Form1.FontItalic = CD1.FontItalic


       ErrorHandler:

         Exit Sub

End Sub
```

Note

Before using Font dialog box one has to remember that font must be installed in a proper manner to the system, Otherwise it can generate error. Or programmer has to check for the error handler. Error can be handle with the use of On error goto statement provided

cancelError property is True for common dialog control.

❑Printer Dialog Box



The CommonDialog control can display two distinct dialogs: the Print Setup dialog box that allows users to select a printer's attributes and the standard Print dialog that lets users select many options of a print job, such as which portion of the document should be printed (all, a page range, or the current selection), the number of copies, and so on. You decide which dialog box appears by setting the cdlPDPrintSetup bit in the Flags property. The complete list of bits that can be set in the Flags property.

```
On Error Resume Next

With CommonDialog1

  ' Prepare to print using the Printer object.

  .PrinterDefault = True

  ' Disable printing to file and individual page printing.

  .Flags = cdlPDDisablePrintToFile Or cdlPDNoPageNums

  If Text1.SelLength = 0 Then

  ' Hide Selection button if there is no selected text.

  .Flags = .Flags Or cdlPDNoSelection

  Else

  ' Else enable the Selection button and make it the default

  ' choice.

  .Flags = .Flags Or cdlPDSelection

  End If

  ' We need to know whether the user decided to print.

  .CancelError = True

  .ShowPrinter
```

```
        If Err = 0 Then
        If .Flags And cdlPDSelection Then
        Printer.Print Text1.SelText
        Else
        Printer.Print Text1.Text
        End If
        End If
    End With
```

Help Windows

You can use the CommonDialog control to display information from HLP files. In this case, no dialog box appears and only a few properties are used. You should assign the HelpFile property the filename and path, and the HelpCommand property an enumerated value that tells what you want to do with that file. Depending on which operation you're performing, you might need to assign a value to either the HelpKey or HelpContext property. The following code snippet shows how you can display the contents page associated with a help file:

```
With CommonDialog1
  ' Note: The path of this file may be different on your system.
  .HelpFile = "C:\Windows\Help\windows.hlp"
  .HelpCommand = cdlHelpContents
  .ShowHelp
End With
```

❐Menu Editor

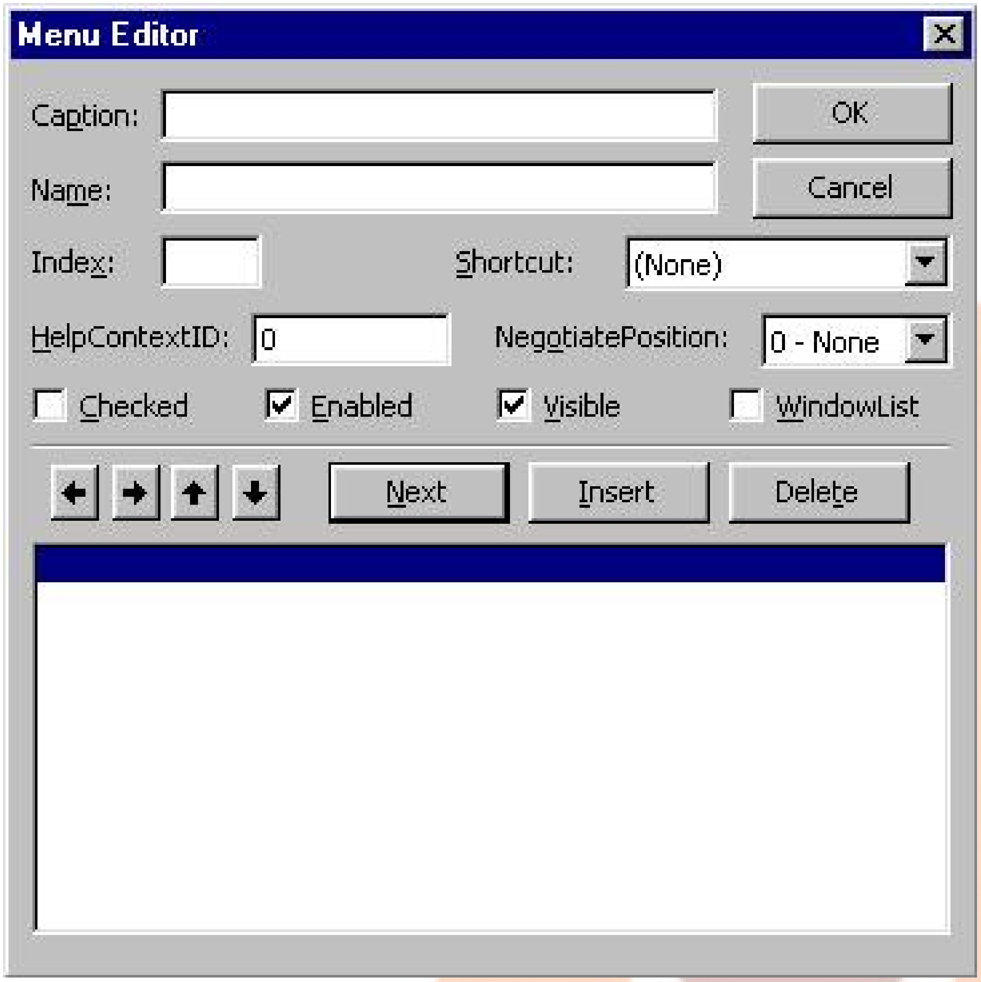


Fig. 1 Shows Menu Editor available in menu : Tools -- Menu Editor (ctrl + E)

Fields	**Details**
Caption	Field value represents menu item in menu
Name	Field value represents menu and useful in programming. It must be unique in list and can not take space in between two names or special character.

Index	To create control array of menu, It helps in generating menus dynamically at run time.
Shortcut	Short cut key related to menu at run time
HelpContextId	Represents Help Id i.e. topic related help available in Help file
NegotiatePosition	Position in menu
Checked	Whether menu is checked or not. can manipulate at run time as <menu.checked> = NOT <menu.checked>
Enabled	Whether menu is Enabled or not. Enabled menu changes to grayed
Visible	Whether menu should be visible to the user or not, can be manipulate at run time
WindowList	Whether window list should be available in menu bar or not. Window list can come with Top level menu and can be assigned to one menu only.
LeftArrowKey	To indent menu item, that generates sub menu
RightArrowKey	To Outdent menu item.
UpArrowKey	To move menu item upwords.
DownArrowKey	To move menu item downwords.

Next	To add next item in menu and to navigate throught menu
Insert	To Insert New item in between existing menu
Delete	To delete existing menu
Ok - Button	To save changes and close menu editor
Cancel - Button	To Undo changes and close menu editor

☐Accessing Menus at Run Time

Menu controls expose only one event, Click. As you expect, this event fires when the user clicks on the menu:

```
Private Sub mnuFileExit_Click()

  Unload Me

End Sub
```

You can manipulate menu items at run time through their Checked, Visible, and Enabled properties. For example, you can easily implement a menu item that acts as a switch and displays or hides a status bar:

```
Private Sub mnuViewStatus_Click()

  ' First, add or remove the check sign.

  mnuViewStatus.Checked = Not mnuViewStatus.Checked

  ' Then make the status bar visible or not.

  staStatusBar.Visible = mnuViewStatus.Checked

End Sub
```

Pop up menu

Visual Basic also supports pop-up menus, those context-sensitive menus that most commercial applications show when you right-click on an user interface object. In Visual Basic, you can display a pop-up menu by calling the form's PopupMenu method, typically from within the MouseDown event procedure of the object:

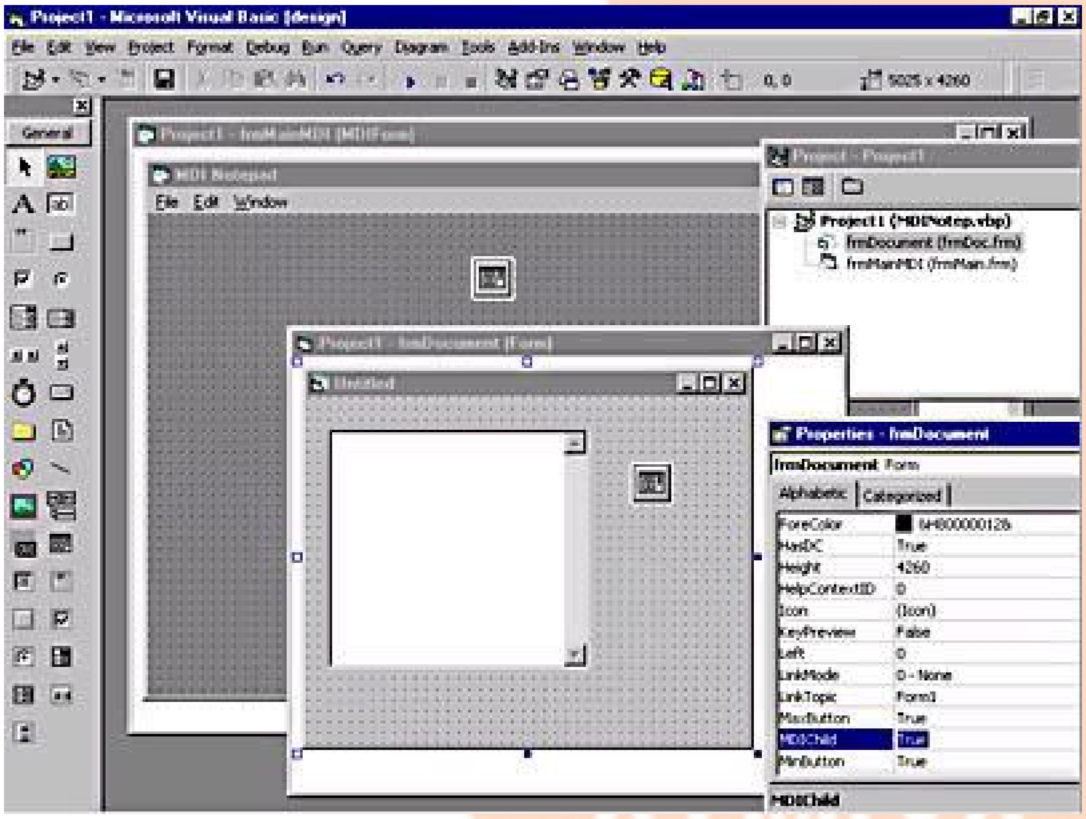
```
Private Sub List1_MouseDown(Button As Integer, Shift As Integer, X As
Single _

  , Y As Single)

        If Button And vbRightButton Then

        ' User right-clicked the list box.

        PopupMenu mnuListPopup

        End If

End Sub
```

The argument you pass to the PopupMenu method is the name of a menu that you have defined using the Menu Editor. This might be either a submenu that you can reach using the regular menu structure or a submenu that's intended to work only as a pop-up menu. In the latter case, you should create it as a top-level menu in the Menu Editor and then set its Visible attribute to False. If your program includes many pop-up menus, you might find it convenient to add one invisible top-level entry and then add all the pop-up menus below it. The complete syntax of the PopupMenu method is quite complex:

PopupMenu Menu, [Flags], [X], [Y], [DefaultMenu]

MDI (Multiple Document Interface)

MDI stands for Multiple Document Interface and is the type of user interface used by most of the applications in the Microsoft Office suite, including Microsoft Word, Microsoft Excel, and Microsoft PowerPoint. Whenever you have an application that should be able to deal with multiple documents at the same time, an MDI interface is probably the best choice.



ForeColor	&H80000012&
HasDC	True
Height	4260
HelpContextID	0
Icon	(Icon)
KeyPreview	False
Left	0
LinkMode	0 - None
LinkTopic	Form1
MaxButton	True
MDIChild	True
MinButton	True

MDIChild

Fig. 1 shows MDI parent and child form with project explorer showing icon difference in the list and MDIChild property as true for child form in properties window.

MDI Applications

You begin developing an MDI application by adding an MDIForm module to the current project. An MDIForm module is similar to a regular Form module, with just a few peculiarities:

- You can have only one MDIForm module in each project; after you add one MDI module to the current project, the Add MDIForm command in the Project menu is disabled, as is the corresponding icon on the main toolbar.

- You can't place most controls directly on an MDIForm surface. More specifically, you can create only menus, invisible controls (such as Timer and CommonDialog controls), and controls that support the Align property (such as PictureBox, Toolbar, and StatusBar controls). The only way to show any other control on an MDIForm object is to place it inside a container control, typically a PictureBox control.

- You can't display text or graphics on an MDIForm surface. Again, you need to place a PictureBox control and display text or graphics inside it.

MDI child forms

An MDIForm object contains one or more child forms. To create such child forms, you add a regular form to the project and set its MDIChild property to True. When you do this, the form's icon in the Project Explorer window changes. You don't have to specify which MDI form this form is a child of because there can be only one MDIForm module per project.

- An MDI child form can't be displayed outside its parent MDIForm. If an MDI child form is the startup form for an application, its parent MDI form is automatically loaded and displayed before the child form becomes visible.

- MDI child forms have other peculiarities as well. For

example, they don't display menu bars as regular forms do: If you add one or more top-level menus to an MDI child form, when the form becomes active its menu bar replaces the MDI parent form's menu bar. For this reason, it's customary for MDI child forms not to include a menu; you define menus only for the main MDIForm module.

- **It's also customary for the Window menu to include a list of all open MDI child forms and to let the user quickly switch to any one of them with a click of the mouse. (See Figure 9-12.) Visual Basic makes it simple to add this feature to your MDI applications: You only have to tick the WindowList option in the Menu Editor for the top-level Window menu. Alternatively, you can create a submenu with the list of all open windows by ticking the WindowList option for a lower level menu item. In any case, only one menu item can have this option ticked.**

CODES

- **When a menu command is invoked in the MDIForm module, you normally apply it to the MDI child form that's currently active, which you do through the ActiveForm property. For example, here's how you execute the Close command on the File menu:**

```
' In the MDI parent form

Private Sub mnuFileClose_Click()

  ' Close the active form, if there is one.

  If Not (ActiveForm Is Nothing) Then Unload ActiveForm

End Sub
```

- **If you wish to open new child of the same type following code can help you out ' Inside the MDIForm module**

```
Private Sub mnuFileNew_Click()

  Dim frmDoc As New frmDocument

  frmDoc.Show

End Sub
```

OR

```
Private Sub mnuFileNew_Click()

       Dim frmDoc As frmDocument

       set frmDoc = New frmDocument

       Load frmDoc

End Sub
```

- **MDIForm modules support an additional method that's not exposed by regular forms: the Arrange method. This method provides a quick way to programmatically arrange all the child forms in an MDI application. You can tile all child forms horizontally or vertically, you can arrange them in a cascading fashion, or you can line up all the minimized forms in an orderly fashion near the bottom of the MDI parent form. To this purpose, you usually create a Window menu with four commands: Tile Horizontally, Tile Vertically, Cascade, and Arrange Icons. This is the code behind these menu items:**

```
Private Sub mnuTileHorizontally_Click()

  Me.Arrange vbTileHorizontal

End Sub

Private Sub mnuTileVertically_Click()

  Me.Arrange vbTileVertical

End Sub

Private Sub mnuCascade_Click()

  Me.Arrange vbCascade

End Sub

Private Sub mnuArrangeIcons_Click()

  Me.Arrange vbArrangeIcons

End Sub
```

□Difference Between MDI and SDI

MDI (Multiple Doc. Interface)	**SDI (Single Doc. Interface)**
Application can be able to deal with multiple documents at the same time.	Application can be able to deal with single documents at the same time.
Microsoft Office member are the example of MDI application.	Notepad is example of SDI application.
We can have only one MDIForm module in each project.	We can have multiple SDI Form in one project.
We can't place most controls directly on an MDIForm surface. For that one require to place picture box and then place controls on picture box.	We can place any or all controls on SDI form.
We can't display text or graphics on an MDIForm surface.	We can use graphical methods on standard form directly.
An MDIForm object contains one or more child forms.	There is no concepts of child form in SDI application
An MDI child form can't be displayed outside its parent MDIForm	Two separate form can be displayed independently.

MDI child forms don't display menu bars. If you add one or more top-level menus to an MDI child form it will show menu to MDI parent form.	SDI forms contains menu items separately on its own.
We can arrange child forms with the use of arrange method from parent form.	We can not arrange forms like MDI and no Arrange method available with SDI.

§ Date

Syntax : **Date()**

Returns : **Variant Data Type**

Details : **It gives system date**

---

§ DateAdd

Syntax : **DateAdd(interval, number, date)**

Returns : **Variant (date)**

Details : **DateAdd function adds specified interval to the given date.**

Interval can be Year (yyyy), Quarter (q), Month (m), Day of year (y), Day (d), Weekday (w), Week (ww), Hour (h), Minute (n), Second (s)

Example : **DateAdd("m", 1, "1-Jan-01")**

Above example adds one month to "1-Jan-01" and returns "1-Feb-01"

---

§ DateDiff

Syntax : **DateDiff(interval, date1, date2)**

Returns : Variant (Long)

Details : **It returns difference between two dates in terms of interval**

Interval can be Year (yyyy), Quarter (q), Month (m), Day of year (y), Day (d), Weekday (w), Week (ww), Hour (h), Minute (n), Second (s)

Example : **DateAdd("m", "1-Mar-01", "1-Jan-01")**

Above example returns 2 as a difference of two dates in terms of month

---

§ DatePart

Syntax : **DatePart(interval, date)**

Returns : Integer

Details : This function returns part of specified date

Example : **This example takes a date and, using the DatePart function, displays the quarter of the year in which it occurs.**

```
Dim TheDate As Date ' Declare variables.

Dim Msg

TheDate = InputBox("Enter a date:")

Msg = "Quarter: " & DatePart("q", TheDate)

MsgBox Msg
```

---

§ DateSerial

Syntax : **DateSerial(year, month, day)**

Returns : **Variant (Date)**

Details : **Returns a Date for a specified year, month, and day.**

Example : **TDate = DateSerial(1975, 6, 26) ' Return a date.**

number represented for year (1975), month (6), and day (26) returns Date with the user of DateSerial.

---

§ **DateValue**

Syntax : **DateValue(date)**

Returns : Date

Details : **Converts a string to a Variant (Date)**

Example : **CDate = DateValue("Jun 26, 1975") ' Return a date to Cdate which is date type variable.**

---

§ **Day**

Syntax : **Day(date)**

Returns : Integer

Details : **specify a whole number between 1 and 31**

---

§ **Hour**

Syntax : **Hour(time)**

Returns : **Integer**

Details : **specifying a whole number between 0 and 23**

---

§ Minute

Syntax : **Minute(time)**

Returns : **Integer**

Details : **specify number between 0 and 59**

---

§ Month

Syntax : **Month(date)**

Returns : **Integer**

Details : **specify number between 1 and 12**

---

§ Second

Syntax : **Second(time)**

Returns : **Integer**

Details : **specifying number between 0 and 59**

---

§ **TimeSerial**

Syntax : **TimeSerial(hour, minute, second)**

Returns : **Date**

Details : **Returns a Time for a specified Hour, Minute and Second.**

Example : **TDate = TimeSerial(16, 35, 17) ' represents 4:35:17 PM.**

---

§ **TimeValue**

Syntax : **TimeValue(time)**

Returns : **Date**

Details : **Returns a Date for a specified year, month, and day.**

Example : **TDate = TimeValue("4:35:17 PM") ' Return a time.**

---

§ **WeekDay**

Syntax : **Weekday(date)**

Returns : **Integer**

Details : Returns Number value coresponding day like sunday as 0, monday as 1 like that...

Example :

Dim TDate, MyWeekDay

TDate = #Jun 26, 1975# ' Assign a date.

MyWeekDay = Weekday(TDate) ' MyWeekDay contains 4 because

' TDate represents a Wednesday.

---

§ **Year**

Syntax : **Year(date)**

Returns : **Integer**

Details : Returns Year part of the date

§ Abs

Syntax : **Abs(number)**

Returns : same dataType as passed

Details : **specify absolute value of a number passed as argument**

Example : Abs(-23.43) = 23.43 and Abs(12) = 12

---

§ Atn

Syntax : **Atn(number)**

Returns : **Double**

Details : Returns arctangent of number

The Atn function takes the ratio of two sides of a right triangle (number) and returns the corresponding angle in radians. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The range of the result is -pi/2 to pi/2 radians. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi. Note Atn is the inverse trigonometric function of Tan,

Example : **pi = 4 * Atn(1) ' Calculate the value of pi.**

---

§ Cos

Syntax : **Cos(number)**

Returns :Double

Details : Returns **cosine of an angle.**

The Cos function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

---

§ **Exp**

Syntax : **Exp(number)**

Returns : **Double**

Details : **specifying e (the base of natural logarithms) raised to a power**

The constant e is approximately 2.718282. Note The Exp function complements the action of the Log function and is sometimes referred to as the antilogarithm.

---

§ **Log**

Syntax : **Log(number)**

Returns : **Double**

Details : **Returns natural logarithm of a number**

The natural logarithm is the logarithm to the base e. You can

calculate base-n logarithms for any number x by dividing the natural logarithm of x by the natural logarithm of n as follows:

Logn(x) = Log(x) / Log(n)

Example : **The following example illustrates a custom Function that calculates base-10 logarithms:**

```
Static Function Log10(X)

  Log10 = Log(X) / Log(10#)

End Function
```

---

§ **Rnd**

Syntax : **Rnd(number)**

Returns : **Single**

Details : Generates random number in specified number range

Example : Rnd(255) returns any value between o to 255

---

§ **Sgn**

Syntax : **Sgn(number)**

Returns : **Integer**

Details : Returns **the sign of a number**

Example : Sgn(-234.53) = -1 , Sgn(43) = + 1

---

§ **Sin**

Syntax : **Sin(number)**

Returns : **Double**

Details : **sine of an angle**

The Sin function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.The result lies in the range -1 to 1.

---

§ **Sqr**

Syntax : **Sqr(number)**

Returns : **Double**

Details : **Returns square root of a number**

Example : Sqr(25) = 5, Sqr(1) = 1

---

§ **Tan**

Syntax : **Tan(number)**

Returns : **Double**

Details : **Returns tangent of an angle**

Tan takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

§ Asc

Syntax : **Asc(string)**

Returns : Integer

Details : **Returns an Integer representing the character code corresponding to the first letter in a string.**

Example : Asc("a") = 97, Asc("ab") = 97, Asc("b") = 98

---

§ Chr

Syntax : **Chr(charcode)**

Returns : String

Details : **Returns a String containing the character associated with the specified character code.**

Note The ChrB function is used with byte data contained in a String. Instead of returning a character, which may be one or two bytes, ChrB always returns a single byte. The ChrW function returns a String containing the Unicode character except on platforms where Unicode is not supported, in which case, the behavior is identical to the Chr function.

Example : Chr(97) = "a" , Chr(98) = "b", **Chr(65) = A, Chr(62) = ">", Chr(37) = "%"**

---

§ Format

Syntax : **Format(expression, format)**

Returns : **String**

Details : Returns formatted expression **according to instructions contained in a format expression.**

Example :

Dim MyTime, MyDate, MyStr

MyTime = #17:04:23#

MyDate = #January 27, 1993#

' Returns current system time in the system-defined long time format.

MyStr = Format(Time, "Long Time")

' Returns current system date in the system-defined long date format.

MyStr = Format(Date, "Long Date")

MyStr = Format(MyTime, "h:m:s") ' Returns "17:4:23".

MyStr = Format(MyTime, "hh:mm:ss AMPM") ' Returns "05:04:23 PM".

MyStr = Format(MyDate, "dddd, mmm d yyyy") ' Returns "Wednesday,

' Jan 27 1993".

' If format is not supplied, a string is returned.

MyStr = Format(23) ' Returns "23".

' User-defined formats.

MyStr = Format(5459.4, "##,##0.00") ' Returns "5,459.40".

MyStr = Format(334.9, "###0.00") ' Returns "334.90".

MyStr = Format(5, "0.00%") ' Returns "500.00%".

MyStr = Format("HELLO", "<") ' Returns "hello".

MyStr = Format("This is it", ">") ' Returns "THIS IS IT".

---

§ **Instr**

Syntax : **InStr([start,]string1, string2[, compare])**

Returns : **Long**

Details : **specify the position of the first occurrence of one string within another.**

[start] - an optional argument which specifies string position from

where search has to start.

string1 - compulsary argument

string2 - compulsary argument

[compare] - an optional argument which specifies comparision type like vbBinaryCompare, vbTextCompare etc.

Return Values

If	InStr returns
string1 is zero-length	0
string1 is Null	Null
string2 is zero-length	start
string2 is Null	Null
string2 is not found	0
string2 is found within string1	Position at which match is found
start > string2	0

The InStrB function is used with byte data contained in a string. Instead of returning the character position of the first occurrence of one string within another, InStrB returns the byte position.

Example :

Dim SearchString, SearchChar, MyPos

SearchString ="XXpXXpXXPXXP" ' String to search in.

SearchChar = "P" ' Search for "P".

' A textual comparison starting at position 4. Returns 6.

MyPos = Instr(4, SearchString, SearchChar, 1)

' A binary comparison starting at position 1. Returns 9.

MyPos = Instr(1, SearchString, SearchChar, 0)

' Comparison is binary by default (last argument is omitted).

MyPos = Instr(SearchString, SearchChar) ' Returns 9.

MyPos = Instr(1, SearchString, "W") ' Returns 0.

---

§ **Lcase**

Syntax : **LCase(string)**

Returns : String

Details : **Returns a String that has been converted to lowercase.**

Example : Lcase("eBookMark") = "ebookmark"

---

§ Left

Syntax : **Left(string, length)**

Returns : **String**

Details : **Returns String containing a specified number of characters from the left side of a string.**

Example : **Left("eBookMark",5) = "eBook"**

---

§ Len

Syntax : **Len(string | varname)**

Returns : Long

Details : **Returns number of characters in a string or the number of bytes required to store a variable.**

Example : Len("eBookMark") = 9 ' returns no of character in string

Dim i as Single

len(i) = 4 ' returns no of bytes required for i to store any value

---

§ Trim

Syntax : **LTrim(string), RTrim(string), Trim(string)**

Returns : **String**

Details : **Returns a String containing a copy of a specified string without leading spaces (LTrim), trailing spaces (RTrim), or both leading and trailing spaces (Trim).**

Example :

Ltrim(" eBookMark") = "eBookMark"

Rtrim("eBookMark ") = "eBookMark"

Trim(" eBookMark ") = "eBookMark"

---

§ **Mid**

Syntax : **Mid(string, start[, length])**

Returns : String

Details : **Returns String containing a specified number of characters from a string. Note Use the MidB function with byte data contained in a string, as in double-byte character set languages. Instead of specifying the number of characters, the arguments specify numbers of bytes. For sample code that uses MidB, see the second example in the example topic.**

Example :

Mid("eBookMark", 2, 4) = "Book"

Mid("eBookMark", 6, 1) = "M"

---

§ MonthName

Syntax : **MonthName(month[, abbreviate])**

Returns : String

Details : **Returns a string indicating the specified month.**

Example :

MonthName(1) = "January"

MonthName(1,0) = "January"

MonthName(1,1) = "Jan" 'abbreviated

MonthName(2,1) = "Feb"

---

§ Right

Syntax : **Right(string, length)**

Returns : String

Details : **Returns a String containing a specified number of characters from the right side of a string.**

Example :

Right("eBookMark", 4) = "Mark"

Right("eBookMark", 1) = "k"

- ## Space

Syntax : **Space(number)**

Returns : **String**

Details : **Returns String consisting of the specified number of spaces. It is useful for representing data in structured manner.**

Example :

"eBookMark" & Space(10) & "is nice" = "eBookMark is nice"

- ## StrComp

Syntax : **StrComp(string1, string2[, compare])**

Returns : **Integer**

Details : **Returns a Integer indicating the result of a string comparison.**

Compare is a optional argument that can hold value like vbBinaryCompare (0) that Performs a binary comparison and vbTextCompare (1) that Performs a textual comparison.

Return Values

If	**StrComp returns**
string1 is less than string2	-1
string1 is equal to string2	0
string1 is greater than string2	1
string1 or string2 is null	null

Example :

MyStr1 = "ABCD": MyStr2 = "abcd" ' Define variables.

MyComp = StrComp(MyStr1, MyStr2, 1) ' Returns 0.

MyComp = StrComp(MyStr1, MyStr2, 0) ' Returns -1.

MyComp = StrComp(MyStr2, MyStr1) ' Returns 1.

---

§ **StrConv**

Syntax : **StrConv(string, conversion, LCID)**

Returns : String

Details : **Returns String converted as specified. LCID (Optional) The LocaleID, if different than the system LocaleID. (The system LocaleID is the default.)**

The conversion argument settings are:

Constant	Value	Description
vbUpperCase	1	Converts the string to uppercase characters.
vbLowerCase	2	Converts the string to lowercase characters.
vbProperCase	3	Converts the first letter of every word in string to uppercase.
vbUnicode	64	Converts the string to Unicode using the default code page of the system.
vbFromUnicode	128	Converts the string from Unicode to the default code page of the system.

The following are valid word separators for proper casing: Null (Chr$(0)), horizontal tab (Chr$(9)), linefeed (Chr$(10)), vertical tab (Chr$(11)), form feed (Chr$(12)), carriage return (Chr$(13)), space (SBCS) (Chr$(32)).

Example :

This example uses the StrConv function to convert a Unicode string to an ANSI string.

```
Dim i As Long

Dim x() As Byte

x = StrConv("ABCDEFG", vbFromUnicode) ' Convert string.

For i = 0 To UBound(x)
```

```
    Debug.Print x(i)

  Next
```

---

§ **String**

Syntax : **String(number, character)**

Returns : String

Details : **Returns String containing a repeating character string of the length specified.**

Example :

String(5, "*") ' Returns "***".**

String(10, "ABC") ' Returns "AAAAAAAAAA".

---

§ **StrReverse**

Syntax : **StrReverse(expression)**

Returns : String

Details : **Returns a string in which the character order of a specified string is reversed.**

---

§ **Ucase**

Syntax : **UCase(string)**

Returns : String

Details : **Returns String containing the specified string, converted to uppercase.**

Example : Ucase("eBookMark") = "EBOOKMARK"

---

§ **WeekDayName**

Syntax : **WeekDayName(weekday, [abbreviate])**

Returns : String

Details : **Returns a string that indicates the day of the week.**

Example :

WeekdayName(2) = "Monday"

WeekdayName(2, 0) = "Monday"

WeekdayName(2, 1) = "Mon"

§ Err

Syntax : Err()

Details : **Function Err() As ErrObject Member of VBA.Information Returns the error number for the error that occurred**

Example : Err.Number

---

§ IsArray

Syntax : **IsArray(VarName)**

Returns : Boolean

Details : **Function IsArray(VarName) As Boolean Member of VBA.Information Returns True if the variable is an array**

Example :

```
Dim i(10) as integer, j as boolean, k as single

j = IsArray(i) ' value of j = True

k = IsArray(k) 'value of k = False
```

---

§ Isdate

Syntax : **IsDate(expression)**

Returns : **Boolean**

Details : **Returns a Boolean value indicating whether an expression can be converted to a date.**

Example :

Dim MyDate, YourDate, NoDate, MyCheck

MyDate = "February 12, 1969": YourDate = #2/12/69#: NoDate = "Hello"

MyCheck = IsDate(MyDate) ' Returns True.

MyCheck = IsDate(YourDate) ' Returns True.

MyCheck = IsDate(NoDate) ' Returns False

---

§ **IsEmpty**

Syntax : **IsEmpty(expression)**

Returns : Boolean

Details : **Returns a Boolean value indicating whether a variable has been initialized. IsEmpty returns True if the variable is uninitialized, or is explicitly set to Empty; otherwise, it returns False. False is always returned if expression contains more than one variable. IsEmpty only returns meaningful information for variants.**

Example :

Dim MyVar, MyCheck

MyCheck = IsEmpty(MyVar) ' Returns True.

MyVar = Null ' Assign Null.

MyCheck = IsEmpty(MyVar) ' Returns False.

MyVar = Empty ' Assign Empty.

MyCheck = IsEmpty(MyVar) ' Returns True.

---

§ **IsError**

Syntax : IsError(expression)

Returns : Boolean

Details : **Returns a Boolean value indicating whether an expression is an error value. The IsError function is used to determine if a numeric expression represents an error. IsError returns True if the expression argument indicates an error; otherwise, it returns False.**

---

§ **IsMissing**

Syntax : IsMissing(argname)

Returns : Boolean

Details : **Returns a Boolean value indicating whether an optional Variant argument has been passed to a procedure. The required argname argument contains the name of an optional Variant**

procedure argument. Use the IsMissing function to detect whether or not optional Variant arguments have been provided in calling a procedure. IsMissing returns True if no value has been passed for the specified argument; otherwise, it returns False. Note IsMissing does not work on simple data types (such as Integer or Double) because, unlike Variants, they don't have a provision for a "missing" flag bit.

Example :

```
Dim ReturnValue

' The following statements call the user-defined function
procedure.

ReturnValue = ReturnTwice() ' Returns Null.

ReturnValue = ReturnTwice(2) ' Returns 4.



' Function procedure definition.

Function ReturnTwice(Optional A)

  If IsMissing(A) Then

  ' If argument is missing, return a Null.

  ReturnTwice = Null

  Else

  ' If argument is present, return twice the value.
```

```
    ReturnTwice = A * 2

    End If

  End Function
```

---

§ **IsNull**

Syntax : **IsNull(expression)**

Returns : **Boolean**

Details : **Returns a Boolean value that indicates whether an expression contains no valid data (Null). The Null value indicates that the Variant contains no valid data. Null is not the same as Empty, which indicates that a variable has not yet been initialized. It is also not the same as a zero-length string (""), which is sometimes referred to as a null string.**

Example :

```
Dim MyVar, MyCheck

MyCheck = IsNull(MyVar) ' Returns False.



MyVar = ""

MyCheck = IsNull(MyVar) ' Returns False.



MyVar = Null
```

MyCheck = IsNull(MyVar) ' Returns True.

---

§ **IsNumeric**

Syntax : IsNumeric(expression)

Returns : Boolean

Details : Returns a Boolean value indicating whether an expression can be evaluated as a number. IsNumeric returns True if the entire expression is recognized as a number; otherwise, it returns False. IsNumeric returns False if expression is a date expression.

Example :

Dim MyVar, MyCheck

MyVar = "53" ' Assign value.

MyCheck = IsNumeric(MyVar) ' Returns True.

MyVar = "459.95" ' Assign value.

MyCheck = IsNumeric(MyVar) ' Returns True.

MyVar = "45 Help" ' Assign value.

MyCheck = IsNumeric(MyVar) ' Returns False.

---

§ IsObject

Syntax : IsObject(identifier)

Returns : Boolean

Details : **Returns a Boolean value indicating whether an identifier represents an object variable. IsObject is useful only in determining whether a Variant is of VarType vbObject. This could occur if the Variant actually references (or once referenced) an object, or if it contains Nothing. IsObject returns True if identifier is a variable declared with Object type or any valid class type, or if identifier is a Variant of VarType vbObject, or a user-defined object; otherwise, it returns False. IsObject returns True even if the variable has been set to Nothing.**

Example :

```
Dim MyInt As Integer, YourObject, MyCheck ' Declare
variables.

Dim MyObject As Object

Set YourObject = MyObject ' Assign an object reference.

MyCheck = IsObject(YourObject) ' Returns True.

MyCheck = IsObject(MyInt) ' Returns False.
```

---

§ QBColor

Syntax : **QBColor(color)**

Returns : Long

Details : **Returns a Long representing the RGB color code corresponding to the specified color number in the range 0–15.**

Number	**Color**
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Yellow
7	White
8	Gray
9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red

13	Light Magenta
14	Light Yellow
15	Bright White

Example :

This example shows that form background color get change every second for 16 colors as QbColor(0) to QbColor(15)

```
Private Sub Timer1_Click()

        Static i as Integer

        '---interval of timer is set to 1000

        Form1.BackColor = QbColor(i)

        i = i + 1

        If i = 16 then

                i=0

        end if

End Sub
```

---

§ **RGB**

Syntax : **RGB(red, green, blue)**

Returns : Long

Details : **Returns a Long whole number representing an RGB color value. Red, Green, Blue are Required arguments. It is number in the range of 0–255. The value for any argument to RGB that exceeds 255 is assumed to be 255.**

Example :

RGB(0, 0, 0) = Black

RGB(0, 0, 255) = Blue

RGB(0, 255, 0) = Green

RGB(255, 0, 0) = Red

---

§ **TypeName**

Syntax : **TypeName(varname)**

Returns : **String**

Details : **Returns a String that provides information about a variable.**

Example :

' Declare variables.

Dim NullVar, MyType, StrVar As String, IntVar As Integer, CurVar As Currency

```
Dim ArrayVar (1 To 5) As Integer

NullVar = Null ' Assign Null value.

MyType = TypeName(StrVar) ' Returns "String".

MyType = TypeName(IntVar) ' Returns "Integer".

MyType = TypeName(CurVar) ' Returns "Currency".

MyType = TypeName(NullVar) ' Returns "Null".

MyType = TypeName(ArrayVar) ' Returns "Integer()".
```

Collection Vs. Array

Collections are objects exposed by the VBA library. They can be used in Visual Basic applications to store groups of related data. In this sense, Collections are similar to arrays, but the similarities stop here because of these substantial differences:

- Collection objects don't need to be predimensioned for a given number of elements; you can add items to a Collection, and it will grow as needed.

- You can insert items in the middle of a Collection without worrying about making room for the new element; likewise, you can delete items without having to shift all other items to fill the hole. In both cases, the Collection object takes care of all these chores automatically.

- You can store nonhomogeneous data in a Collection, whereas arrays can host only data of the type set at compile time (with the exception of Variant arrays). In general, you can store in a Collection any value that you could store in a Variant variable (that is, everything except fixed-length strings and possibly UDTs).

- A Collection offers a way to associate a key with each item so that later you can quickly retrieve that item even if you don't know where it's stored in the Collection. You can also read items by their numerical index in the collection, as you would do with regular arrays.

- In contrast to the situation for arrays, once you have added an item to a Collection you can read the item but not modify it. The only way to modify a value in a Collection is to delete the old value and add the new one.

Collection in Detail

With all these advantages, you might wonder why collections haven't supplanted arrays in the hearts of Visual Basic developers. The main reason is that Collections are slow, or at least they're noticeably slower than arrays. To give you an idea, filling an array of 10,000 Long elements is about 100 times faster than filling a Collection of the same size. Take this into account when you're deciding which data structure best solves your problem. The first thing you must do before using a Collection is create it. Like all objects, a Collection should be declared and then created, as in the following code:

```
Dim EmployeeNames As Collection

Set EmployeeNames = New Collection
```

Or you can declare an auto-instancing collection with one single line of code:

```
Dim EmployeeNames As New Collection
```

- **ADD Method**

You can add items to a Collection object by using its Add method; this method expects the value you're adding and a string key that will be associated with that value:

EmployeeNames.Add "John Smith", "Marketing"

where value can be virtually anything that can be stored in a Variant. The Add method usually appends the new value to the collection, but you can decide where exactly you want to store it using either the before argument or the after argument:

```
' Insert this value before the first item in the collection.

EmployeeNames.Add "Anne Lipton", "Sales"

' Insert this new value after the element added previously.

EmployeeNames.Add value2, "Robert Douglas", ,"Sales"
```

Unless you have a good reason to store the new value somewhere other than at the end of the Collection, I suggest that you not use the before or after arguments because they slow down the Add method. The string key is optional. If you specify it and there's another item with the same key, the Add method will raise an error 457-"This key is already associated with an element of this collection." (Keys are compared in a case-insensitive way.)

Once you have added one or more values, you can retrieve them using the Item method; this method is the default member of the Collection class, so you can omit it if you want. Items can be read using their numeric indices (as you do with arrays) or their string keys:

```
' All the following statements print "Anne Lipton".

Print EmployeeNames.Item("Sales")

Print EmployeeNames.Item(1)

Print EmployeeNames("Sales")
```

Print EmployeeNames(1)

- **Count method**

returns the number of items in the collection:

EmployeeNames.Count

' Retrieve the last item in the EmployeeNames collection.

Print EmployeeNames.Item(EmployeeNames.Count)

- **Remove Item**

You can remove items from a Collection object using the Remove method; this method accepts either a numeric index or a string key:

' Remove the Marketing Boss.

EmployeeNames.Remove "Marketing"

If the key doesn't exist, the Collection object raises an error 5- "Invalid procedure call or argument." Collections don't offer a native way to remove all the items in a single operation, so you're forced to write a loop. Here's a general function that does it for you:

```
Sub RemoveAllItems(col As Collection)

  Do While col.Count
  col.Remove 1
  Loop
End Sub
```

A faster way to remove all the items in a Collection is to destroy the Collection object itself by setting it to Nothing or to another fresh, new instance:

```
' Both these lines destroy the current contents
' of the Collection.
Set EmployeeNames = Nothing
Set EmployeeNames = New Collection
```

Note

Finally, as I mentioned before, Collections don't allow you to modify the value of an item. If you want to change the value of an item, you must first delete it and then add a new item.

§ CBool

Syntax : **CBool(expression)**

Returns : **Boolean.**

Details : **If expression is zero, False is returned; otherwise, True is returned.**

Example :

```
Dim A, B, Check

A = 5: B = 5 ' Initialize variables.

Check = CBool(A = B) ' Check contains True.

A = 0 ' Define variable.

Check = CBool(A) ' Check contains False.
```

---

§ CByte

Syntax : **CByte(expression)**

Returns : **Byte**

Details : **CByte(expression) returns byte value for given expression**

Example :

```
Dim MyDouble, MyByte

MyDouble = 125.5678 ' MyDouble is a Double.
```

MyByte = CByte(MyDouble) ' MyByte contains 126.

---

§ **CCur**

Syntax : **CCur(expression)**

Returns : **Currency**

Details : **CCur(expression) function returns Currency datatype value for given expression**

Example :

```
Dim MyDouble, MyCurr

MyDouble = 543.214588 ' MyDouble is a Double.

MyCurr = CCur(MyDouble * 2)

' Convert result of MyDouble * 2 (1086.429176) ' to a
Currency (1086.4292).
```

---

§ **CDate**

Syntax : **CDate(date)**

Returns : **Date**

Details : **The following example uses the CDate function to convert a string to a date. In general, hard coding dates and times as strings (as shown in this example) is not recommended. Use date and time**

literals (such as #10/19/1962#, #4:45:23 PM#) instead.

Example :

> **MyDate = "October 19, 1962"&#9;' Define date.**
>
> **MyShortDate = CDate(MyDate)&#9;' Convert to Date data type.**
>
> **MyTime = "4:35:47 PM"&#9;&#9;' Define time.**
>
> **MyShortTime = CDate(MyTime)&#9;' Convert to Date data type.**

---

§ **CDbl**

Syntax : **CDbl(expression)**

Returns : **Double**

Details : **Use the CDbl function to provide internationally aware conversions from any other data type to a Double subtype.**

Example :

> **Dim MyCurr, MyDouble**
>
> **MyCurr = CCur(234.456784) ' MyCurr is a Currency (234.4567).**
>
> **MyDouble = CDbl(MyCurr * 8.2 * 0.01)**
>
> **' Convert result to a Double (19.2254576).**

---

§ CInt

Syntax : **CInt(expression)**

Returns : **Integer**

Details : **Use the CInt function to provide internationally aware conversions from any other data type to an Integer subtype.**

Example :

```
Dim MyDouble, MyInt

MyDouble = 2345.5678 ' MyDouble is a Double.

MyInt = CInt(MyDouble) ' MyInt contains 2346.
```

---

§ Same way other function workes, Add to Conversion functions for

CLng	Long
CSng	Single
CStr	String

---

§ Hex

Syntax : **Hex(number)**

Returns : **hexadecimal**

Details : **Returns a string representing the hexadecimal value of a number.**

Example :

```
Dim MyHex

MyHex = Hex(5) ' Returns 5.

MyHex = Hex(10) ' Returns A.

MyHex = Hex(459) ' Returns 1CB.
```

---

§ **Oct**

Syntax : **Oct(number)**

Returns : **octal**

Details : **Returns a string representing the octal value of a number.**

Example :

```
Dim MyOct

MyOct = Oct(4) ' Returns 4.

MyOct = Oct(8) ' Returns 10.

MyOct = Oct(459) ' Returns 713.
```

---

§ Val

Syntax : **Val(string)**

Returns : Double

Details : **Returns the numbers contained in a string as a numeric value of appropriate type.**

Example :

```
Dim MyValue

MyValue = Val("2457") ' Returns 2457.

MyValue = Val(" 2 45 7") ' Returns 2457.

MyValue = Val("24 and 57") ' Returns 24.
```

Graphics with VB

Graphics Category:

1. **Bitmap:** Bitmap Graphics are images that can be displayed on various controls and processed on a pixel-by-pixel basis. You will be able to khow more about graphics controls like form, picturebox, and imagebox in chapter 3 and 4.

2. **Vector:** Vector Graphics are images generated by graphics commands such as the line and circle commands.

Difference

Bitmap	Vector
Bitmap graphics gets distortion when resolutions gets low.	Vector Graphics are not tied to a specific monitor resilution. Means that can be displayed at various resolutions.
We can create images of landscape in bitmap graphics.	We can't create images of landscape in vector graphics.
Bitmap graphics can be created using visual basic tools like shape, line with the use of form, picturebox and imagebox.	Vector graphics can be generated on picturebox, form with the use of Line, circle and many more graphic functions.

Coordinate System (ScaleMode property)

Constant Name	Value	Description
vbUser	0	User-defined coordinate system
vbTwips	1	Twips (1,440 twips per inch) **Default**
vbPoints	2	Points (72 points per inch)
vbPixels	3	Pixels
vbCHaracters	4	Characters (120 twips wide, 240 twips high)
vbInches	5	Inches
vbMillimeters	6	Millimeters
vbCentimeters	7	Centimeters

VECTOR Methods

- **PSet Method:**

To set a single point in a graphic object (form or picture box) to a particular color, use the PSet method. We usually do this to designate a starting point for other graphics methods. The syntax is:

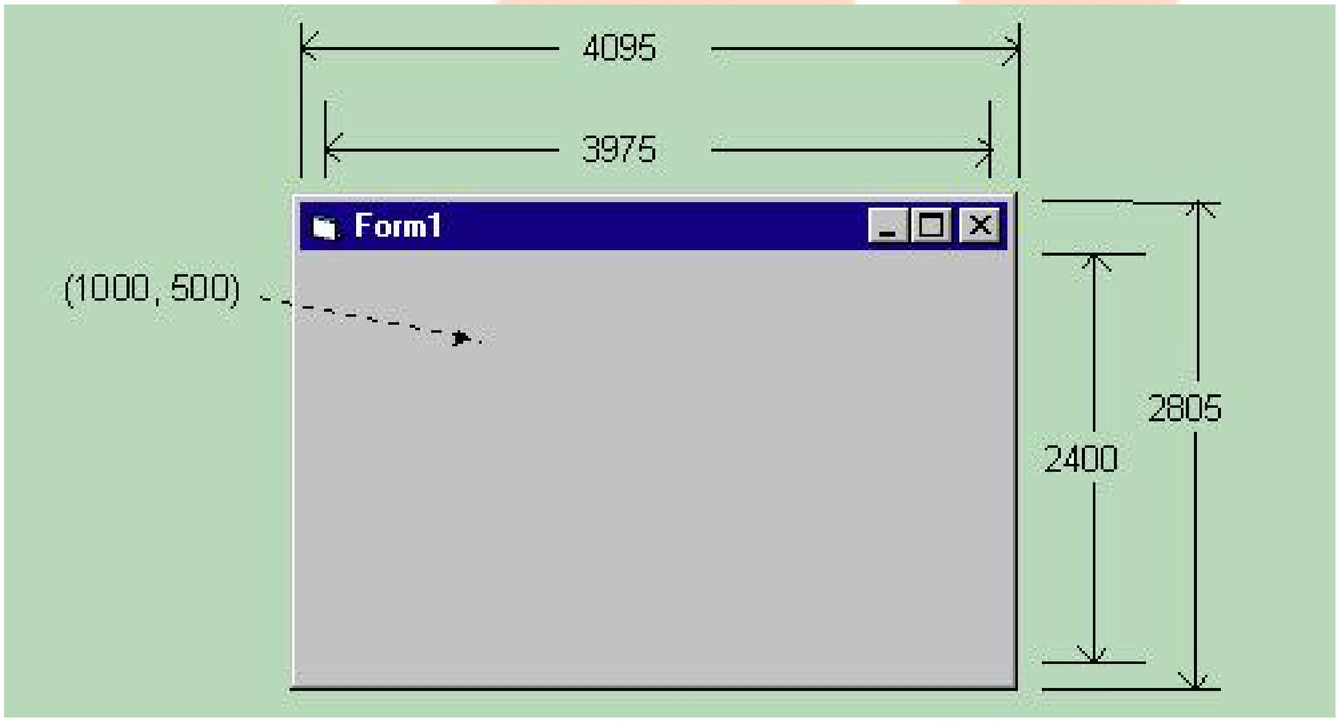
ObjectName.PSet (x, y), Color

where ObjectName is the object name, (x, y) is the selected point, and Color is the point color. If the ObjectName is

omitted, the current form is assumed to be the object. If Color is omitted, the object's ForeColor property establishes the color. PSet is usually used to initialize some further drawing process. Pset can be viewed only when objects AutoRedraw property is True. Size of point can be incresed with the use of DrawWidth property.

This form has a ScaleWidth of 3975 (Width 4095) and a ScaleHeight of 2400 (Height 2805). The command:

PSet (1000, 500)



- **Line Method:**

The Line method is very versatile. We can use it to draw line segments, boxes, and filled boxes. To draw a line, the syntax is:

ObjectName.Line (x1, y1) - (x2, y2), Color

where ObjectName is the object name, (x1, y1) the starting coordinate, (x2, y2) the ending coordinate, and Color the line color. Like PSet, if ObjectName is omitted, drawing is done to the current form and, if Color is omitted, the object's ForeColor property is used.

To draw a line from (CurrentX, CurrentY) to (x2, y2), use:

```
ObjectName.Line - (x2, y2), Color
```

To draw a box bounded by opposite corners (x1, y1) and (x2, y2), use:

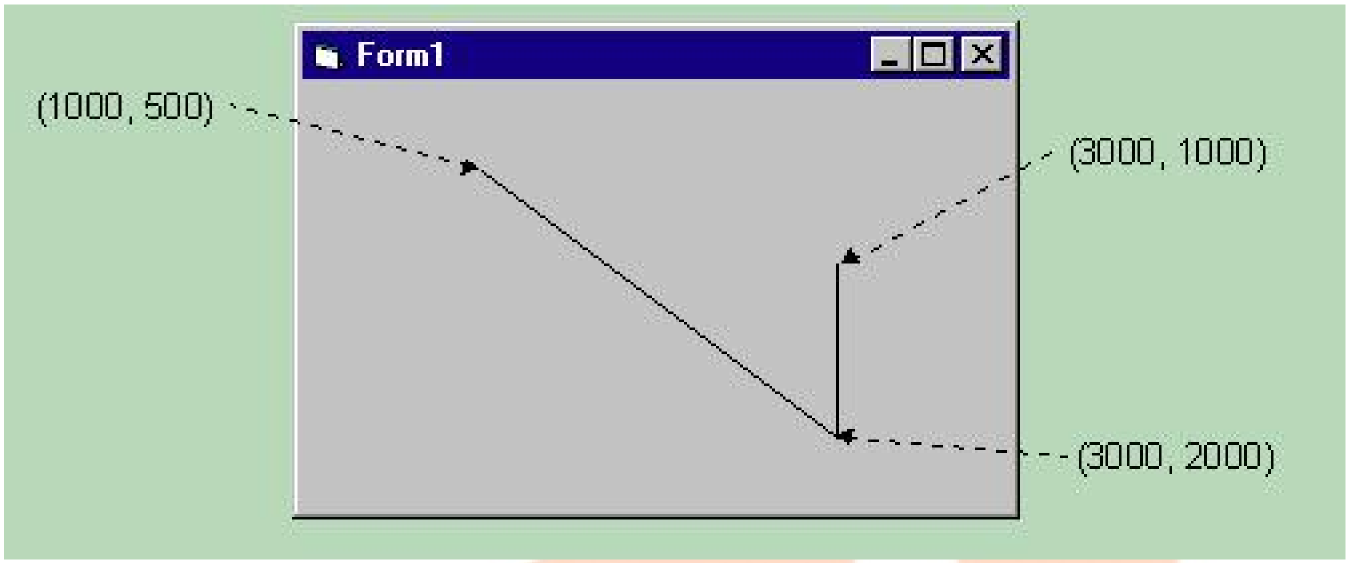
```
ObjectName.Line (x1, y1) - (x2, y2), Color, B
```

and to fill that box (using the current FillPattern), use:

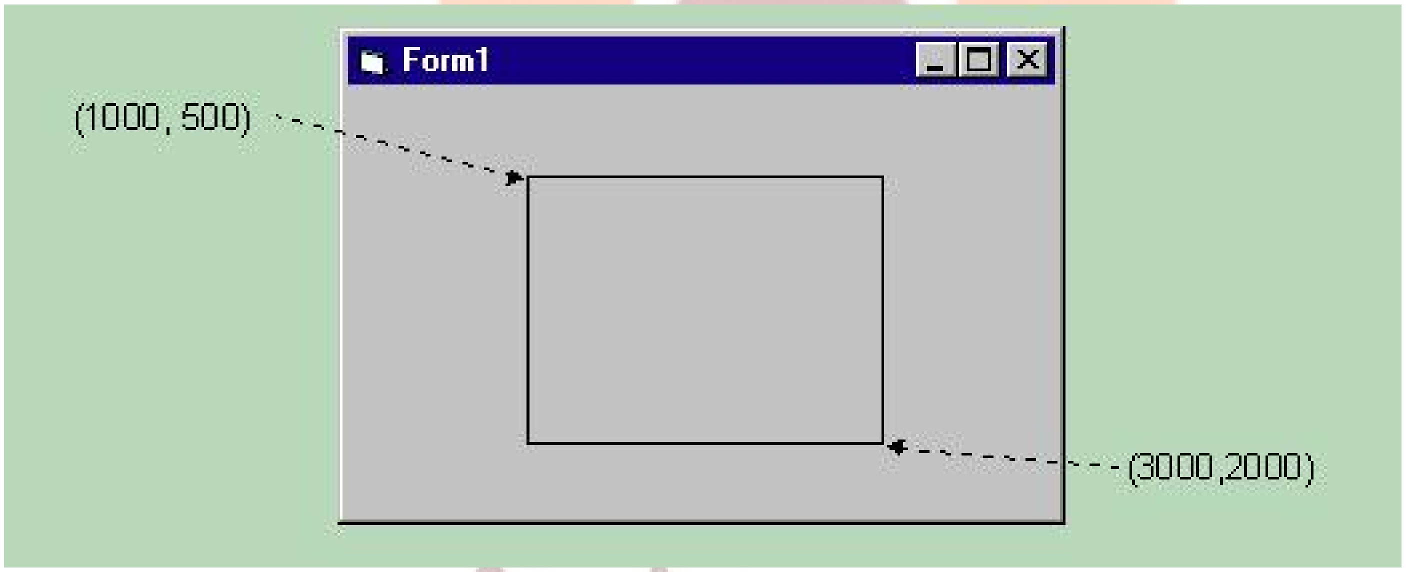
```
ObjectName.Line (x1, y1) - (x2, y2), Color, BF
```

Line Method Examples:

```
Line (1000, 500) - (3000, 2000)

Line - (3000, 1000)
```



Line (1000, 500) - (3000, 2000), , B



□DrawStyle property

Constant Name	Value	Description
vbSolid	0	(Default) Solid
vbDash	1	Dash

vbDot	2	Dot
vbDashDot	3	Dash-dot
vbDashDotDot	4	Dash-dot-dot
vbInvisible	5	Transparent
vbInsideSolid	6	Inside solid

FillStyle property

vbFSSolid	0	Solid
vbFSTransparent	1	(Default) Transparent
vbHorizontalLine	2	Horizontal Line
vbVerticalLine	3	Vertical Line
vbUpwardDiagonal	4	Upward Diagonal
vbDownwardDiagonal	5	Downward Diagonal
vbCross	6	Cross
vbDiagonalCross	7	Diagonal Cross

- **Circle Method:**

The Circle method can be used to draw circles, ellipses, arcs, and pie slices. We'll only look at drawing circles. The syntax is:

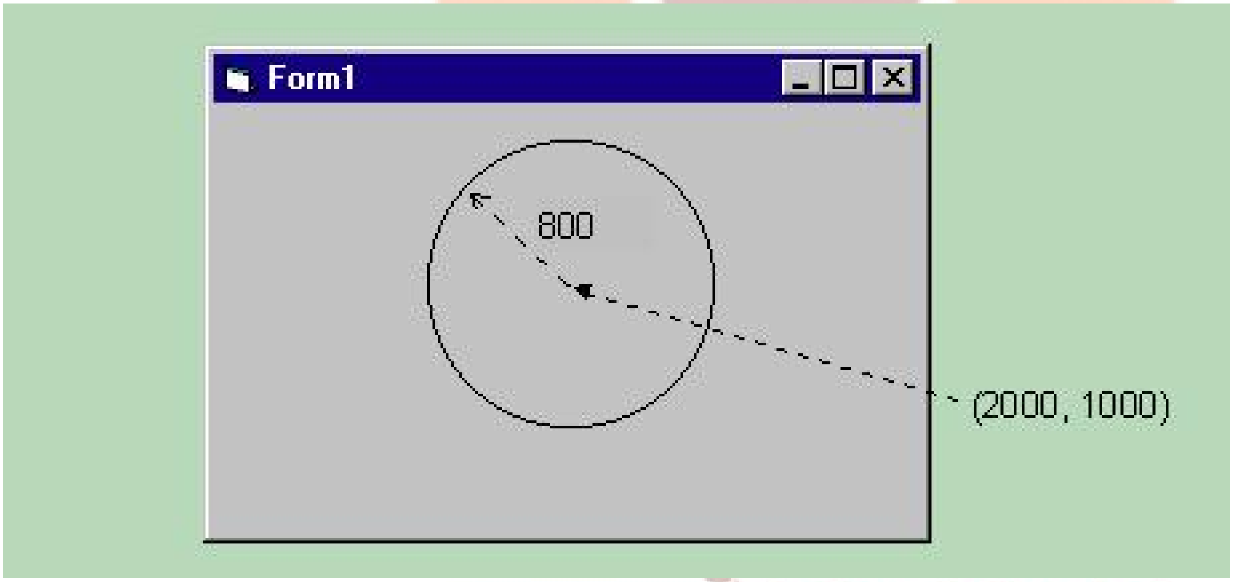
ObjectName.Circle (x, y), r, Color

This command will draw a circle with center (x, y) and radius r, using Color.

Circle Example:

With the same example form, the command:

Circle (2000, 1000), 800



- **Print Method:**

Another method used to 'draw' to a form or picture box is the Print method. Yes, for these objects, printed text is drawn to the form. The syntax is:

ObjectName.Print [information to print]

Here the printed information can be variables, text, or some combination. If no object name is provided, printing is to the current form. Information will print beginning at the object's CurrentX and CurrentY value. The color used is specified by the object's ForeColor property and the font is specified by the object's Font characteristics.

- **Cls Method:**

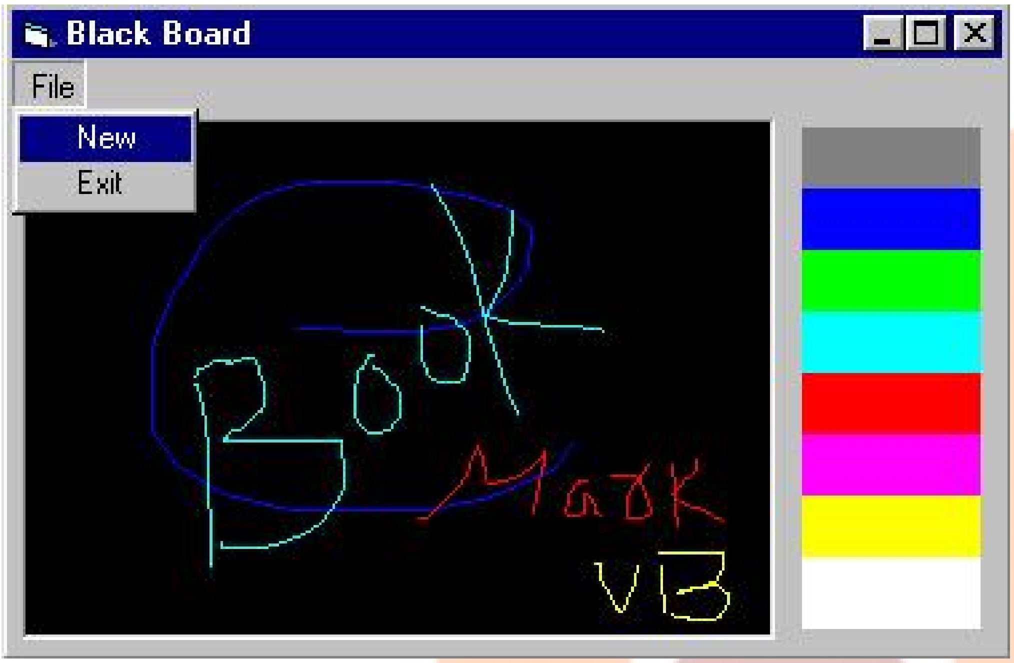
To clear the graphics drawn to an object, use the Cls method. The syntax is:

ObjectName.Cls

If no object name is given, the current form is cleared. Recall Cls only clears the lowest of the three display layers. This is where graphics methods draw.

Example (Black Board)

Design



Table

Sr No	Name	Control Type	Properties

1	frmDraw	Form	Caption="Black Board"
			
			BorderStyle=1-Fixed Single

2	picDraw	Picture Box	Name=picDraw
3	lblColor(0 to 7)	Label	Index=0 to 7
4	mnuFile	Menu	Name = mnuFile
5	mnuFileNew	Menu	Name = mnuNewFile
6	mnuFileExit	Menu	Name = mnuFileExit

Events

'DrawOn will be used to indicate whether you are drawing or not.

```
Option Explicit

Dim DrawOn As Boolean
```

'The Form_Load procedure loads colors into each of the label boxes to allow choice of drawing color. It also sets the BackColor to black and the ForeColor to Bright White.

```
Private Sub Form_Load()

    'Load drawing colors into control array

    Dim I As Integer

    For I = 0 To 7

    lblColor(I).BackColor = QBColor(I + 8)

    Next I

    picDraw.ForeColor = QBColor(15) ' Bright White

    picDraw.BackColor = QBColor(0) ' Black

End Sub
```

In the mnuFileNew_Click procedure, we check to see if the user really wants to start over. If so, the picture box is cleared with the Cls method.

```
Private Sub mnuFileNew_Click()
```

```
    'Make sure user wants to start over

    Dim Response As Integer

    Response = MsgBox("Are you sure you want to start a new
    drawing?", vbYesNo + vbQuestion, "New Drawing")

    If Response = vbYes Then picDraw.Cls

End Sub
```

In the mnuFileExit_Click procedure, make sure the user really wants to stop the application.

```
Private Sub mnuFileExit_Click()

    'Make sure user wants to quit

    Dim Response As Integer

    Response = MsgBox("Are you sure you want to exit the
    Blackboard?", vbYesNo + vbCritical + vbDefaultButton2,
    "Exit Blackboard")

    If Response = vbYes Then End

End Sub
```

When the left mouse button is clicked, drawing is initialized at the mouse cursor location in the picDraw_MouseDown procedure.

```
Private Sub picDraw_MouseDown(Button As Integer, Shift As
Integer, X As Single, Y As Single)

    'Drawing begins
    If Button = vbLeftButton Then

    DrawOn = True

    picDraw.CurrentX = X

    picDraw.CurrentY = Y

    End If

End Sub
```

When drawing ends, the DrawOn switch is toggled in picDraw_MouseUp.

```
Private Sub picDraw_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)

    'Drawing ends

    If Button = vbLeftButton Then DrawOn = False

End Sub
```

While mouse is being moved and DrawOn is True, draw lines in current color in the picDraw_MouseMove procedure.

```
Private Sub picDraw_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)

    'Drawing continues

    If DrawOn Then picDraw.Line -(X, Y), picDraw.ForeColor

End Sub
```

Finally, when a label box is clicked, the drawing color is changed in the lblColor_Click procedure.

```
Private Sub lblColor_Click(Index As Integer)

    'Make audible tone and reset drawing color

    Beep

    picDraw.ForeColor = lblColor(Index).BackColor

End Sub
```

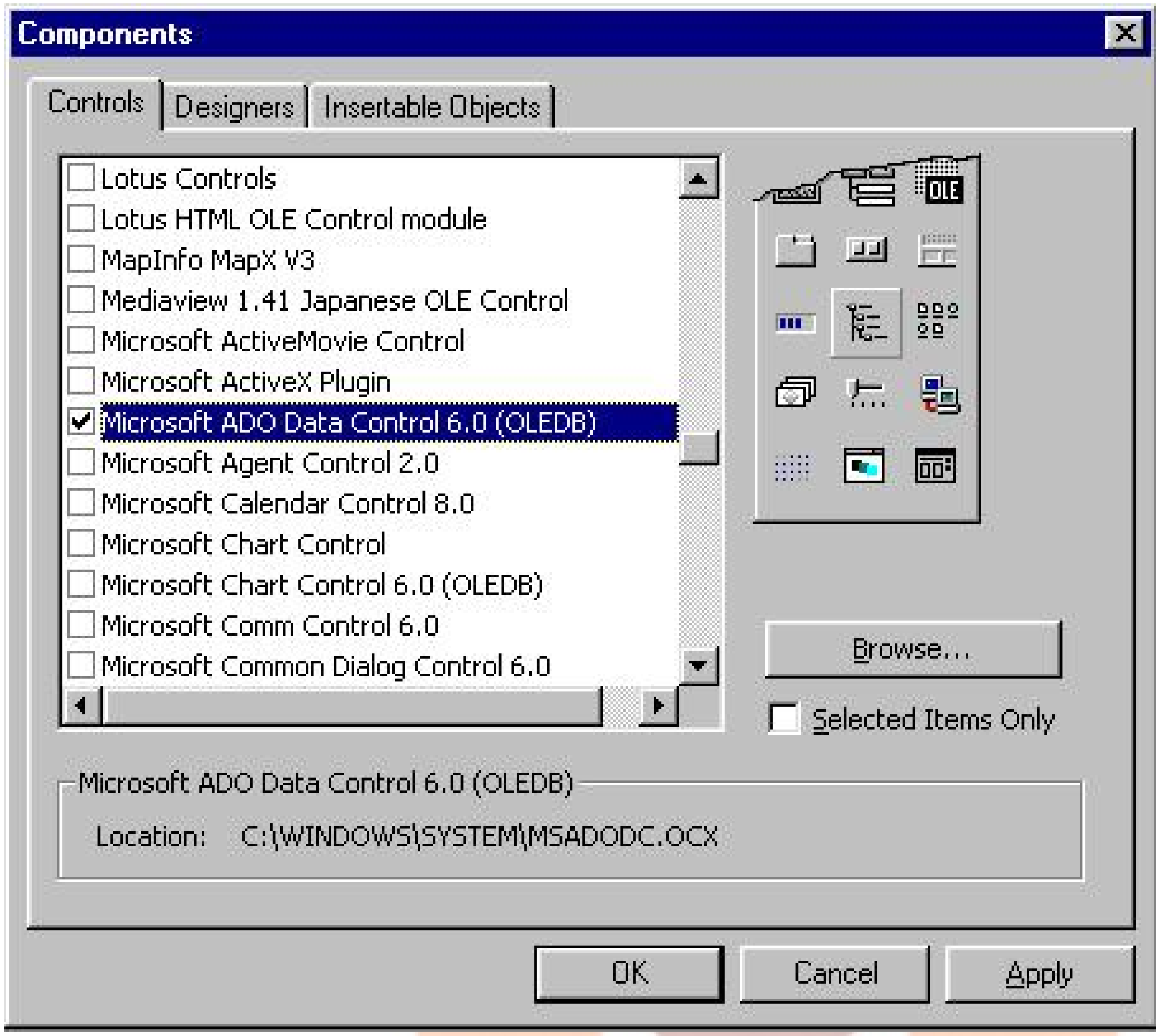
Run the application. Click on the label boxes to change the color you draw with. Fun, huh? Save the application.

ADO Collection

Introduction

The ADO Data control uses Microsoft ADO to quickly create connections between data-bound controls and data providers. Data-bound controls are any controls that feature a DataSource property, 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. When you bind controls to an ADO Data control, each field is automatically displayed and updated when navigating through records. This is done internally by Visual Basic; you do not have to write any code.

Library



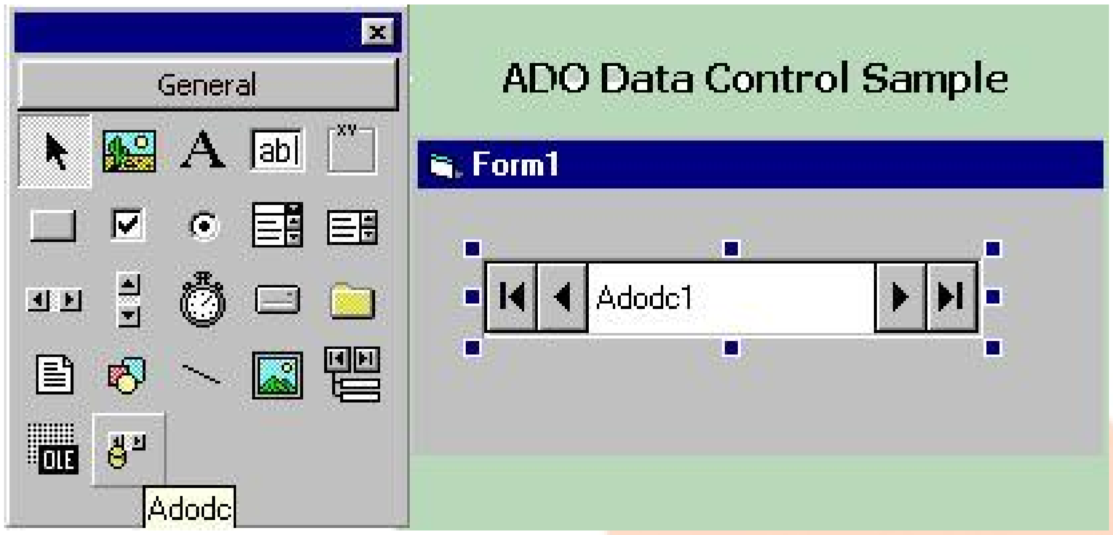
- [ ] Lotus Controls
- [ ] Lotus HTML OLE Control module
- [ ] MapInfo MapX V3
- [ ] Mediaview 1.41 Japanese OLE Control
- [ ] Microsoft ActiveMovie Control
- [ ] Microsoft ActiveX Plugin
- [✓] Microsoft ADO Data Control 6.0 (OLEDB)
- [ ] Microsoft Agent Control 2.0
- [ ] Microsoft Calendar Control 8.0
- [ ] Microsoft Chart Control
- [ ] Microsoft Chart Control 6.0 (OLEDB)
- [ ] Microsoft Comm Control 6.0
- [ ] Microsoft Common Dialog Control 6.0

- [ ] Selected Items Only

Microsoft ADO Data Control 6.0 (OLEDB)

Location: C:\WINDOWS\SYSTEM\MSADODC.OCX

☐Data Control



Connecting to a Data Source

At design time, you can create a connection to a data source by setting the ConnectionString property of the ADO Data control to a valid connection string. Then, set the RecordSource property to a table (or SQL statement) from which to retrieve records. Setting these properties is an easy process because Visual Basic provides Property Pages to set the values. When setting the ConnectionString property of the ADO Data control, you have three data source options.

- **Use Data Link File**

This option specifies that you are using a custom connection string that connects to the data source. When this is selected, you can click Browse to access the Organize Data Sources dialog box, from which you can select your Data Link file.

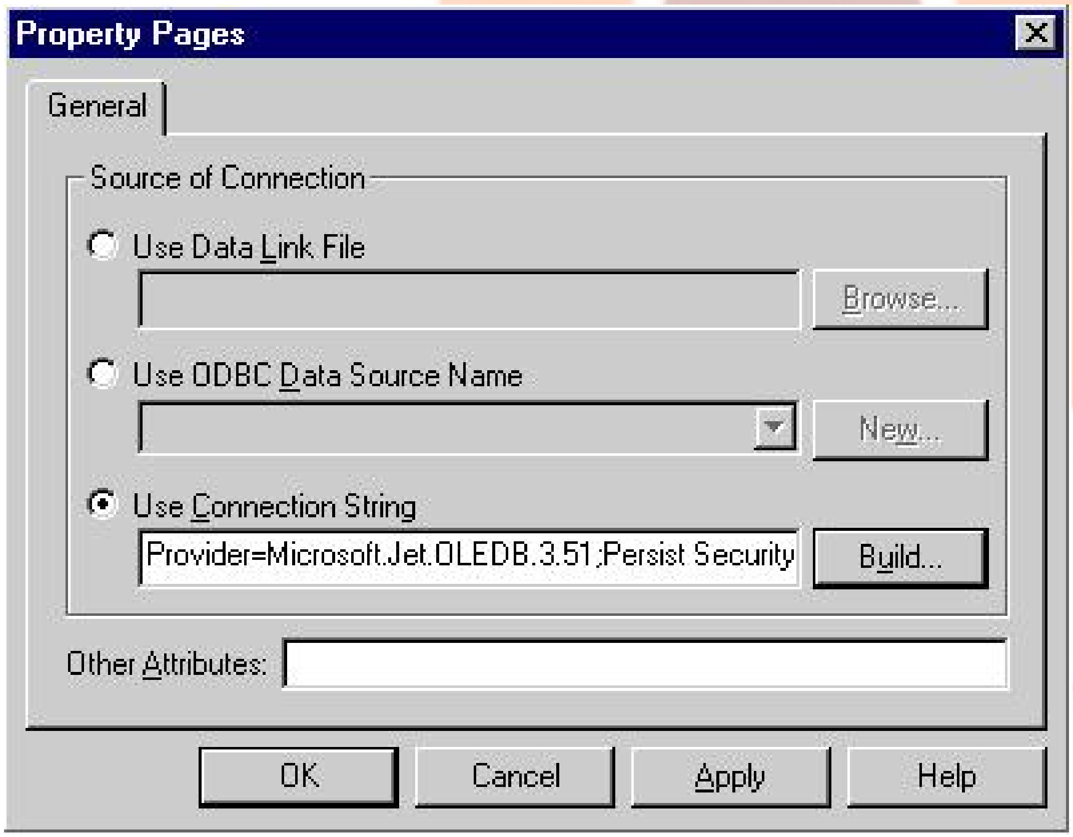
- **Use ODBC Data Source Name**

This option specifies that you are using a system-defined data source name (DSN) for the connection string. You can access a list

of all system-defined DSNs through the combo box in the Property Page dialog box, as illustrated in Figure below. When this option is selected, you can click New to access the Create New Data Source Wizard dialog box to add to or modify DSNs on the system.

- **Use Connection String**

This option specifies that you are using a connection string to access data. If the Use Connection String text box is empty, the wizard appears, or you can click Build to access the Data Link Properties dialog box. Use this dialog box to specify the connection, authentication, and advanced information required to access data using an OLE DB provider.



Provider=Microsoft.Jet.OLEDB.3.51;Persist Security

Setting a Connection String

In the following procedure, we will focus on using a connection string to connect to a data source. In this process, you will choose an OLE DB provider, specify a database name and location, and test the connection.

To set the ConnectionString property value

1. **Place an ADO Data control on a form.**

2. **On the Properties window, click the ConnectionString property, then click the ellipsis (&ldots;) to open the Property Pages.**

3. **Click on the ellipsis located on the right side of the ConnectionString property within the Properties window.**

4. **Click the Use Connection String option, then click Build.**

5. **Select the Microsoft Jet 3.51 OLE DB Provider, then click Next.**

6. **Click the ellipsis to the right of the Select or enter database name text box to browse the database name.**

7. **On the Select Access Database dialog box, click Nwind.mdb, then click Open.**

8. **Click Test Connection in the Data Link Properties window.**

   A message box will appear notifying you whether or not the connection succeeded.

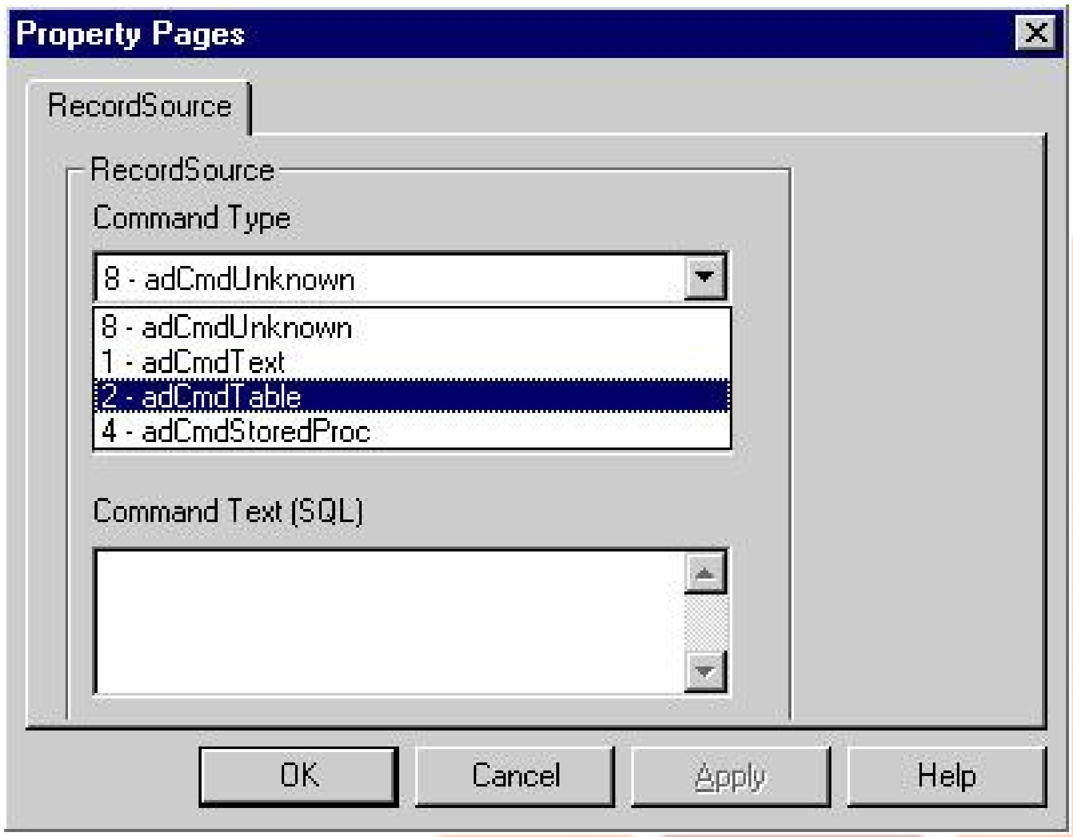
9. **Click OK to close the message box, then click OK to close the Data Link Properties window.**

A string value will be automatically generated for the Use Connection String value as illustrated in Figure 7.8.

10. Click OK to close the ConnectionString Property Pages window.

Setting the RecordSource Property

After you set the ConnectionString property of the ADO Data control to connect to a database, you can set the RecordSource property to establish where the records will come from. The RecordSource property can be set to either a database table name, a stored query name, or a Structured Query Language (SQL) statement. To improve performance, avoid setting the RecordSource property to an entire table. Set the RecordSource to a n SQL string that retrieves only the necessary records. An SQL query must use syntax appropriate for the data source. In other words, Microsoft Access and Microsoft SQL Server use different SQL syntax; therefore, you must use the appropriate syntax for the particular database.



In the RecordSource property page dialog box, you set the command type parameter that tells ADO which type of command object to use. The following table explains the different command type options.

Value	**Description**
adCmdUnknown	The type of command in the CommandText property is not known. This is the default value.
adCmdText	Evaluates CommandText as a textual definition of a command or stored procedure call.
adCmdTable	Evaluates CommandText as a table name whose columns are all returned by an internally generated SQL query.

adCmdStoredProc	Evaluates CommandText as a stored procedure name. This can be a stored procedure in a SQL Server database or a query in an Access database.

Binding Controls

After you set the ConnectionString and RecordSource properties for the ADO Data control, you can add a bound control to display data on your form.

A bound control is one that is "data-aware." When an ADO Data control moves from one record to the next, either through code or when the user clicks the ADO Data control arrows, all bound controls connected to the ADO Data control change to display data from fields in the current record. In addition, if the user changes the data in the bound control, those changes are automatically posted to the database as the user moves to another record. The benefit of using bound controls is that it minimizes the amount of code you must write. Because the value of the bound control is automatically retrieved from and written to the database, there is little or no programming involved.

- **The type of control depends on the type of data stored in the field.**

Data Type	Control
String, date, and numeric	TextBox

Boolean	CheckBox
Memo fields	Multi-line TextBox
Binary data	OLE Container
Picture	PictureBox

Setting the DataSource and DataField Properties

In order to bind a control to an ADO Data control, you must set the DataSource and DataField properties of the bound control. The DataSource property specifies the source through which the control is bound to the database (for example, an ADO Data control).

The DataField property specifies a valid field name in the Recordset object created by the data source. This value determines which field is displayed in the bound control. The DataSource and DataField properties can be set at design time in the Properties window. You can also set the DataSource and DataField properties at run time. If you set the DataSource property at run time using code, you must use the Set keyword because the DataSource property is an object. The following example sets the DataSource and DataField properties:

```
Set txt1.DataSource = Adodc1

txt1.DataField = "CompanyName"
```

□ADODB in detail

The following code is added to declare a connection object and a recordset object to access a database.

Dim cn as ADODB.Connection

Dim rs as ADODB.Recordset

cn = connection object which varry from data provider to provider

rs = recordset object is used to access data from table where connection object is used to direct recordset for table. Syntex for this:

rs.open sql, cn, adOpenForwardOnly, adLockReadOnly

here we notice that several parameters are required to opne the recordset. The forst parameter passes the SQL query to the recordset. The second paramenter, cn, tells the recordset to use the connection object cn to get to the database. The next paramenter tells ADO which type of database cursor to use on the recordset. ADO can use many types of cursors. The table below lists the possible options and what they do:

Cursor Type	Desciption
AdOpenForwardOnly	This type of cursor can only be used to move forward through the recordset. This option is used when a listbox or combo box is to be populated.

AdOpenKeyset	This is the best type of cursor to use when we expect a large recordset because we are not informed when changes are made to data that can affect out recordset.
AdOpenDynamic	This cursor allows us to see all the changes made by other users that affect our recordset. It is the most powerful type of cursor but the slowest one.
AdOpenStatic	The static cursor is useful when we have a small recordset.

The last parameter tells ADO that we want the recordset to be read only.

LockType	Description
AdLockReadOnly	This lock mode is used when no additions, updates or deletions are allowed from the recordset.
AdLockPesimistic	In pessimistic locking, the record is locked as soon as editing begins and remains locked until editing is completed, or the cursor moves to another record.

AdLockOptimistic	This occurs when the .Update method is called on the record. The record is unlocked even while we edit, but is temporarily locked when the changes are saved to the database.
AdLockBatchOptimistic	This option allows us to perform optimistic locking when we are updating a batch of records.

Lesson Summary

- **The ADO Data control is a graphic control (with record navigation buttons) and an easy-to-use interface that allows you to create database applications with a minimum of code. To use the ADO Data control in Visual Basic 6.0, you must add it to the toolbox.**

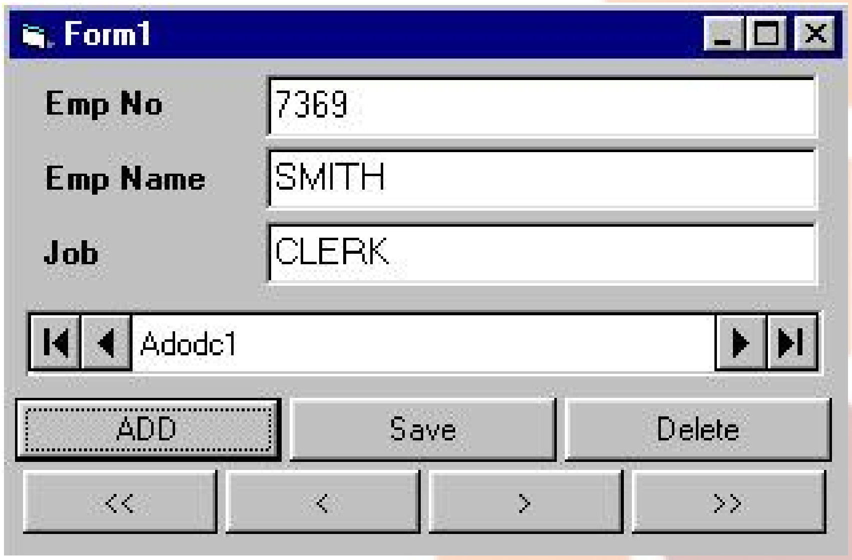
- **At design time, you can create a connection to a data source by setting the ConnectionString property of the ADO Data control to a valid connection string. After you set the ConnectionString property of the ADO Data control to connect to a database, you can set the RecordSource property to establish where the records will come from.**

- **Once you have set the ConnectionString and RecordSource properties for the ADO Data control, you can add a bound**

control to display data on your form.

ADO Bound programming (Example: 1)

Screen



Table

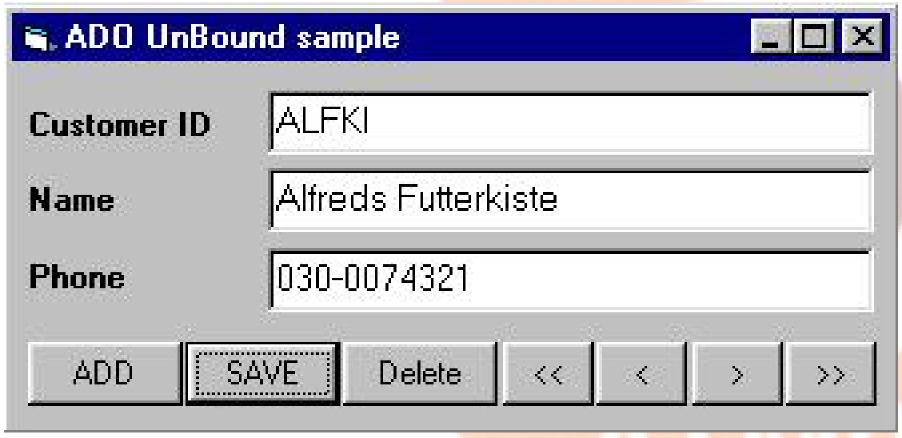
SrNo	Object Name	Type	Properties
1	txtEmpNo	Textbox	.FontSize = 10 Datasource : Ado1 Datafield : EMPNO

2	txtEname	Textbox	.FontSize = 10 Datasource : Ado1 Datafield : ENAME
3	txtJob	Textbox	.FontSize = 10 Datasource : Ado1 Datafield :JOB
4	cmdAdd	Command Button	.Caption "ADD"
5	cmdSave	Command Button	.Caption "SAVE"
6	cmdDelete	Command Button	.Caption "Delete"
7	cmdFirst	Command Button	.Caption "<<"
8	cmdPrev	Command Button	.Caption "<"
9	cmdNext	Command Button	.Caption ">"
10	cmdLast	Command Button	.Caption ">>"

11	Form1	Form	.Border style 1 - Fixed Single
12	Label 1 , Label 2, Label 3	Labels	FontBold = True .FontSize = 8 .Autosize = True

ADO UnBound programming (Example: 2)

Screen



Table

SrNo	Object Name	Type	Properties
1	txtCuId	Textbox	.FontSize = 10
2	txtName	Textbox	.FontSize = 10
3	txtPhone	Textbox	.FontSize = 10

4	cmdAdd	Command Button	.Caption "ADD"
5	cmdSave	Command Button	.Caption "SAVE"
6	cmdDelete	Command Button	.Caption "Delete"
7	cmdFirst	Command Button	.Caption "<<"
8	cmdPrev	Command Button	.Caption "<"
9	cmdNext	Command Button	.Caption ">"
10	cmdLast	Command Button	.Caption ">>"
11	frmADOUnbound	Form	.Caption ADO UnBound sample Border style 1 - Fixed Single
12	Label 1 , Label 2, Label 3	Labels	FontBold = True .FontSize = 8 .Autosize = True

Code

```
Option Explicit

    '---connection object

        Dim db As New ADODB.Connection

    '---recordset object

        Dim rs As New ADODB.Recordset
```

```
Private Sub cmdAdd_Click()

    '---adds new record in the database

    rs.AddNew

    '---keep textboxes clean for user entry

    txtCuId.Text = vbNullString

    txtName.Text = vbNullString

    txtPhone.Text = vbNullString

End Sub
```

```
Private Sub cmdDelete_Click()

    '---deletes record from database
    rs.Delete

    '---keep textboxes clean for user entry

    txtCuId.Text = vbNullString

    txtName.Text = vbNullString

    txtPhone.Text = vbNullString

End Sub
```

```
Private Sub cmdFirst_Click()

    '---moves cursor to first record

    rs.MoveFirst

    '---assign data to text boxes

    Data

End Sub
```

```
Private Sub cmdLast_Click()

    '---moves cursor to last position

    rs.MoveLast

    '---assign data to text boxes
    Data
End Sub
```

```
Private Sub cmdNext_Click()
    '---check record is last or not (EOF = End of file)
    If rs.EOF = False Then
      rs.MoveNext
      If rs.EOF = False Then
      '---assign data to text boxes
      Data
      End If
    End If
End Sub
```

```
Private Sub cmdPrev_Click()
'---check record is First or not (BOF = Beginning of file)
    If rs.BOF = False Then
      rs.MovePrevious
```

```
        If rs.BOF = False Then
        Data
        End If
    End If
End Sub
```

```
Private Sub cmdSave_Click()
    '---assign value from textbox to recordset
    rs("CustomerID") = txtCuId.Text
    rs("CompanyName") = txtName.Text
    rs("Phone") = txtPhone.Text
    '---update record
    rs.Update
End Sub
```

```
Private Sub Form_Load()
    '---Open database connection
    db.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
    Source=C:\Program Files\Microsoft Visual
    Studio\VB98\Nwind.mdb;Persist Security Info=False"

    '---open recordset for table data

    rs.Open "select * from Customers", db, adOpenStatic,
    adLockOptimistic

    '---if record available then assign first record to text boxes

    If rs.RecordCount > 0 Then

      Data

    End If

End Sub
```

```
Private Sub Data()

    '---private function/ procedure to assign data from recordset
    to textbox

    txtCuId.Text = rs("CustomerID")

    txtName.Text = rs("CompanyName")

    txtPhone.Text = rs("Phone")

End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)
```

```
    '---close record set

    rs.Close

    '---release memory

    Set rs = Nothing

    '---close database connection

    db.Close

    Set db = Nothing

End Sub
```

◘ DAO Collection

◻Introduction

DAO is an object-oriented interface to Microsoft Jet, the engine that powers Access. Developers can design an MDB database using Access and then use DAO from a Visual Basic application to open the database, add and retrieve records, and manage transactions. The best thing about DAO is that it doesn't limit you to Jet databases because you can directly open any database for which an ODBC driver exists. Or you can use Jet attached tables, which are virtual tables that appear to belong to an MDB database but actually retrieve and store data in other ODBC sources.

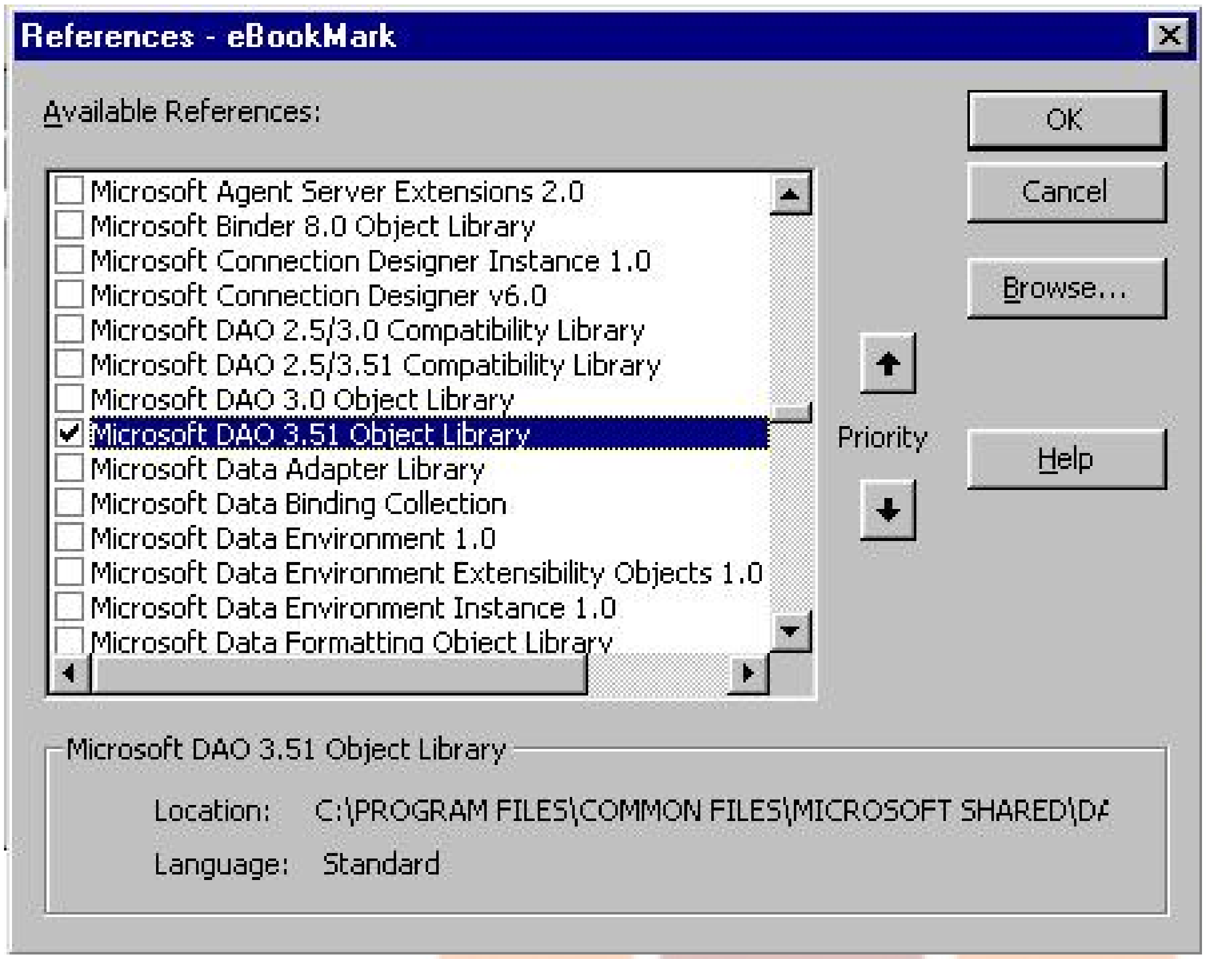
Even if you can use DAO to access non-Jet sources, you can clearly see that it was devised with Access databases in mind. For example, even if your application doesn't use MDB databases, you still have to load the entire Jet engine DLL in memory. (And you also have to distribute it to your users). Even worse, DAO doesn't expose many of the capabilities that you could use if working directly with ODBC API functions. For example, you can't perform asynchronous queries or connections using DAO, nor can you work with multiple result sets.

This control lets you bind one or more controls on a form to a data source and offers buttons for navigating through the records of the database table you've connected to. At first, it seems that the Data control is a great tool because it lets you quickly create effective

user interfaces to work with your data. After some testing, however, developers tend to abandon the Data control because its many limitations are difficult to overcome. Apart from performance considerations, the Data control has one serious disadvantage: It ties your front-end applications to the data in the back-end database. If you later want to access data in another database, you have to revise all the forms in your application. If you want to add complex validation rules to database fields, you must add code in every single module of the program. These (and other problems) are the typical defects of a 2-tier architecture, which in fact is being abandoned in favor of 3-tier (or n-tier) architectures, where one or more intermediate layers between the application and the database provide services such as data validation, business rules, workload balance, and security. Alas, if you want to embrace the n-tier philosophy, you should forget about the Data control.

Visual Basic 4 included the improved DAO 3.0 version, which features a special DLL that allows programmers who work with 32-bit technology to access 16-bit databases. Visual Basic 5 programmers can use DAO 3.5. In the Visual Basic 6 package, you'll find DAO 3.51, which is substantially similar to the previous one. This suggests that Microsoft doesn't plan to improve DAO further, even though version 4 has been announced for Microsoft Office 2000.

Library


References - eBookMark

Available References:

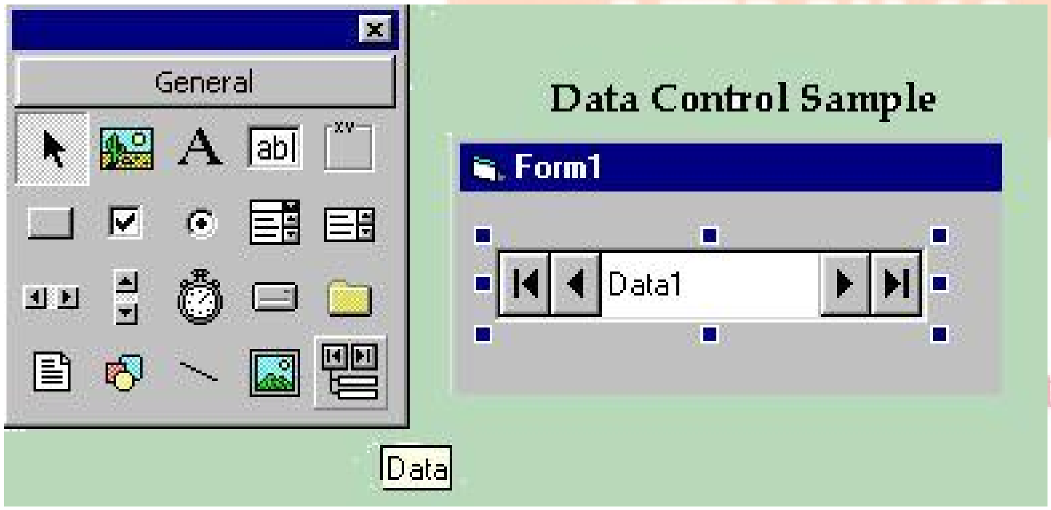
- [ ] Microsoft Agent Server Extensions 2.0
- [ ] Microsoft Binder 8.0 Object Library
- [ ] Microsoft Connection Designer Instance 1.0
- [ ] Microsoft Connection Designer v6.0
- [ ] Microsoft DAO 2.5/3.0 Compatibility Library
- [ ] Microsoft DAO 2.5/3.51 Compatibility Library
- [ ] Microsoft DAO 3.0 Object Library
- [✓] Microsoft DAO 3.51 Object Library
- [ ] Microsoft Data Adapter Library
- [ ] Microsoft Data Binding Collection
- [ ] Microsoft Data Environment 1.0
- [ ] Microsoft Data Environment Extensibility Objects 1.0
- [ ] Microsoft Data Environment Instance 1.0
- [ ] Microsoft Data Formatting Object Library

Microsoft DAO 3.51 Object Library

Location: C:\PROGRAM FILES\COMMON FILES\MICROSOFT SHARED\D

Language: Standard

☐Data Control



☐Accessing and Navigating Databases

in order to work with data access objects, a reference has to be set to the appropriate DAO library, There are two DAO libraries supported by Visual Basic 6.0. They are:

- **Microsoft DAO 3.51 Objects library**
- **Microsoft DAO 2.5/3.51 Compitibility Layer**

Opening a Database

To open an existing database, the OpenDatabase method of the workspace is used.

OpenDatabase (dbname, [options], [readonly], [connect])

The following code opens the employee_details database.

Dim db as database

Set db = OpenDatabase ("employee_details")

In the above code,db is **a variable that represents the Database objects. By default, a database that is opened can be shared and modified by any user. To specify that the database is to be opened for exclusive use, the following statement can be used:**

Set db = OpenDatabase ("employee_details " ,True)

In the above statement the **TRUE** value indicates that no other users will be able to open the database. The default value is False.

To open the employee_details database in the read only mode, the following statement is used:

Set db = OpenDatabase ("employee_details ",False,True)

In the above statement , the TRUE value specified as the third argument will provide only a read access on the database.

Recordset

A Recordset is an object that contains a set of records from the database.There are majorly five type of Recordset objects:

- **Table-Type Recordset**

The table_type recordset object is a set of records that represents a single table which can be used to add, change or delete records. They are the fastest type of recordset.

- **Dynaset-Type Recordset**

The dynaset-type of Recordset object is a set of records that represents a table, or attached tables, or the results of queries containing fields from one or more tables. A Dynaset enables us to update data from more than one table.

- **Snapshot-Type Recordset**

The snapshot-Type Recordset can refer any table, attached table or query. A snapshot cannot be updated and does not reflect changes to data made by the users.

- **Dynamic Type Recordset**

This Recordset type represents a query result set from one or more base tables in which we add, change , or delete records from a row-returning query.Further ,records that other user add , delete, or edit in the base tables also appear in our Recordset. This type is only available in ODBCDirect workspaces, and corresponds to an ODBC dynamic cursor.

- **Forward Only Type Recordset**

This Recordset type is identical to a snapshot except that we can only scroll forward through its records.This improves performance in situations where we only need to make a single pass through a result set. In an ODBCDirect workspace, this type corresponds to an ODBC forward_only cursor.

Creating a Recordset

The OpenRecordset method is used to open a Recordset and create a Recordset variable.

Example

To create a read-only Recordset for the table employee, the following code is used:

Dim rs as Recordset

Set rs = db.OpenRecordset("employee", dbOpentable, dbReadonly)

In the above statement, db is the variable that represents the **Database** object. Here dbOpenTable specifies the type of Recordset to be created.

Navigating a Recordset

After creating a Recordset object, the various **Move** methods can be used to navigate through the records in a Reocrdset.

MoveFirst	moves to the first row in the Recordset.
MoveNext	moves to the next row in the reocrdset.
MovePrevious	method moves to the previous row in the recordset
MoveLast	method moves to the last row in the recordset.

Using BOF and EOF to Navigate through Recordsets

The Recordset object provides two properties for the user to know when he has moved to the begining or end of the recordset.

- The EOF (End of File) property is True when the user moves beyond the last record in the recordset.
- The BOF (Begining of File) property is True when the user has moved to a position before the first record in the recordset.

Modifying and Deleting Records

To manipulate a record in a recordset, the following methods are used.

Edit Method : The user can edit the current record using the Edit method. The Update method is used to save the necessary changes made to the record.

AddNew Method : AddNew method is used to create a new record in the database.

Delete Method : This method can be used to delete an existing record in the dynaset-type or table-type Recordset. The Jet engine deletes the current record without any warning when the Delete method is used.

Finding Records

The Find method can be used to locate a record in a dynaset-type or snapshot-type Recordset.Visual Basic supports four **Find** methods.

FindFirst	method finds the first record satisfying the specified criteria.
FindLast	method finds the last record satisfying the specified criteria.
FindNext	method finds the next record satisfying the specified criteria,searching forward from the current record.
FindPrevious	method finds the previous record satisfying the specified criteria, searching backward from the current record.

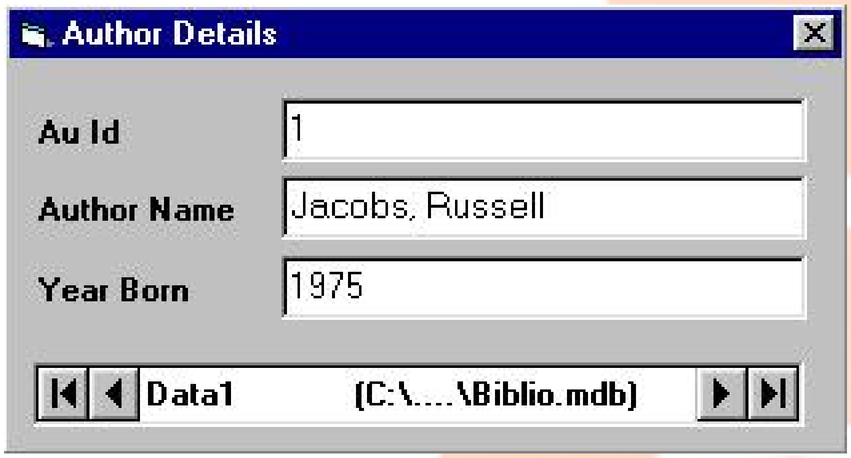
When the database engine finds a match for the criteria that is specified, it moves to that record. If no match is found, the current record is unchanged and the Recordset object's **No Match** property is set to True.

Performing Indexed Searches Using the Seek Method

The **Seek** Method can be used to locate a record in a table-type Reocrdset.This method performs an indexed search for the first occurence of the record that matches the indexed criteria.Dynasets and Snapshots cannot use the **Seek** method.

☐ DAO Bound programming (Example: 1)

Screen



Au Id: 1
Author Name: Jacobs, Russell
Year Born: 1975

Table

Sr No	Object Name	Type	Properties

1	txtAuId	TextBox	.FontSize = 10
			
			DataSource = Data1
			
			DataField = Au_Id

2	txtAuthor	TextBox	.FontSize = 10
			
			DataSource = Data1
			
			DataField = Author

3	txtYearBorn	TextBox	.FontSize = 10
			
			DataSource = Data1
			
			DataField = Year Born

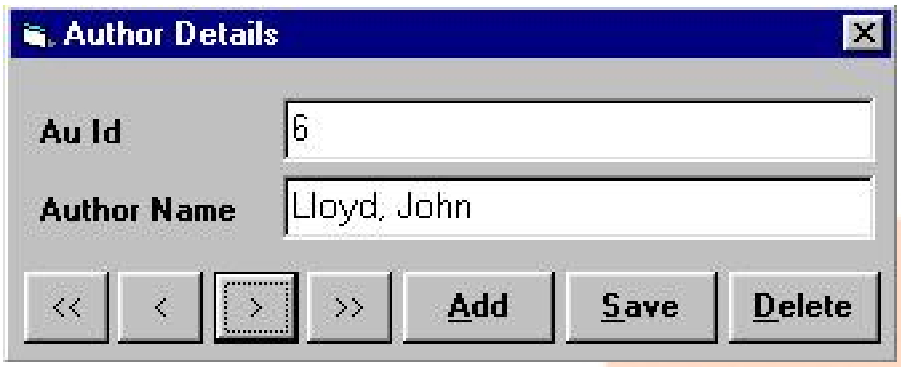
4	Data1	Data Control	Name = Data1
			
			DatabaseName = "C:\Program Files\Microsoft Visual Studio\VB98\Biblio.mdb"
			
			RecordSource = Authors

5	frmAuthor	Form	BorderStyle = 1 - Fixed Single
			
			Caption = "Author Details"

6	label1, label2, label3	Labels	.FontBold = True .FontSize = 8

☐DAO UnBound programming (Example: 2)

Screen



Table

Sr No	Object Name	Type	Properties
1	txtAuId	TextBox	.FontSize = 10
2	txtAuthor	TextBox	.FontSize = 10

3	frmAuthor	Form	BorderStyle = 1 - Fixed Single
			
			Caption = "Author Details"

4	cmdAdd	Command Button	.Caption = "Add"
5	cmdSave	Command Button	.Caption = "Save"
6	cmdDelete	Command Button	.Caption = "Delete"

7	cmdFirst	Command Button	.Caption = "<<"
8	cmdPrev	Command Button	.Caption = "<"
9	cmdNext	Command Button	.Caption = ">"
10	cmdLast	Command Button	.Caption = ">>"
11	label1, label2, label3	Labels	.FontBold = True .FontSize = 8

Code

```
Option Explicit

'---required variable declaration

Dim db As Database

Dim rs As Recordset
```

```
Private Sub cmdAdd_Click()

    '---generates new record in database
    rs.AddNew

End Sub
```

```
Private Sub cmdDelete_Click()

    '---deletes record

    rs.Delete

    '---sets text box clean for user

    txtAuId.Text = ""

    txtAuthor.Text = ""

End Sub
```

```
Private Sub cmdFirst_Click()

    '---sets recordset to first record

    rs.MoveFirst

    '---assigns record to text boxes

    txtAuId.Text = rs("Au_Id")

    txtAuthor.Text = rs("Author")

End Sub
```

```
Private Sub cmdLast_Click()

    '---sets recordsets to last record

    rs.MoveLast

    '---assigns record to text boxes

    txtAuId.Text = rs("Au_Id")

    txtAuthor.Text = rs("Author")

End Sub
```

```
Private Sub cmdNext_Click()

    '---sets recordsets to next record

    rs.MoveNext

    '---assigns record to text boxes

    txtAuId.Text = rs("Au_Id")

    txtAuthor.Text = rs("Author")

End Sub
```

```
Private Sub cmdPrev_Click()

    '---sets recordsets to previous record
    rs.MovePrevious

    '---assigns record to text boxes

    txtAuId.Text = rs("Au_Id")

    txtAuthor.Text = rs("Author")

End Sub
```

```
Private Sub cmdSave_Click()

    '---assign text box values to recordset

    rs("Au_id") = txtAuId.Text

    rs("Author") = txtAuthor.Text

    '---update in database

    rs.Update

End Sub
```

```
Private Sub Form_Load()

    '---open database object

    Set db = OpenDatabase("C:\Program Files\Microsoft Visual
    Studio\VB98\Biblio.mdb")
```

```
        '---fatch records in recordset

        Set rs = db.OpenRecordset("Authors", dbOpenDynaset)

End Sub
```

□ Types of Errors

Debugging in Visula Basic

Debugging is the process by which errors are identified and resolved in source code.Typically, debugging is conducted at various levels. This chapter identifies at least three levels of debugging that occur in virtually every software program of any size.

Handling Logical errors

The first level of debugging is the implementation of a debugging procedure and the necesary tuning to make it operate as expected. Logical errors occur when the application dosen't perform as intended and produces incorrect results. For example, when a programmer sets to write a procedure, he/she would have in mind only the expected output of that procedure, i.e the programmer knows the procedure is supposed to acomplish, but has not yet determined the optimal method of coding its functionality.

Runtime error

Second-level debugging is the activity requried to make a functional unit of code interact according to plan with other units of code, typically before shipping or deploying the completed project.Generally the condition arises when our procedure does not run properly in a test case.

Human Errors

is priliminary the diagnosis and repair of problems occuring in applications that have already been deployed. These bugs are often the most difficult to locate because they occur on a remote machine. They may be the result of unanticipated circumstances, such as:

User actions (keystrokes, menu choices, options settings)

Program configurationn (option settings,states, call stack)

System configuration (operating system versions, system DLL versions, disk space memory drivers and so on)

Areas of occurrence of Bugs

In Visual Basic,errors can occur anywhere during the entire development cycle such as

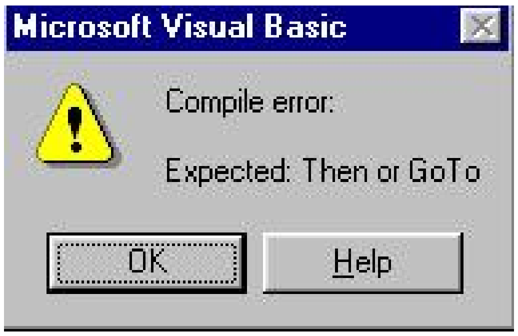
- **Design Time**
- **Compile Time**
- **Run Time**

□ Design Time Bugs

Errors that occur in the IDE and before the program is compiled are called design-time bugs.These are the most common ones.These bugs are caused by misuse of some component.

The following example shows that while you declare 'xy' variable and going to check condition for the same and you omit "Then" expression which is required, that time design time error occurs which is displayed the moment the user navigates to the next statement.

```
Private Sub Command1_Click()

    Dim xy as integer

    xy = 11

    If xy = 10

    ...........

End Sub
```

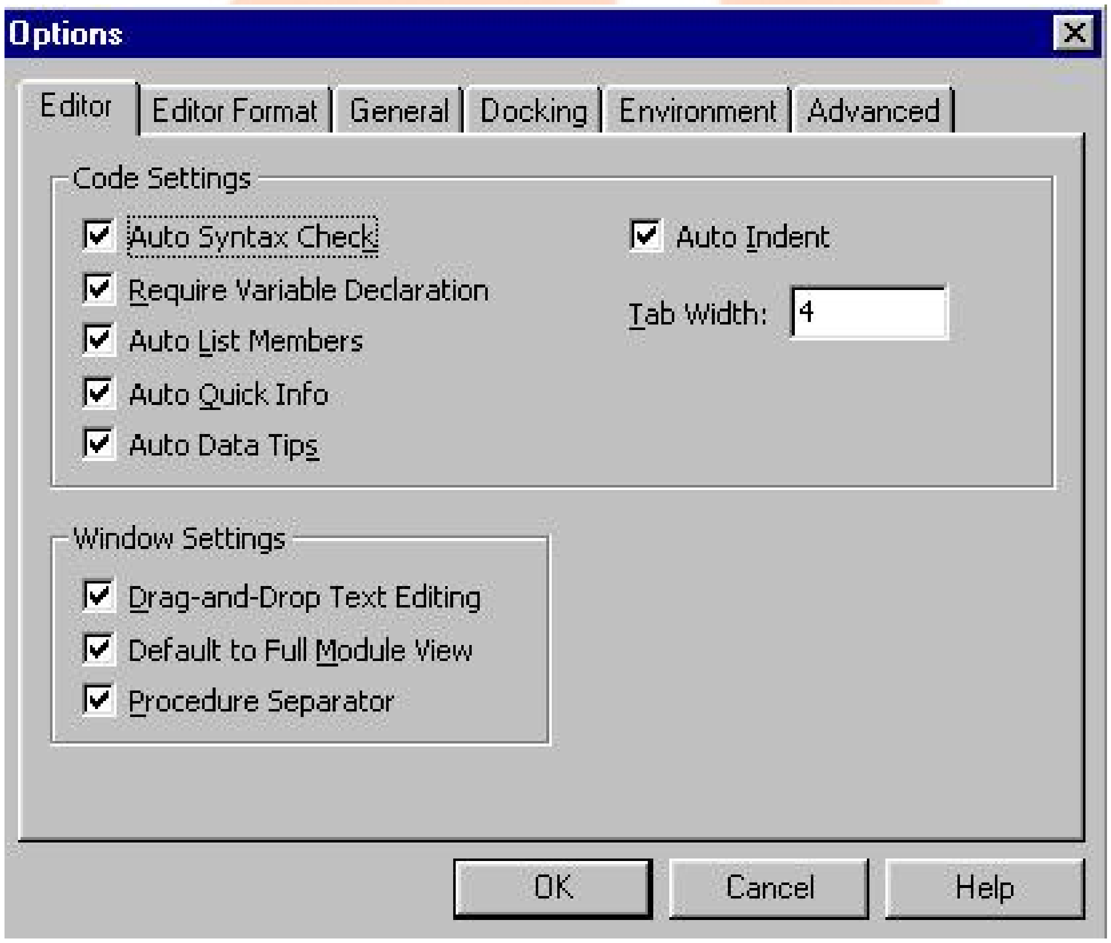


Microsoft Visual Basic

Compile error:

Expected: Then or GoTo

This type of error can be trapped using Auto Syntax Check option in option dialog box, which is shown below



Options

Editor | Editor Format | General | Docking | Environment | Advanced

Code Settings

- [✓] Auto Syntax Check
- [✓] Require Variable Declaration
- [✓] Auto List Members
- [✓] Auto Quick Info
- [✓] Auto Data Tips
- [✓] Auto Indent

Tab Width: 4

Window Settings

- [✓] Drag-and-Drop Text Editing
- [✓] Default to Full Module View
- [✓] Procedure Separator

□ Run time errors

- **A visual Basic *runtime error* is the exception generated by Visual Basic when it acertains that the code is about to perform something illegal. An illegal function could be something as simple as trying to determine the size of a file that does not exist or attempting to multiply two numbers, the result of wwhich exceeds the storage space that could be contained by the datatype.**

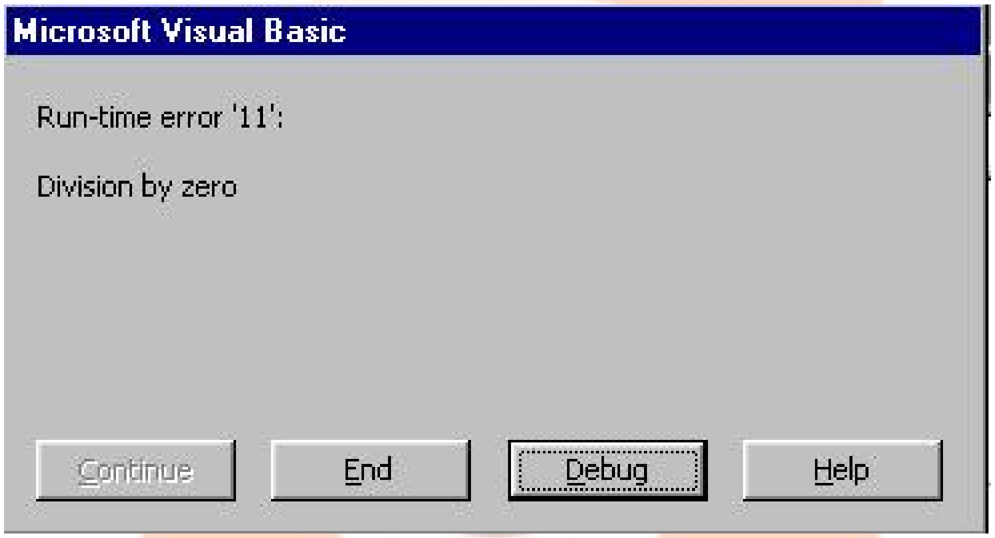
However, using an error handler could prevent this error dialogue box from being shown and save the progam from crashing. By practice we can also use out knowledge that a particualr error identified by its error number will be generated to help guide our logic. This sort of "inline" error handling is very powerful and can be handled using an error handler that checks for the condition of say Eror 11, and then does something to circumvent the situation. This error can be finally rectified by changing the code suitably.

Example:

Consider the following example where we declare two variables x and y. Try dividing x (assigned a value 9) by y (assigned 0) assigning the quotient to another variable z.

```
Private Sub Form_Load()

    Dim x As Integer

    Dim y As Integer
```

```
        Dim z As Integrer

        x = 9

        y = 0

        z = x / y

    End Sub
```


Microsoft Visual Basic

Run-time error '11':

Division by zero

On executing the above application, a run time error is reported as shown in Example. indicating a division operation by zero which is not possible.

Note:

This type of Error can be handled using Err object which is explained in next topic.

□ Complie-Time Error

Compile-time bugs are those that occur when we attempt to create the program executable file (EXE) or run the project. Visual Basic can locate compile-time bugs if the Visual Basic application is set up correctly. Visual Basic sets several options for the users, which can be changed as needed.

Compile-time bugs are detected by Visual Basic automatically when the program is compiled using F5. If the *Start With Full Compile* options is not used, the ?***Compile On Demand*** can be turned on, and the developer will not find many bugs until the line of code with the bug is actually executed.

Compile errors occur as a result of incorrectly constructed code such as a Next statement without a corresponding For statement or programming mistakes that violate the rules of Basic, such as a mispelt word, missing separator, or type mismatch. Compile errors include syntax errors. These include mismatched parentheses or an incorrect number of arguments passed to an intinsic function.

Err Object

- **Run-time errors are trappable. That is, Visual Basic recognizes an error has occurred and enables you to trap it and take corrective action. If an error occurs and is not trapped, your program will usually end in a rather unceremonious manner.**

- **Error trapping is enabled with the On Error statement:**

```
On Error GoTo errlabel
```

Yes, this uses the dreaded GoTo statement! Any time a run-time error occurs following this line, program control is transferred to the line labeled errlabel. Recall a labeled line is simply a line with the label followed by a colon (:).

- **The best way to explain how to use error trapping is to look at an outline of an example procedure with error trapping.**

```
Sub SubExample()

        [Declare variables, ...]

On Error GoTo HandleErrors

        [Procedure code]

Exit Sub
```

```
    HandleErrors:

        [Error handling code]

End Sub
```

Once you have set up the variable declarations, constant definitions, and any other procedure preliminaries, the On Error statement is executed to enable error trapping. Your normal procedure code follows this statement. The error handling code goes at the end of the procedure, following the HandleErrors statement label. This is the code that is executed if an error is encountered anywhere in the Sub procedure. Note you must exit (with Exit Sub) from the code before reaching the HandleErrors line to avoid inadvertent execution of the error handling code.

- **Since the error handling code is in the same procedure where an error occurs, all variables in that procedure are available for possible corrective action. If at some time in your procedure, you want to turn off error trapping, that is done with the following statement:**

```
    On Error GoTo 0
```

- **Once a run-time error occurs, we would like to know what the error is and attempt to fix it. This is done in the error handling** code.

- **· Visual Basic offers help in identifying run-time errors. The Err object returns, in its Number property (Err.Number), the number associated with the current error condition. (The Err function has other useful properties that we won't cover here - consult on-line help for further information.) The Error() function takes this error number as its argument and returns a string** description of the error. Consult on-line help for Visual Basic run-time error numbers and their descriptions

- Once an error has been trapped and some action taken, control must be returned to your application. That control is returned via the Resume statement. There are three options

  Resume

    Lets you retry the operation that caused the error. That is, control is returned to the line where the error occurred. This could be dangerous in that, if the error has not been corrected (via code or by the user), an infinite loop between the error handler and the procedure code may result.

  Resume Next

    Program control is returned to the line immediately following the line where the error occurred.

  Resume label

    Program control is returned to the line labeled label.

- Be careful with the Resume statement. When executing the error handling portion of the code and the end of the procedure is encountered before a Resume, an error occurs. Likewise, if a Resume is encountered outside of the error handling portion of the code, an error occurs.

General Error Handling Procedure

- **Development of an adequate error handling procedure is application dependent. You need to know what type of errors you are looking for and what corrective actions must be taken if these errors are encountered. For example, if a 'divide by zero' is found, you need to decide whether to skip the operation or do something to reset the offending denominator.**

- **What we develop here is a generic framework for an error handling procedure. It simply informs the user that an error has occurred, provides a description of the error, and allows the user to Abort, Retry, or Ignore. This framework is a good starting point for designing custom error handling for your applications.**

- **The generic code (begins with label HandleErrors) is:**

```
HandleErrors:

Select Case MsgBox(Error(Err.Number), vbCritical +
vbAbortRetryIgnore, "Error Number" + Str(Err.Number))

  Case vbAbort

  Resume ExitLine

  Case vbRetry

  Resume

  Case vbIgnore

  Resume Next

End Select
```

ExitLine:

Exit Sub

Let's look at what goes on here. First, this routine is only executed when an error occurs. A message box is displayed, using the Visual Basic provided error description [Error(Err.Number)] as the message, uses a critical icon along with the Abort, Retry, and Ignore buttons, and uses the error number [Err.Number] as the title. This message box returns a response indicating which button was selected by the user. If Abort is selected, we simply exit the procedure. (This is done using a Resume to the line labeled ExitLine. Recall all error trapping must be terminated with a Resume statement of some kind.) If Retry is selected, the offending program line is retried (in a real application, you or the user would have to change something here to correct the condition causing the error). If Ignore is selected, program operation continues with the line following the error causing line.

- **To use this generic code in an existing procedure, you need to do three things:**

1. Copy and paste the error handling code into the end of your procedure.

2. Place an Exit Sub line immediately preceding the HandleErrors labeled line.

3. Place the line, On Error GoTo HandleErrors, at the beginning of your procedure.

For example, if your procedure is the SubExample seen earlier, the modified code will look like this:

Sub SubExample()

```
        .[Declare variables, ...]

    On Error GoTo HandleErrors

        [Procedure code]

    Exit Sub

    HandleErrors:

    Select Case MsgBox(Error(Err.Number), vbCritical +
    vbAbortRetryIgnore, "Error Number" + Str(Err.Number))

        Case vbAbort

            Resume ExitLine

        Case vbRetry

            Resume

        Case vbIgnore

            Resume Next

    End Select

    ExitLine:

    Exit Sub

    End Sub
```

Again, this is a very basic error-handling routine. You must determine its utility in your applications and make any modifications necessary. Specifically, you need code to clear error conditions

before using the Retry option.

- **One last thing. Once you've written an error handling routine, you need to test it to make sure it works properly. But, creating run-time errors is sometimes difficult and perhaps dangerous. Visual Basic comes to the rescue! The Visual Basic Err object has a method (Raise) associated with it that simulates the occurrence of a run-time error. To cause an error with value Number, use:**

Err.Raise Number

- **We can use this function to completely test the operation of any error handler we write. Don't forget to remove the Raise statement once testing is completed, though! And, to really get fancy, you can also use Raise to generate your own 'application-defined' errors. There are errors specific to your application that you want to trap.**

- **To clear an error condition (any error, not just ones generated with the Raise method), use the method Clear:**

Err.Clear

Example

Simple Error Trapping

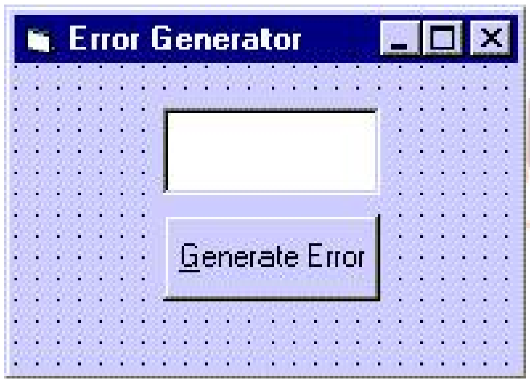
1. Start a new project. Add a text box and a command button.

2. Set the properties of the form and each control:

Sr No	Object Name	Properties

1	Form 1	BorderStyle : 1-Fixed Single Caption : Error Generator Name : Error
2	Command 1	Caption :Generate Error Default : True Name : cmdGenError
3	Text 1	Name : txtError Text : {Blank]

The form should look something like this:



3. Attach this code to the cmdGenError_Click event.

```
Private Sub cmdGenError_Click()

      On Error GoTo HandleErrors

      Err.Raise Val(txtError.Text)
```

```
 Err.Clear

 Exit Sub

 HandleErrors:

 Select Case MsgBox(Error(Err.Number), vbCritical +
vbAbortRetryIgnore, "Error Number" + Str(Err.Number))

  Case vbAbort

        Resume ExitLine

  Case vbRetry

        Resume

  Case vbIgnore

        Resume Next

 End Select

 ExitLine:

Exit Sub

End Sub
```

In this code, we simply generate an error using the number input in the text box. The generic error handler then displays a message box which you can respond to in one of three ways.

4. Save your application. Try it out using some of these typical error numbers (or use numbers found with on-line help). Notice how program control changes depending on which button is clicked.

Error Number	Error Description
6	Overflow
9	Subscript out of range
11	Division by zero
13	Type mismatch
16	Expression to complex
20	Resume with Error
52	Bad file name or number
53	File not found
55	File already open
61	Disk Full
70	Permission denied.
92	For loop not initialized

□ Error Trapping Options in VB

Visual Basic Debugging Tools

The best way to keep bugs out is to prevent them in the first place. Visual Basic gives a programmer several tools, including IDE options and compile directives, to help achieve this goal.

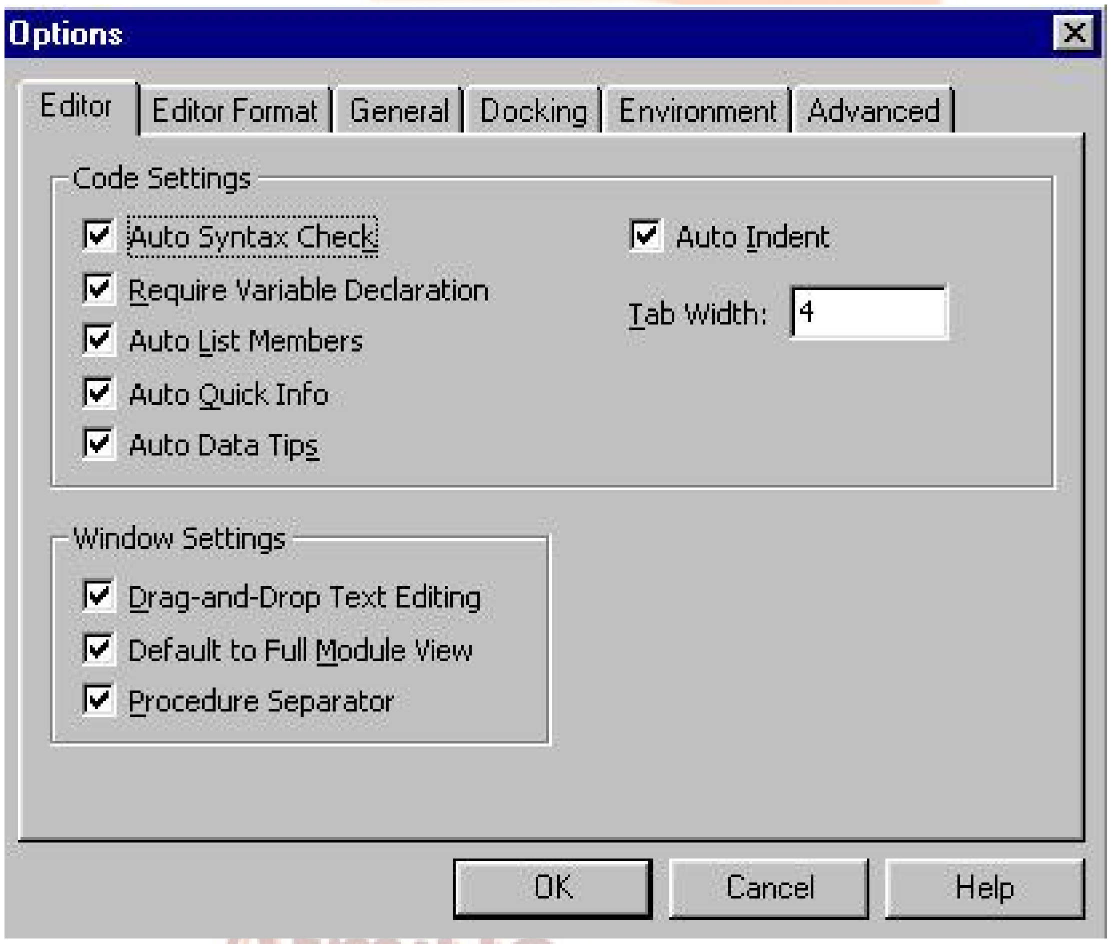
IDE Options

Visual Basic offers several IDE options that can help the user write better code.

- Auto Syntax Check
- Require Variable Declaration
- Auto List Members
- Auto Quick Info
- Auto data tips
- Option Explicit
- Option Compare Text

Additional settings and debugging aids can be viewed in a dialogue box shown in the Fig given below using the followwing steps

- Select options from Tools menu
- Choose the Editor tab in the options dialogue box.



Code Settings

[✓ Auto Syntax Check]
[✓ Require Variable Declaration]
[✓ Auto List Members]
[✓ Auto Quick Info]
[✓ Auto Data Tips]
[✓ Auto Indent]
Tab Width: 4

Window Settings

[✓ Drag-and-Drop Text Editing]
[✓ Default to Full Module View]
[✓ Procedure Separator]

Debugging tools are designed to help the user with logic and run-time errors. Visual Basic provides several buttons in the ToolBar

that are helpful for debugging. They are

- **BreakPoint -** Defines a line in the Code Window where Visual Basic suspends execution of application.
- **Instant Watch** - Lists the current value of an expression while the application is in the breakmode.
- **Calls** - Presents a dialogue box that shows all procedures that have been called but not yet run completely.

Stepping

Visual Basic provides several built-in methods for controlling the execution of the program in real time. It is possible to execute the program line-by-line or procedure-by-procedure or a combination of the two. These basic debugging actions are called Stepping. Because it enables the developer to walk through the program, examining the variables and logic. Stepping is the most powerful debugging tool offered by Visual Basic. The various Debug commands are discussed below.

- **Step Into -** Executes the next executable line of the code in the application and steps into procedures.Step into moves program execution to the procedure called by the method or property the cursor is currently pointing at. This option can be accessed from the Debug menu or by pressing F8. This enables to check every line of code as it is being executed

- **Step Over -**

  Similar to Step Into. The difference in use occurs when the current statement contains a call to a procedure.

Step Over executes the procedure as a unit, and then steps to the next statement in the current procedure. Therefore, the next statement displayed is the next statement in the current procedure regardless of whether the current statement is a call to another procedure. Available in break mode only.

- **Step Out -** Moves program execution back to the calling procedure.This is the functional equivalent of Exit Sub or Exit Function. It simply exists the current procedure without executing any more code in that procedure.

While debugging an application, we should clearly understand which of the three modes such as design time, run time or break mode we are in at a given time. Break mode of an application is viewed by clicking CTRL+BREAK at run time.

Using the Debug Window

Debug window is one of Visual Basic's windows that allows to run individual commands immediately, by typing in the command and pressing the Enter key. It automatically opens at run time.

- In break mode, the Debug window can be used to execute individual lines of code, view or change values of variales and properties, and view watch expressions.
- At run time , it can be used to display data or messages as the program runs.
- At design time, the developer can view the previous output to the Debug window, but cannot execute code.

The Debug Window has two parts - Watch Window and Immediate Window.The split bar separates the Debug window into two panes.The upper pane displays the Watch Window.The lower pane displays the Immmediate Window.

Watch Window

Watch pane displays the current watch expressions, which are expressions whose values are decided by the user as the code is executed.The Watch pane appears automatically if the watch expressions are defined in the project. At times we might want to monitor the value of a variable for a certain state-for example to determine whether a flag is set to True.

Monitoring Data with Watch Expressions

Many debugging problems are bot traced immediately with a single statement. So, we need to observe the behaviour of a variable or an expression throughout the procedure. Visual Basic automatically monitors the watch expressions, which appears in the Watch pane of the Debug window where the values can be observed. To add a watch expression, the following steps are needed

- Add Watch command is chosen from the Debug menu.
- The Expression is entered in the Expression box.
- If necessary,the scope of the variables is set and an option button is selected.
- The OK button is clicked.

To edit a Watch expression,

- Edit Watch command is chosen from the Debug menu. A dialogue box is displayed.

- The necessary changes are made and the OK button is clicked.

To delete a watch expression, the desired watch expression is selected in the watch pane and the Delete key is pressed.

Immediate Window

The immediate pane appears by default the first time the Debug window is opened. From break mode, the code is executed immediately by entering it in this pane.This window is the right place for the users to modify data or to test functions during development. We can enter any valid expression in this window and VB will execute it. If a refrence is made to an object outside the scope of the current code execution.Visual Basic will generate an error.

Testing Data and Procedures in Immediate Pane

While debugging an application, sometimes it may be necessary to execute individual procedures, evaluate expressions or assign new values to the variables or properites.The Immediate Pane can be used to accomplish such tasks. Expressions can be evaluated by printing in the immediate Pane. Information can be printed in the Immediate Pane by using Print methods directly. These printing techniques offer several advantages over watch expressions.

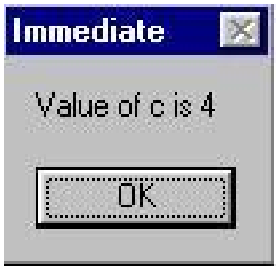
The data or messages can be viewed at run time.

Feedback is displayed in a separate area, so that it does not interface with the output that the user is seeing.

Since the coding is saved as part of the form, these statements need to be redefined the next time we work on the application.

Example

```
Private Sub Form_Load()

    Dim a As Integer

    Dim b, c As Integer

    a = 10

    b = 6

    c = a - b

    MsgBox "Value of c is " & c

    Debug.Print " Value of c is : " & c

End Sub
```



Immediate

Value of c is 4

Note

In break mode, a statement in the Immediate pane is executed in the context or scope displayed in the Procedure box.For example, if we type Print *variablename*, the output is the value of a local variable.This is similar to the Print method causing a procedure in execution to be halted.

Working in Break Mode

Break mode halts the operation of an application and gives a snapshot of its condition at any time.when an application is in break mode,we can

- Modify code in the application
- Observe the condition of application's interface
- Determine which active procedures have been called.Watch the value of the variables properties and statements.
- Change the values of variables and properties.
- Run Visual Basic statements immediately
- Manually control the operation of an application.

In addition to all these,there is also a Locals window that shows the value of every variable,and each member of all objects that are currently in scope.The Locals window can be viewed by selecting Locals from the View menu.

Using a Breakpoint to selectively Halt Execution

At runtime, a breakpoint tells Visual Basic to halt before executing a specific line of code.A breakpoint can be set or removed at design time or at break mode.To set or remove a breakpoint,

- The insertion point is moved to the line of code where the breakpoint is to be set or removed.
- The Toggle Breakpoint is chosen from the Debug menu or the Toggle Breakpoint button is chosen in the ToolBar.

Once a breakpoint is set,Visual Basic highlights the selected line in bold.On reaching the breakpoint, the application is halted and its current state examined.Checking the results of an application is easy,because it is possible to move the focus among the forms and modules of the application,code window and debug window.

Using Stop Statements to Enter Break Mode

Placing a stop statement in a procedure is an alternative way of setting a breakpoint.Whenever Visual Basic encounters a Stop statement, it halts execution and switches to breakmode.Although Stop statements act like breakpoints,they are not set and cleared in the same way.If the current porject is reloaded, the breakpoints are cleared,but Stop statemenst remain until they are manually removed.Stop statements also remain in the application when an executable file is created and they act just as End statements.

Debug Object

The Immediate window can also be used programmatically to output messages or state variables as the programs runs, by refrerring to it as the Debug object.This object is a good tool for getting information while we are debugging.The object has two methods *Print* and *Assert*

Print

The print method prints the result of any legal expression to Immediate window.This method can be used to display data in the Immediate window while the application runs.The *print* method is also used for **tracing**. *Tracing* provides a method for physically monitoring the entrance and exit of a procedure.The Print method of the Debug object is one of the most useful of all the built-in debugging aids ofered by Visual Basic.

Assert

The *Assert* method takes an expression that returns a Boolean True or False value and stops execution if the expression is False.The Assert method works only in the development enviroment,but it is very useful.It passes execution only if the expression we pass evaluates to False(0).For example, if variable x has to be trapped when it is less than 0, the following logic is requried.

Example

Private sub Form_Load()

Dim x As integer

x = - 1

Debug.Assert Not (x< 0)

End sub

In the above coding, to create False(0) value for Assert, use the Not operator to invert the truth of the expression to trap a value of X less than 0.

Image List

Introduction

The ImageList control is most often used as a container for images and icons that are employed by other controls, such as TreeView, ListView, TabStrip, and ImageCombo controls. For this reason, it makes sense to describe it before any other controls. The ImageList control is invisible at run time, and to display one of the images it contains you must draw it on a form, a PictureBox control, or an Image control, or associate it with another control.

Using the ImageList control as a repository for images that are then used by other controls offers a number of advantages. Without this control, in fact, you would have to load images from disk at run time using a LoadPicture function, which slows down execution and increases the number of files to be distributed with your program, or an array of Image controls, which slows down form loading. It's much easier and more efficient to load all the images in the ImageList control at design time and then refer to them from the other controls or in source code.

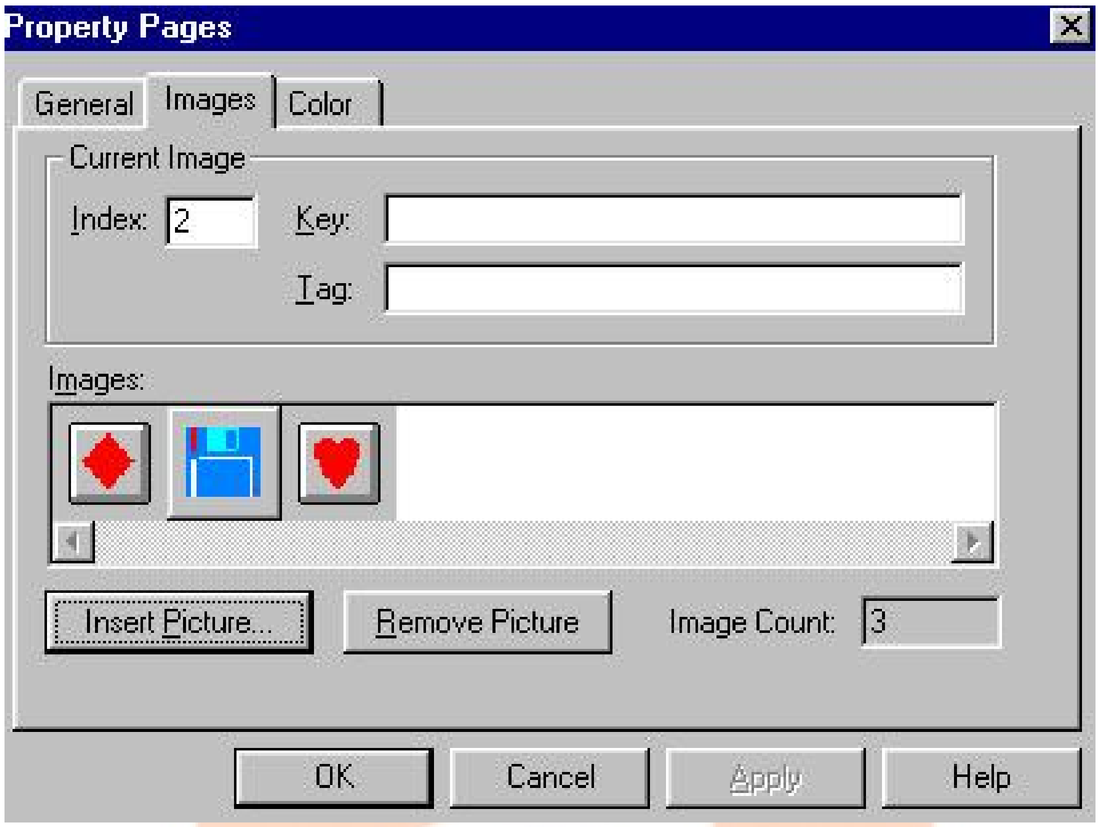
Adding Images

The ImageList control exposes a ListImages collection, which in turn contains a number of ListImage objects. Each ListImage item holds an individual image. As with any collection, an individual ListImage object can be referenced through its numerical index or its string

key (if it has one). Each ListImage object can hold an image in one of the following graphic formats: bitmap (.bmp), icon (.ico), cursor (.cur), JPEG (.jpg), or GIF (.gif). The latter two formats weren't supported by the ImageList control distributed with Visual Basic 5.

Adding images at design time

Adding images at design-time is easy. After you place an ImageList control on a form, right-click on it, select the Properties command from the pop-up menu, and switch to the Images tab, as shown in Figure 10-3. All you have to do now is click on the Insert Picture button and pick up your images from disk. You should associate a string key with each image so that you can refer to it correctly later, even if you add or remove other images in the future (which would affect its numerical index). Of course, all string keys must be unique in the collection. You can also specify a string for the Tag property of an image, for example, if you want to provide a textual description of the image or any other information associated with this image. Visual Basic never directly uses this property, so you're free to store any string data in it.



. *The Images tab of the Properties window of an ImageList control*.

Images added to the ListImages collection can be of any size, with a caveat: If you're going to use these images inside another common control, all the images after the first one will be resized and stretched to reflect the size of the first image added to the control. This isn't an issue if you're going to display these images on a form, a PictureBox control, or an Image control.

If the ImageList control doesn't contain any images, you can set the size you want the images to be in the General tab of the Properties dialog box. Trying to do this when the control already contains one or more ListImage items raises an error.

Adding images at run time

Adding images at run time requires you to use the Add method of the ListImages collection, the syntax of which is the following:

```
Add([Index], [Key], [Picture]) As ListImage
```

If you omit the Index argument, you add the new image at the end of the collection. The following code creates a new ListImage item and associates it with a bitmap loaded from disk

```
Dim li As ListImage

Set li = ImageList1.ListImages.Add(, "Cut", _

LoadPicture("d:\bitmaps\cut.bmp"))
```

You don't need to assign the result value of the Add method to a ListImage object unless you want to assign a string to the Tag property of the object just created. Even in that case, you can do without an explicit variable:

```
With ImageList1.ListImages.Add(, "Cut",
LoadPicture("d:\bitmaps\cut.bmp"))

  .Tag = "The Cut icon"

End With
```

Removing the Images at Run time and Design Time

You can use a numerical index or a string key

To remove the associated image.

```
ImageList1.ListImages.Remove "Cut"
```

You can also remove all the images in one operation by using the collection's Clear method:

Remove all images.

```
ImageList1.ListImages.Clear
```

Extracting individual images

Each ListImage object exposes a Picture property, which lets you extract the image and assign it to another control, typically a PictureBox or Image control:

```
Set Picture1.Picture = ImageList1.ListImages("Cut").Picture
```

ListImage objects also expose an ExtractIcon method, which creates an icon out of the image and returns it to the caller. You can therefore use this method whenever an icon is expected, as in this code:

```
Form1.MouseIcon = ImageList1.ListImages("Pointer").ExtractIcon
```

Creating transparent images

The ImageList control has a MaskColor property whose value determines the color that should be considered transparent when you're performing graphical operations on individual ListImage objects or when you're displaying images inside other controls. By default, this is the gray color (&HC0C0C0), but you can change it both at design time in the Color tab of the Properties dialog box and at run time via code.

When a graphical operation is performed, none of the pixels in the image that are the color defined by MaskColor are transferred. To actually display transparent images, however, you must ensure that the UseMaskColor property is set to True, which is its default value. You can modify this value in the General tab of the Properties dialog box or at run time via code:

```
Make white the transparent color.

ImageList1.MaskColor = vbWhite

ImageList1.UseMaskColor = True
```

Creating composite images

The ImageList control also includes the ability to create composite images by overlaying two individual images held in ListImage objects. This can be accomplished using the Overlay method. Figure 10-4 shows two individual images and then what you can get by overlaying the second one on the first one:

```
PaintPicture ImageList1.ListImages(1).Picture, 0, 10, 64, 64

PaintPicture ImageList1.ListImages(2).Picture, 100, 10, 64, 64

PaintPicture ImageList1.Overlay(1, 2), 200, 10, 64, 64
```

Tree View

Introduction

The Visual Basic 6 version of the TreeView control has a number of improvements and now supports check boxes beside each item and full row selection. Moreover, individual nodes can have different Bold, Foreground, and Background attributes.

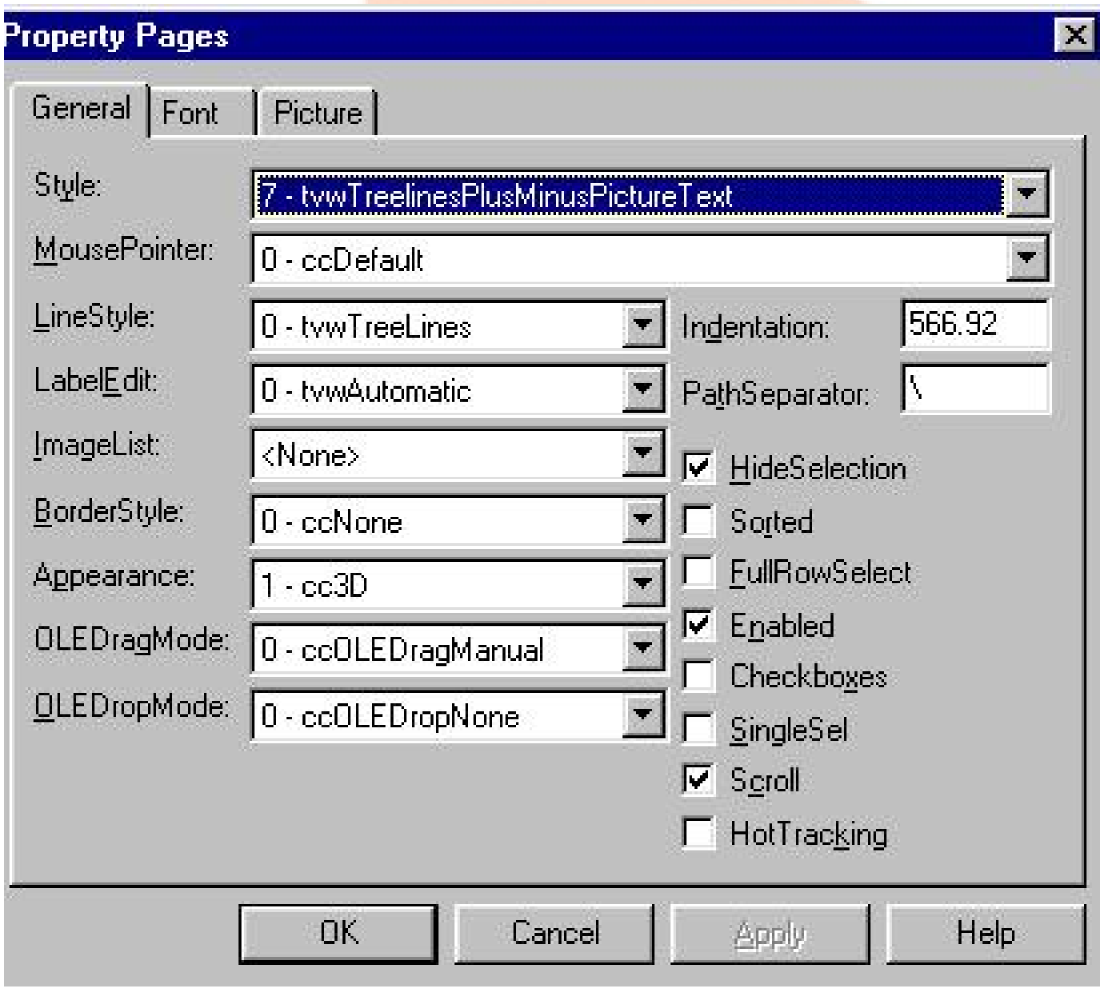
The TreeView control exposes a Nodes collection, which in turn includes all the Node objects that have been added to the control. Each individual Node object exposes a number of properties that let you define the look of the control. Typically, a TreeView control has one single root Node object, but you can also create multiple Node objects at the root level.

Setting Design-Time Properties

Immediately after creating a TreeView control on a form, you should display its Properties dialog box (shown in Figure 10-5), which you do by right-clicking on the control and selecting the Properties menu item. Of course, you can also set properties that appear in this page at run time, but you rarely need to change the appearance of a TreeView control once it has been displayed to the user.

The Style property affects which graphical elements will be used inside the control. A TreeView control can display four graphical elements: the text associated with each Node object, the picture associated with each Node object, a plus or minus sign beside each

Node object (to indicate whether the Node is in collapsed or expanded state), and the lines that go from each Node object to its child objects. The Style property can be assigned one of eight values, each one representing a different combination of these four graphical elements. In most cases, you use the default value, 7-tvwTreelinesPlusMinusPictureText, which displays all graphical elements.



Style: 7 - tvwTreelinesPlusMinusPictureText
MousePointer: 0 - ccDefault
LineStyle: 0 - tvwTreeLines
LabelEdit: 0 - tvwAutomatic
ImageList: <None>
BorderStyle: 0 - ccNone
Appearance: 1 - cc3D
OLEDragMode: 0 - ccOLEDragManual
OLEDropMode: 0 - ccOLEDropNone
Indentation: 566.92
PathSeparator: \
[✓] HideSelection
[ ] Sorted
[ ] FullRowSelect
[✓] Enabled
[ ] Checkboxes
[ ] SingleSel
[✓] Scroll
[ ] HotTracking

The General tab of the Properties dialog box of a TreeView control.

The LineStyle property affects how lines are drawn. The value 0-tvwTreeLines doesn't display lines among root Node objects (this is the default setting), whereas the value 1-tvwRootLines also displays lines among all root Nodes and makes them appear as if they were children of a fictitious Node located at an upper level. The Indentation property states the distance in twips between vertical

dotted lines.

The LabelEdit property affects how the end user can modify the text associated with each Node object. If it's assigned the value 0-tvwAutomatic (the default), the end user can edit the text by clicking on the Node at run time; if it's assigned the value 1-tvwManual, the edit operation can be started only programmatically, by your issuing a StartLabelEdit method.

The ImageList combo box lets you select which ImageList control will be used to retrieve the images of individual Node objects. The combo box lists all the ImageList controls located on the current form.

If you set the Checkboxes property to True, a check box appears beside each Node object so that the end user can select multiple Node objects.

Adding Node objects

One of the shortcomings of the TreeView control is that you can't add items at design time as you can with ListBox and ComboBox controls. You can add Node objects only at run time using the Add method of the Nodes collection.The Add method's syntax is the following:

```
Add([Relative],[Relationship],[Key],[Text],[Image],[SelectedImage]) As Node
```

Relative and Relationship indicate where the new Node should be inserted. Key is its string key in the Nodes collection, Text is the label that will appear in the control, and Image is the index or the string key in the companion ImageList control of the image that will appear beside the Node. SelectedImage is the index or key of the image that will be used when the Node is selected. For example, if you're creating a TreeView control that mimics Windows Explorer

and its directory objects, you might write something like this:

```
Dim nd As Node

Set nd = Add(, , ,"C:\System", "Folder", "OpenFolder
```

You can control the appearance of individual Node objects by setting their ForeColor, BackColor, and Bold properties, the effects of which are shown in Figure 10-6. This new feature permits you to visually convey more information about each Node. Typically, you set these properties when you add an item to the Nodes collection:

```
With TV.Nodes.Add(, , , "New Node")

  .Bold = True

  .ForeColor = vbRed

  .BackColor = vbYellow

End With
```

Showing information about a Node

Users expect the program to do something when they click on a Node object in the TreeView control—for example, to display some information related to that object. To learn when a Node is clicked, you have to trap the NodeClick event. You can determine which Node has been clicked by looking at the Index or Key property of the Node parameter passed to the event procedure. In a typical situation, you store information about a Node in an array of String or UDT items:

```
Private Sub TreeView1_NodeClick(ByVal Node As MSComctlLib.Node)

  ' info() is an array of strings that hold nodes' descriptions.

  lblData.Caption = info(Node.Index)

End Sub
```

The NodeClick event differs from the regular Click event in that the latter fires whenever the user clicks on the TreeView control, whereas the former is activated only when the user clicks on a Node object.

Editing Node text

By default, the user can click on a Node object to enter Edit mode and indirectly change the Node object's Text property. If you don't like this behavior, you can set the LabelEdit property to 1-tvwManual. In this case, you can enter Edit mode only by programmatically executing a StartLabelEdit method.

Regardless of the value of the LabelEdit property, you can trap the instant when the user begins editing the current value of the Text property by writing code in the BeforeLabelEdit event procedure. When this event fires, you can discover which Node is currently selected by using the TreeView's SelectedItem property, and you can cancel the operation by setting the event's Cancel parameter to True:

```
Private Sub TreeView1_BeforeLabelEdit(Cancel As Integer)

  ' Prevent the root Node's Text property from editing.

  If TreeView1.SelectedItem.Key = "Root" Then Cancel = True

End Sub
```

Similarly, you can find out when the user has completed the editing and reject, if you want to, the new value of the Text property by trapping the AfterLabelEdit event. Typically, you use this event to check whether the new value follows any syntactical rule enforced by the particular object. For example, you can reject empty strings by writing the following code:

```
Private Sub TreeView1_AfterLabelEdit(Cancel As Integer, _

  NewString As String)

  If Len(NewString) = 0 Then Cancel = True

End Sub
```

☐ List View

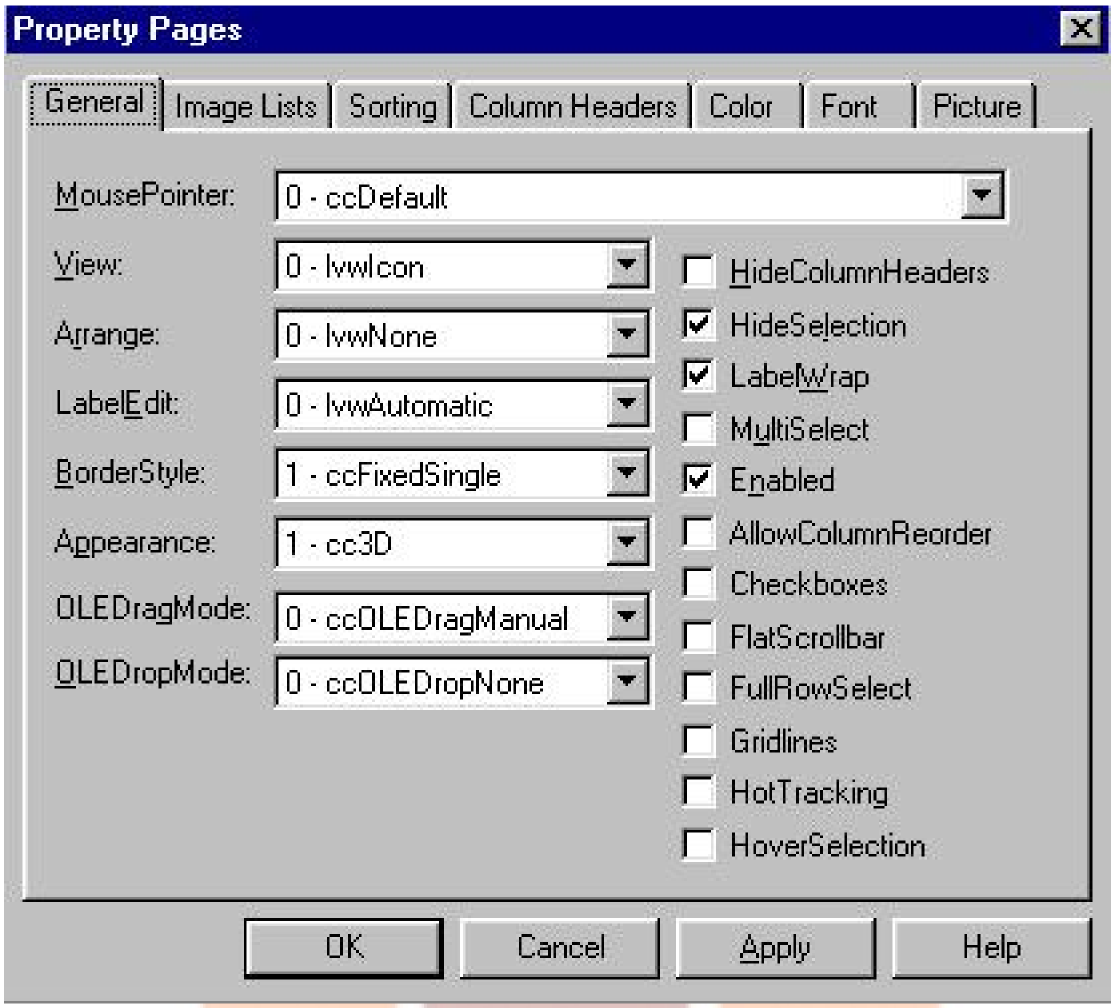
Introduction

Together with the TreeView control, the ListView control has been made popular by Windows Explorer. Now many Windows applications use this pair of controls side by side, and they're therefore called Windows Explorer-like applications. In these applications, the end user selects a Node in the TreeView control on the left and sees some information related to it in the rightmost ListView control.

The ListView control supports four basic view modes: Icon, SmallIcon, List, and Report. To see how each mode is rendered, try the corresponding items in the Windows Explorer View menu. (The Report mode corresponds to the Details menu command.) To give you an idea of the flexibility of this control, you should know that the Windows desktop is nothing but a large ListView control in Icon mode with a transparent background. When used in Report mode, the ListView control resembles a grid control and lets you display well-organized information about each item.

Setting Design-Time Properties

While you can use the regular Properties window to set most properties of a ListView control, it's surely preferable to use a ListView control's custom Properties dialog box.



MousePointer: 0 - ccDefault

View: 0 - lvwIcon

Arrange: 0 - lvwNone

LabelEdit: 0 - lvwAutomatic

BorderStyle: 1 - ccFixedSingle

Appearance: 1 - cc3D

OLEDragMode: 0 - ccOLEDragManual

OLEDropMode: 0 - ccOLEDropNone

- [ ] HideColumnHeaders
- [x] HideSelection
- [x] LabelWrap
- [ ] MultiSelect
- [x] Enabled
- [ ] AllowColumnReorder
- [ ] Checkboxes
- [ ] FlatScrollbar
- [ ] FullRowSelect
- [ ] Gridlines
- [ ] HotTracking
- [ ] HoverSelection

The General tab of the Properties dialog box of a ListView control.

General properties

I've already referred to the View property, which can be one of the following values: 0-lvwIcon, 1-lvwSmallIcon, 2-lvwList, or 3-lvwReport. You can change this property at run time as well as let the user change it (typically by offering four options in the View menu of your application). The Arrange property lets you decide whether icons are automatically aligned to the left of the control (1-lvwAutoLeft) or to the top of the control (2-lvwAutoTop), or whether they shouldn't be aligned at all (0-lvwNone, the default behavior). This property takes effect only when the control is in Icon or SmallIcon display mode.

The LabelEdit property determines whether the user can edit the text

associated with an item in the control. If this property is set to 0-lvwAutomatic, the edit operation can be initiated only by code using a StartLabelEdit method. The LabelWrap Boolean property specifies whether longer labels wrap on multiple lines of text when in Icon mode. The HideColumnHeaders Boolean property determines whether column headers are visible when in Report mode. (The default value is False, which makes the columns visible.) If you assign the MultiSelect property the True value, the ListView control behaves much like a ListBox control whose MultiSelect property has been set to 2-Extended

Column Header

The ColumnHeaders property is new to Visual Basic 6 because previous versions of the ListView control didn't support icons in column headers:

' You can use the same ImageList control for different properties.

```
Set ListView1.Icons = ImageList1

Set ListView1.SmallIcons = ImageList2

Set ListView1.ColumnHeaderIcons = ImageList2
```

You can automatically sort the items in the ListView control by setting a few properties in the Sorting tab of the Properties dialog box. Set the Sorted property to True if you want to sort items. SortKey is the index of the column that will be used for sorting (0 for the first column), and SortOrder is the sorting order (0-lvwAscending or 1-lvwDescending). You can also set these properties at run time.

You can create one or more ColumnHeader objects at design time by using the Column Header tab of the Properties dialog box. You just have to click on the Insert Column button and then type the values of the Text property (which will be displayed in the header), the Alignment property (Left, Right, or Center, although the first column header can only be left-aligned), and the Width in twips. You can also specify a value for the Key and Tag properties and set the index of the icon to be used for this header. (It's an index referred to by the ColumnHeaderIcons property in the ImageList control, or it's 0 if this column header doesn't have any icons

Adding ListItem objects

You add new items to the ListView controls with the ListItems collection's Add method, which has the following

```
Add([Index], [Key], [Text], [Icon], [SmallIcon]) As ListItem
```

Index is the position at which you place the new item. (If you omit Index, the item is added to the end of the collection.) Key is the inserted item's optional key in the ListItems collection, Text is the string displayed in the control, Icon is an index or a key in the ImageList control pointed to by the Icons property, and SmallIcon is an index or a key in the ImageList control pointed to by the SmallIcons property. All these arguments are optional.

The Add method returns a reference to the ListItem object being created, which you can use to set those properties whose values can't be passed to the Add method itself, as in the following example:

Adding ListItem objects

You add new items to the ListView controls with the ListItems collection's Add method, which has the following syntax:

```
Add([Index], [Key], [Text], [Icon], [SmallIcon]) As ListItem
```

Index is the position at which you place the new item. (If you omit Index, the item is added to the end of the collection.) Key is the inserted item's optional key in the ListItems collection, Text is the string displayed in the control, Icon is an index or a key in the ImageList control pointed to by the Icons property, and SmallIcon is an index or a key in the ImageList control pointed to by the SmallIcons property. All these arguments are optional.

The Add method returns a reference to the ListItem object being created, which you can use to set those properties whose values can't be passed to the Add method itself, as in the following example:

```
Create a new item with a "ghosted" appearance.

Dim li As ListItem

Set li = ListView1.ListItems.Add(, , "First item", 1)

li.Ghosted = True
```

Adding ColumnHeaders objects

Often you don't know at design time what columns should be displayed in a ListView control. For example, you might be showing the result of a user-defined query, in which case you don't know the number and the names of the fields involved. In such circumstances, you must create ColumnHeader objects at run time with the Add method of the ColumnHeaders collection, which has this syntax:

```
Add([Index], [Key], [Text], [Width], [Alignment], [Icon]) _

  As ColumnHeader
```

Index is the position in the collection, Key is an optional key, Text is the string displayed in the header, and Width is the column's width in twips. Alignment is one of the following constants: 0-lvwColumnLeft, 1-lvwColumnRight, or 2-lvwColumnCenter. Icon is an index or a key in the ListImage control referenced by the ColumnHeaderIcons property. With the exception of the Tag property, these are the only properties that can be assigned when a ColumnHeader object is created, so you can usually discard the return value of the Add method:

Clear any existing column header.

```
ListView1.ColumnHeaders.Clear

'The alignment for the first column header must be lvwColumnLeft.

ListView1.ColumnHeaders.Add , , "Last Name", 2000, lvwColumnLeft

ListView1.ColumnHeaders.Add , , "First Name", 2000, lvwColumnLeft

ListView1.ColumnHeaders.Add , , "Salary", 1500, lvwColumnRight
```

Adding ListSubItems

Each ListItem object supports a ListSubItems collection, which lets you create values displayed in the same row as the main ListItem object when the control is in Report mode. This collection replaces the SubItems array that was present in previous versions of the control. (The array is still supported for backward compatibility.) You

can create new ListSubItem objects using the Add method of the ListSubItems collection:

```
Add([Index], [Key], [Text], [ReportIcon], [ToolTipText]) _

  As ListSubItem
```

Flex Grid

Introduction

A MSFlexgrid control in Visual Basic is used to create applications that present information in rows and columns. It diaplays information in cells. A cell is a location in the MSFlexGrid at which a row and a column intersect.The user can select a cell at run time by clicking it or by using the arrow keys, but cannot edit or alter the cell's contents.

Use of FlexGrid Control

The MSFlexGrid control displays and operates on tabular data.It allows complete flexibility to sort,merge, and format tables containing strings and pictures.When bound to a Data control,MSFlexGrid displays read_only data.

We can place text or a picture, or both in any cell of a MSFlexGrid.The Row and Col properties specify the current cell in a MSFlexGrid.We can specify the curent cell in code, or the user can change it at run time using the mouse or the arrow keys.The Text property refrences the contents of the current cell.

If a cell's text is too long to be displayed in the cell, and the WordWrap property is set to True,the text wraps to the next line within the same cell.To display the wrapped text,we need to increase the cell's column width(Col Width property) or row heigth (Row

Heightproperty)

The Cols and Rows properties are used to determine the number of columns and row in a MSFlexGrid control.

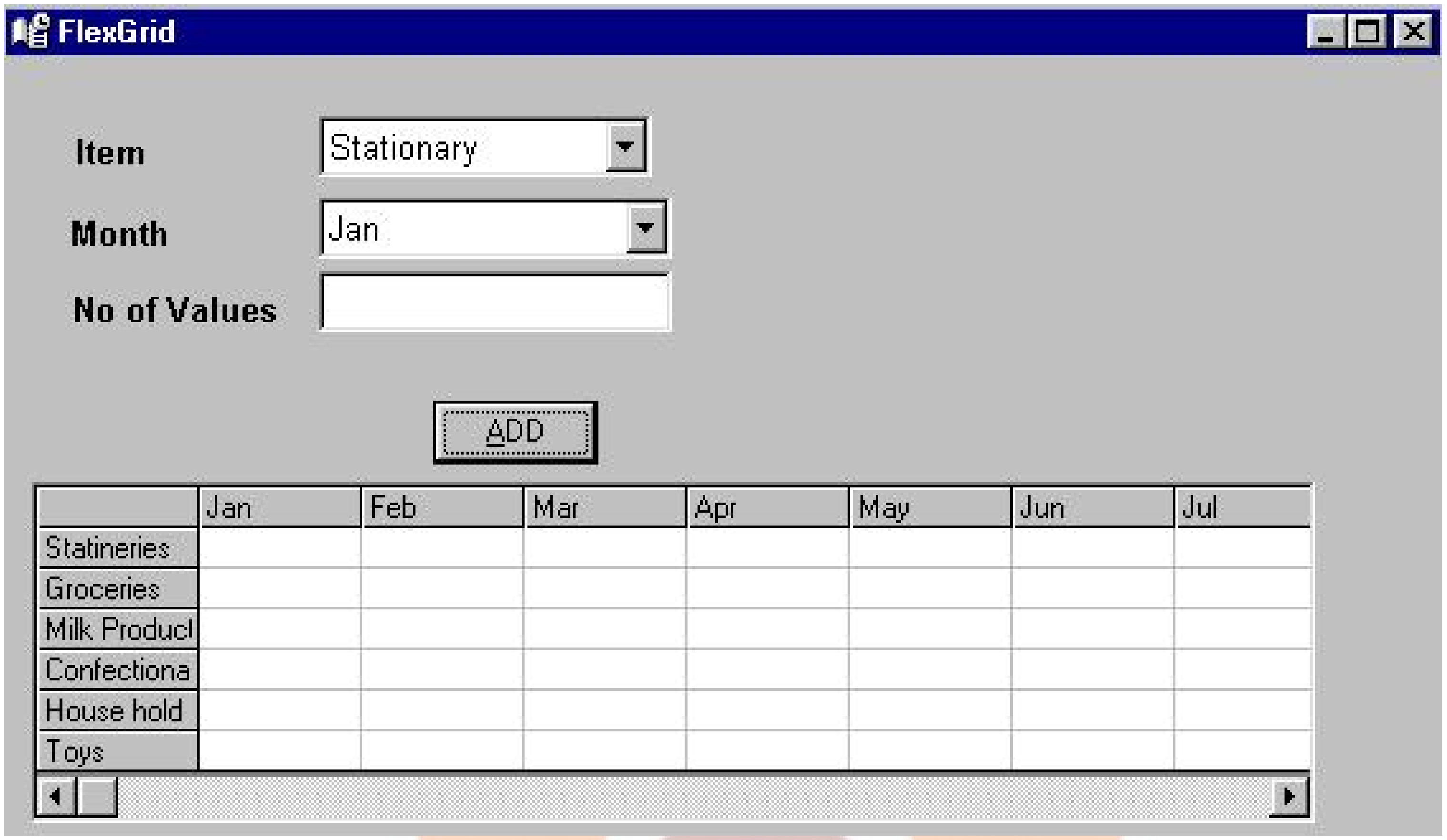
Different Types of Rows and Columns in FlexGrid

Two kinds of rows or columns are created in the MSFlexGrid control.They are fixed and nonfixed.A nonfixed row or column scrolls when the scroll bars are active in the MSFlexGrid control.A fixed row or column does not scroll at any time.FixedRows or FixedCols is generally used for displaying headings.Rows and columns are created by setting the four properties of MSFlexgrid control such as ROws,Cols,FixedRows and FixedCols. Lets us design a small program that uses the MSFlexGrid.

Since the MSFlexGrid control is an OCX control, we must make sure if the control is included in the projects.If the control does not appear in the ToolBox, it is added by selecting Components from Project menu and placing a check mark in the Microsoft MSFlexGrid Control.This places the control in the ToolBox.

Example

Screen :



Item	Stationary						
Month	Jan						
No of Values							

	Jan	Feb	Mar	Apr	May	Jun	Jul
Statineries							
Groceries							
Milk Product							
Confectiona							
House hold							
Toys							

Table :

Sr no	Object Name	Type	Properties
1	cboItem	Combo box	.style=2-Dropdown List
2	cboMonth	Combo box	.style=2-Dropdown List
3	txtSale	Text Box	.FontSize=10
4	Lable1	Lable	.Caption=Item .Autosize=True

5	Lable2	Lable	.Caption=Month .AutoSize=True
6	Lable3	Lable	Caption=No of Values .AutoSize=True
7	flxData	MSFlexGrid	Default Setting

Coding :

```
        Option Explicit

        Dim Mon(12) As String

        Dim Item(6) As String

        Dim i As Integer
```

```
Private Sub Command1_Click()

        flxData.Row = cboItem.ListIndex + 1

        flxData.Col = cboMonth.ListIndex + 1

        flxData.Text = Str(Val(flxData.Text) + Val(txtSale.Text))

End Sub
```

```
Private Sub Form_Load()

'---generate columns and rows

         flxData.Cols = 13

         flxData.Rows = 7

'---assign col/row values

       flxData.Col = 0 'on column 0



       flxData.Row = 1

       flxData.Text = "Statineries"

       flxData.Row = 2

       flxData.Text = "Groceries"

       flxData.Row = 3

       flxData.Text = "Milk Products"

       flxData.Row = 4

       flxData.Text = "Confectionaries"

       flxData.Row = 5

       flxData.Text = "House hold item"

       flxData.Row = 6
       flxData.Text = "Toys"



       flxData.Row = 0

       flxData.Col = 1

       flxData.Text = "Jan"

       flxData.Col = 2

       flxData.Text = "Feb"

       flxData.Col = 3

       flxData.Text = "Mar"

       flxData.Col = 4

       flxData.Text = "Apr"

       flxData.Col = 5

       flxData.Text = "May"

       flxData.Col = 6

       flxData.Text = "Jun"

       flxData.Col = 7

       flxData.Text = "Jul"

       flxData.Col = 8
```

```
        flxData.Text = "Aug"

        flxData.Col = 9

        flxData.Text = "Sep"

        flxData.Col = 10

        flxData.Text = "Oct"

        flxData.Col = 11

        flxData.Text = "Nov"

        flxData.Col = 12

        flxData.Text = "Dec"

'---- For Month

        Mon(0) = "Jan"

        Mon(1) = "Feb"

        Mon(2) = "Mar"

        Mon(3) = "Apr"

        Mon(4) = "May"

        Mon(5) = "Jun"

        Mon(6) = "Jul"

        Mon(7) = "Aug"
Mon(8) = "Sep"

Mon(9) = "Oct"

Mon(10) = "Nov"

Mon(11) = "Dec"



For i = 0 To 11

  cboMonth.AddItem Mon(i)

Next

cboMonth.ListIndex = 0




Item(0) = "Stationary"

Item(1) = "Groceries"

Item(2) = "Milk Products"

Item(3) = "Confectionaries"

Item(4) = "House hold item"

Item(5) = "Toys"
```

```
    For i = 0 To 5
      cboItem.AddItem Item(i)
    Next

    cboItem.ListIndex = 0

End Sub
```

Changing the Cell Eidth and Cell Height

The heigth and width of the cells in the FlexGrid can be widened in order to get a clear picture at run time. A procedure is added to the Form by selecting the Add Procedure command from the project menu in the Code Window.The new procedure is named as SetColWidth.

Scroll Bars of the FlexGris Control

Visual Basic automatically adds horizontal and veritcal scrol bars in the FlexGrid control when the cells do not fit into it. This is because, the default values of the Scrollbars property of the Flexgrid control is set to 3-Both.If we do not want the ScrollBars to appear, the ScrollBars property is set to 0-None at design time or the following

code is added in the Form_load() procedure.

FlexGrid1.ScrollBars =0

Rich Text Box

Introduction

The RichTextBox control is one of the most powerful controls provided with Visual Basic. In a nutshell, it's a text box that's able to display text stored in Rich Text Format (RTF), a standard format recognized by virtually all word processors, including Microsoft WordPad (not surprisingly, since WordPad internally uses the RichTextBox control). This control supports multiple fonts and colors, left and right margins, bulleted lists, and more.

You might need time to get used to the many features of the RichTextBox control. The good news is that the RichTextBox control is code-compatible with a regular multiline TextBox control, so you can often recycle code that you have written for a TextBox control. But unlike the standard TextBox control, the RichTextBox control has no practical limit to the number of lines of text it can contain.

The RichTextBox control is embedded in the RichTx 32.ocx file, which you must distribute with all the applications that use this control.

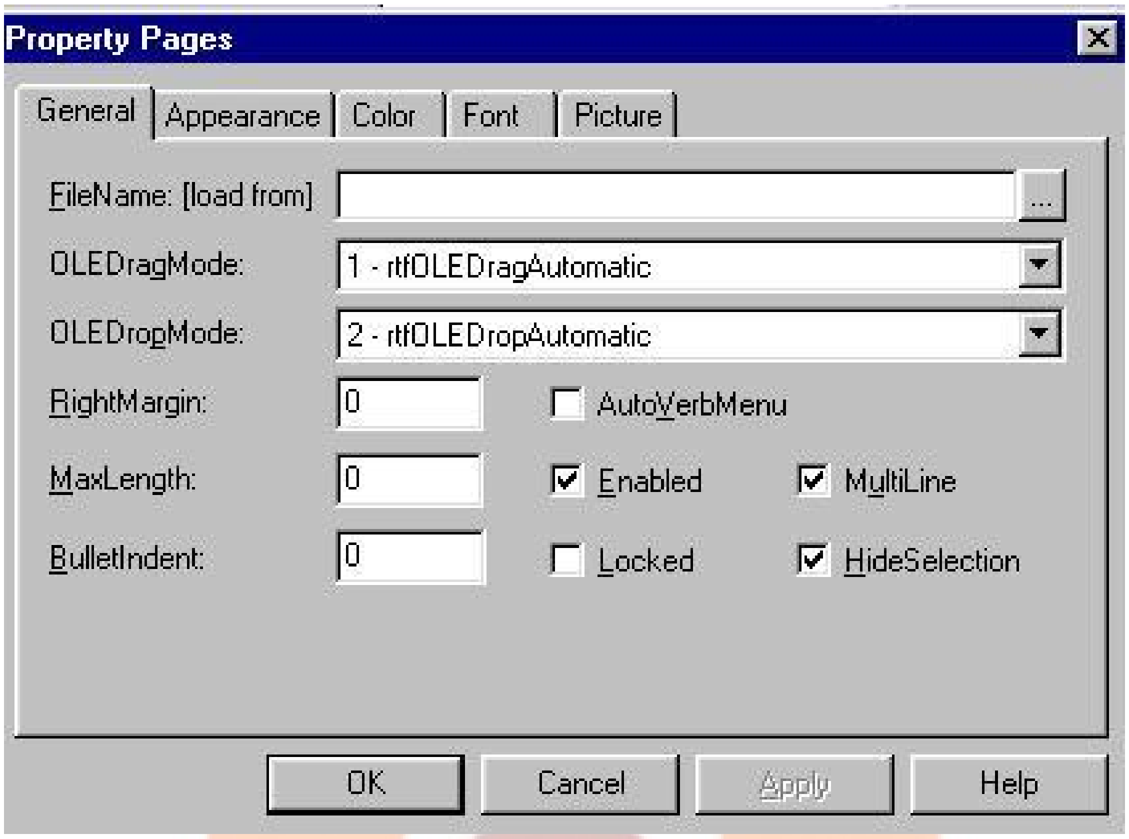
Setting Design-Time Properties

You can set a few useful design-time properties in the General tab of the Property Pages dialog box as you can see in Figure 12-8. For example, you can type the name of a TXT or RTF file that must be loaded when the form is loaded and that corresponds to the Filename property.

The RightMargin property represents the distance in twips of the right margin from the left border of the control. The BulletIndent is the number of twips a paragraph is indented when the SetBullet property is True. The AutoVerbMenu is an interesting property that lets you prevent the standard Edit pop-up menu from appearing when the user right-clicks on the control. If you want to display your own pop-up menu, leave this property as False. All the other properties in this General page are also supported by standard TextBox controls, so I won't describe them here.

In the Appearance tab of the Properties dialog box, you find other properties, such as BorderStyle and ScrollBars, whose meaning should already be known to you. An exception is the DisableNoScroll property: When the ScrollBars property is assigned a value other than 0-rtfNone and you set the DisableNoScroll property to True, the RichTextBox control will always display the scroll bars, even if the current document is so short that it doesn't require scrolling. This is consistent with the behavior of most word processors.

The RichTextBox control is data-aware and therefore exposes the usual Dataxxxx properties that let you bind the control to a data source. In other words, you can write entire TXT or RTF documents in a single field of a database.



FileName: [load from]
OLEDragMode: 1 - rtfOLEDragAutomatic
OLEDropMode: 2 - rtfOLEDropAutomatic
RightMargin: 0
MaxLength: 0
BulletIndent: 0
[ ] AutoVerbMenu
[✓] Enabled
[ ] Locked
[✓] MultiLine
[✓] HideSelection

The General tab of the Properties dialog box of a RichTextBox control.

Run-Time Operations

The RichTextBox control exposes so many properties and methods that it makes sense to subdivide them in groups, according to the action you want to perform.

Loading and saving files

You can load a text file into the control using the LoadFile method, which expects the filename and an optional argument that specifies whether the file is in RTF format (0-rtfRTF, the default) or plain text (1-rtfText):

```
Load an RTF file into the control.

RichTextBox1.LoadFile "c:\Docs\TryMe.Rtf", rtfRTF
```

The name of the file loaded by this method becomes available in the FileName property. You can also indirectly load a file into the control by assigning its name to the FileName property, but in this case you have no way of specifying the format.

You can save the current contents of the control using the SaveFile method, which has a similar syntax:

```
Save the text back into the RTF file.

RichTextBox1.SaveFile RichTextBox1.FileName, rtfRTF
```

The LoadFile and SaveFile methods are a good solution when you want to load or save the entire contents of a file. At times, however, you might want to append the contents of the control to an existing file or store multiple portions of text in the same file. In such cases, you can use the TextRTF property with regular Visual Basic file commands and functions:

Changing character attributes

The RichTextBox control exposes many properties that affect the attributes of the characters in the selected text: These are SelFontName, SelFontSize, SelColor, SelBold, SelItalic, SelUnderline, and SelStrikeThru. Their names are self-explanatory, so I won't describe what each one does. You might find it interesting to note that all of the properties work as they would within a regular word processor. If text is currently selected, the properties set or return the corresponding attributes; if no text is currently selected, they set or return the attributes that are active from the insertion point onward.

The control also exposes a Font property and all the various

Fontxxxx properties, but these properties affect the attributes only when the control is loaded. If you want to change the attribute of the entire document, you must select the whole document first:

```
Change font name and size of entire contents.

RichTextBox1.SelStart = 0

RichTextBox1.SelLength = Len(RichTextBox1.Text)

RichTextBox1.SelFontName = "Times New Roman"

RichTextBox1.SelFontSize = 12

' Cancel the selection.

RichTextBox1.SelLength = 0
```

Changing paragraph attributes

You can control the formatting of all the paragraphs that are included in the current selection. The SelIndent and SelHangingIndent properties work together to define the left indentation of the first line and all the following lines of a paragraph. The way these properties work differs from how word processors usually define these sorts of entities: The SelIndent property is the distance (in twips) of the first line of the paragraph from the left border, whereas the SelHangingIndent property is the indentation of all the following lines relative to the indentation of the first line. For example, this is the code that you must execute to have a paragraph that is indented by 400 twips and whose first line is indented by an additional 200 twips:

```
RichTextBox1.SelIndent = 600 ' Left indentation + 1st line indentation

RichTextBox1.SelHangingIndent = -200 ' A negative value
```

The SelRightIndent property is the distance of the paragraph from the right margin of the document (whose position depends on the RightMargin property). The following code moves the right margin about 300 twips from the right border of the control, and then sets a right indentation of 100 twips for the paragraphs that are currently selected:

```
' RightMargin is measured from the left border.

RichTextBox1.RightMargin = RichTextBox1.Width _ 300

RichTextBox1.SelRightIndent = 100
```

You can control the alignment of a paragraph by means of the SelAlignment enumerated property, which can be assigned the values 0-rtfLeft, 1-rtfRight, or 2-rtfCenter. (The RichTextBox control doesn't support justified paragraphs.) You can read this property to retrieve the alignment state of all the paragraphs in the selection: In this case, the property returns Null if the paragraphs have different alignments.

The SelCharOffset property lets you create superscript and subscript text—in other words, position characters slightly above or below the text baseline. A positive value for this property creates a superscript, a negative value creates a subscript, and a zero value restores the regular text position. You shouldn't assign this property large positive or negative values, though, because they would make the superscript or subscript text unreadable (or even invisible)—the RichTextBox control doesn't automatically adjust the distance

between lines if they contain superscript or subscript text:

'Make the selection superscript text.

RichTextBox1.SelCharOffset = 40

' Don't forget to reduce the characters' size.

RichTextBox1.SelFontSize = RichTextBox1.SelFontSize \ 2

Managing the Tab key

Like a real word processor, the RichTextBox control is capable of managing tab positions on a paragraph-by-paragraph basis. This is achieved using the two properties SelTabCount and SelTabs: The former sets the number of tab positions in the paragraphs included in the selection, and the latter sets each tab position to a given value. Here's a simple example that shows how you can use these properties:

```
Add three tabs, at 300, 600, and 1200 twips from left margin.

RichTextBox1.SelTabCount = 3

' The SelTabs property is zero-based.

' Tabs must be specified in increasing order, otherwise they are ignored.

RichTextBox1.SelTabs(0) = 300

RichTextBox1.SelTabs(1) = 600

RichTextBox1.SelTabs(2) = 1200
```

Status Bar

Introduction

Many applications employ the bottom portion of their windows for displaying information to the end user. The most convenient way to create this interface in Visual Basic is with a StatusBar control.

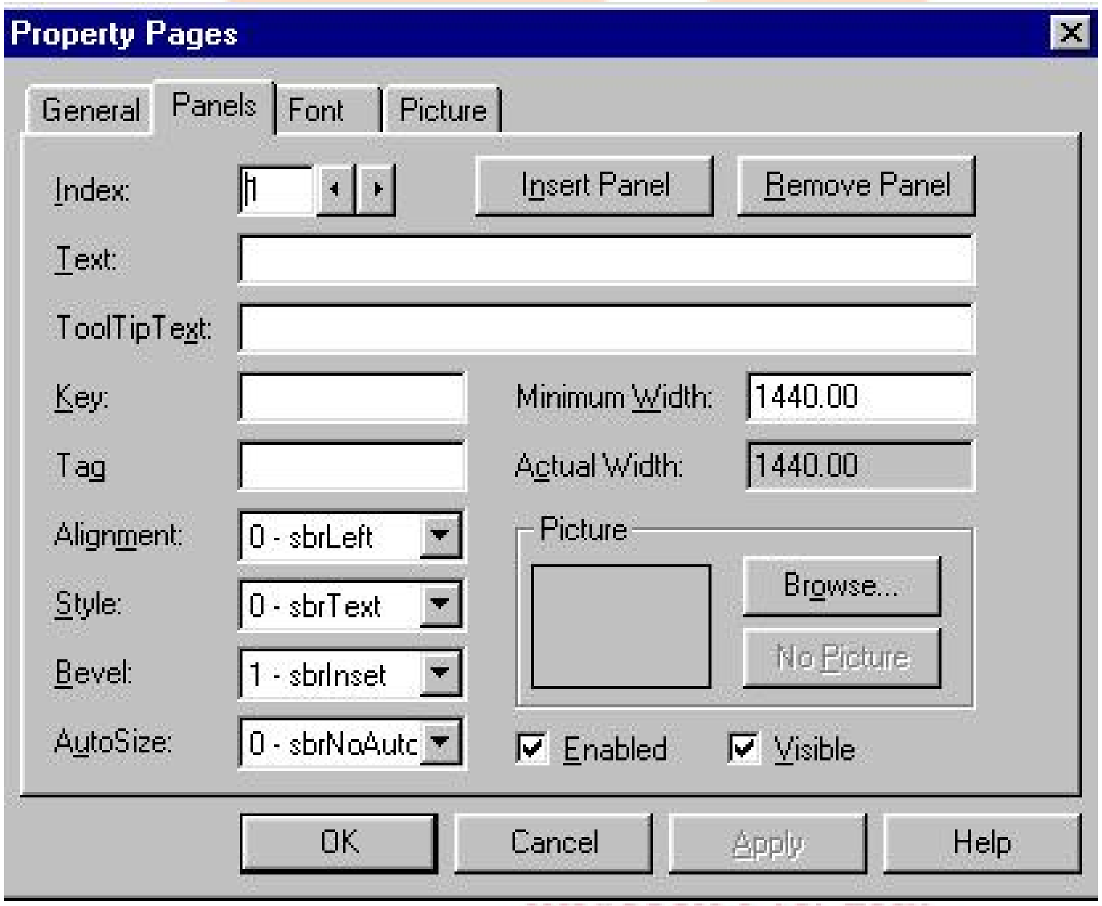
The StatusBar control exposes a Panels collection, which in turn contains Panel objects. A Panel object is an area of the status bar that can hold a piece of information in a given style. The StatusBar control offers several automatic styles (such as date, time, and state of shift keys), plus a generic Text style that lets you show any string in a Panel object. You can also have a StatusBar control work in SimpleText mode, whereby individual Panel objects are replaced by a wider area in which you can display long text messages.

Setting Design-Time Properties

The General tab of the Properties dialog box doesn't contain many interesting properties. In theory, you can set the Style property to 0-sbrNormal (the default) or 1-sbrSimpleText, and you can specify the SimpleText property itself, which will therefore appear as is in the StatusBar. In practice, however, you never change the default settings because you rarely need to create a StatusBar control merely to show a text message. In that case, in fact, you'd be better off with a simpler Label control or a PictureBox control with Align = vbAlignBottom. The only other custom property that appears on this tab is ShowTips, which enables the ToolTipText property of

individual Panel objects.

Move on to the Panels tab of the Property Pages dialog box to create one or more Panel objects, as shown in Figure 10-22. Each Panel object has a number of properties that finely determine its appearance and behavior. The most interesting property is Style, which affects what's shown inside the Panel. The default value is 0-sbrText, which displays the string assigned to the Text property. You can use a Panel object as an indicator of the status of a particular shift key using one of the settings 1-sbrCaps, 2-sbrNum, 3-sbrIns, or 4-sbrScrl. You can also automatically display the current time or date using the 5-sbrTime or 6-sbrDate setting.



Index: 1
Text:
ToolTipText:
Key:
Tag
Minimum Width: 1440.00
Actual Width: 1440.00
Alignment: 0 - sbrLeft
Style: 0 - sbrText
Bevel: 1 - sbrInset
AutoSize: 0 - sbrNoAutc
[✓] Enabled
[✓] Visible

The Panels tab of the Property Pages dialog box of a StatusBar control.

As I mentioned previously, the Text property is the string that appears in the Panel object when Style is sbrText. Key is the optional key that identifies a Panel object in the Panels collection; Tag and

ToolTipText have the usual meanings. The Alignment property determines the position of the Panel's contents (0-sbrLeft, 1-sbrCenter, or 2-sbrRight). The Bevel property affects the type of border drawn around the Panel: Its default value is 1-sbrInset, but you can change it to 2-sbrRaised or opt to have no 3-D border with 0-sbrNoBevel.

The MinWidth property is the initial size of the Panel object in twips. The AutoSize property affects the behavior of the Panel object when the form is resized: 0-sbrNoAutoSize creates a fixed-size Panel. The setting 1-sbrSpring is for Panels that resize with the parent form. (When there are multiple Panels with this setting, all of them shrink or expand accordingly.) The setting 2-sbrContents is for Panels whose widths are determined by their contents.

You can display an icon or a bitmap inside a Panel. At design time, you do this by loading an image from disk. Note that this is an exception among common controls, which usually refer to images by way of a companion ImageList control. The reason for this practice is that you might want to load images of different sizes in each Panel, whereas an ImageList control can contain images only of the same width and height.

Run-Time Operations

You won't want to perform many operations on a StatusBar control at run time. But you might need to change the Text property of a given Panel object whose Style property is 0-sbrText, as in the following example:

```
Display a message in the third panel.

StatusBar1.Panels(3).Text = "Hello World!"
```

For longer messages, you can change the Style property of the StatusBar control and assign a string to its SimpleText property:

```
' Display a message in the entire status bar.

StatusBar1.Style = sbrSimple

StatusBar1.SimpleText = "Saving data to file..."

' A lengthy operation

' ...

' Remember to restore the Style property.

StatusBar1.Style = sbrText
```

Creating and removing Panel objects

You rarely create and destroy Panel objects at run time, but it's good to know that you can do it if you really need to. To accomplish this, use the Add method of the Panels collection, which has the following syntax:

```
Add([Index], [Key], [Text], [Style], [Picture]) As Panel
```

where each argument corresponds to a property of the Panel object being created. For example, this code creates a new Panel in the leftmost position in the StatusBar control:

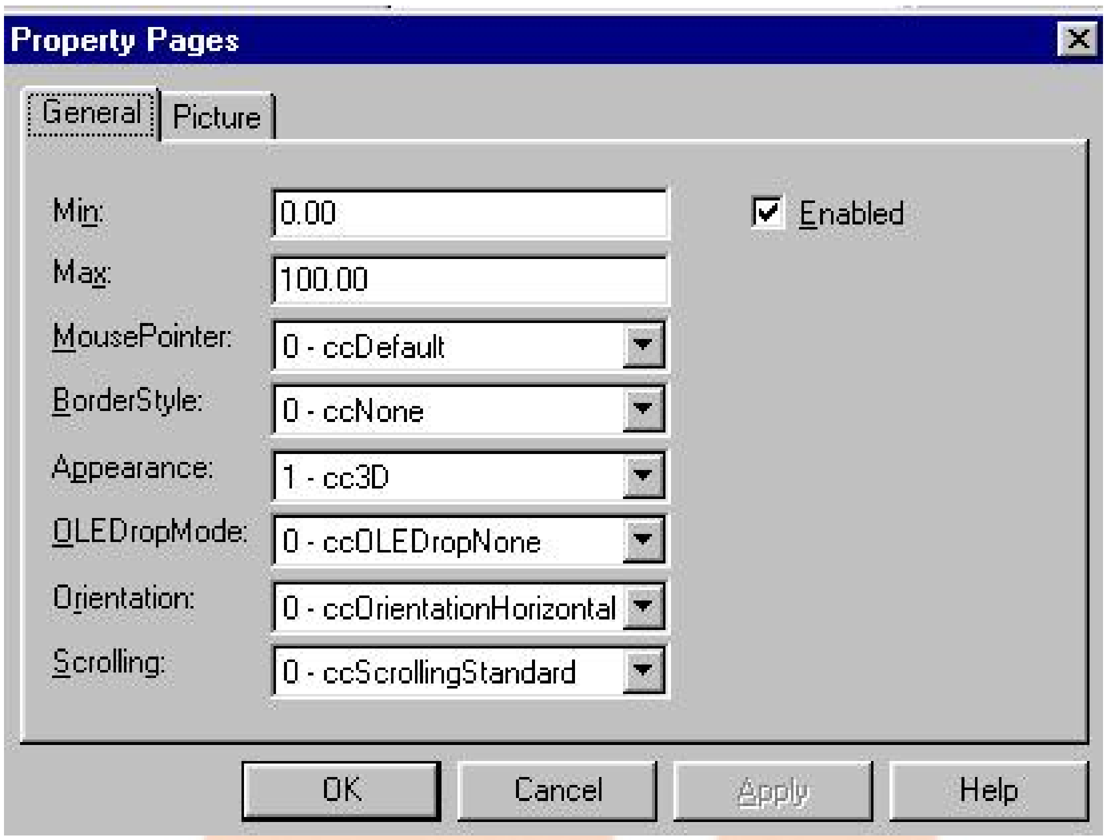
Progress Bar

Introduction

The ProgressBar control is used to inform the user about the progress state of a lengthy operation. This control is the simplest one among those contained in the MsComCtl.OCX file because it doesn't have any dependent objects and it doesn't expose any custom events.

Setting Design-Time Properties

You have to set up a few properties at design time after you drop a ProgressBar control on a form; in most cases, you can accept the default values. The most important properties are Min and Max, which determine the minimum and maximum values that can be displayed by the progress bar.


Property Pages

General | Picture

Min: 0.00

Max: 100.00

MousePointer: 0 - ccDefault

BorderStyle: 0 - ccNone

Appearance: 1 - cc3D

OLEDropMode: 0 - ccOLEDropNone

Orientation: 0 - ccOrientationHorizontal

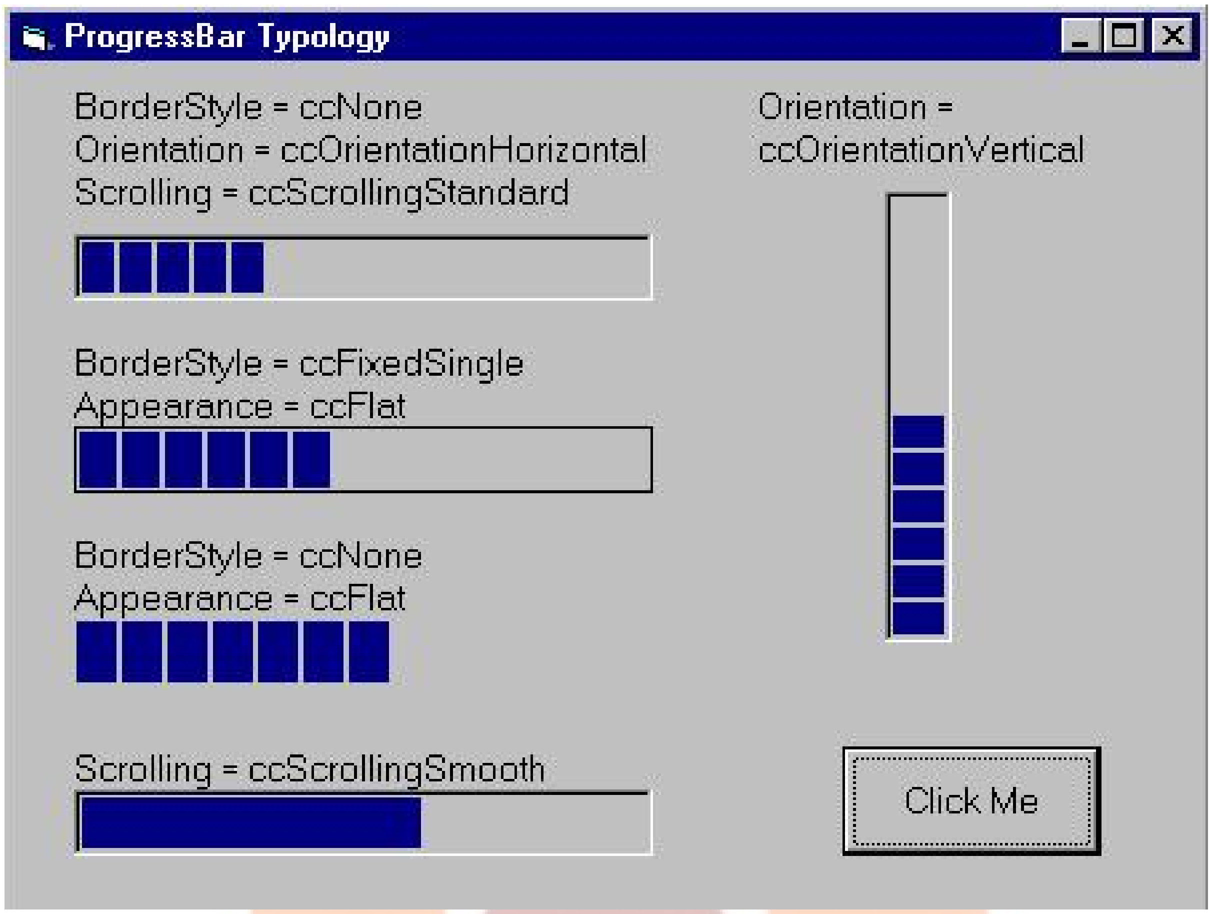
Scrolling: 0 - ccScrollingStandard

☑ Enabled

The ProgressBar control that comes with Visual Basic 6 includes two new properties, Orientation and Scrolling. The former lets you create vertical progress bars; the latter lets you alternate between a standard segmented bar and a smoother bar, as you can see in Figure 10-24. You can change these values even at run time.

Run-Time Operations

There isn't much to say about run-time interaction with the ProgressBar control. In practice, the only thing you can do through code is set the Value property to a number in the range from Min to Max. Any value outside this interval fires an error 380 "Invalid property value." As I mentioned previously, the ProgressBar control doesn't expose any custom events.


ProgressBar Typology
BorderStyle = ccNone
Orientation = ccOrientationHorizontal
Scrolling = ccScrollingStandard
BorderStyle = ccFixedSingle
Appearance = ccFlat
BorderStyle = ccNone
Appearance = ccFlat
Scrolling = ccScrollingSmooth
Orientation = ccOrientationVertical
Click Me

***The effects of the Orientation, Scrolling, Appearance, and BorderStyle properties on the ProgressBar control*.**

Only two other properties affect the aspect of the control—Appearance and BorderStyle.Shows a number of possible combinations of these properties.

◘ Tool Bar

Introduction

The majority of Windows applications include one or more toolbars, which offer the end user the convenience of executing the most common commands with a click of the mouse. Toolbars should never replace menus—and for good reason: menus can be operated with the keyboard; toolbars can't—but they surely make a program more usable and give it a modern look and feel.

Visual Basic comes with a Toolbar control that can contain buttons and other controls and that can be interactively customized by the end user. The Visual Basic 6 version adds the flat style made popular by Microsoft Internet Explorer and the support for building drop-down menus.

The Toolbar control exposes the Buttons collection, which in turn contains Button objects. Each Button object can be an actual push button, a separator, or a placeholder for another control placed on the toolbar (typically a TextBox control or a ComboBox control). In addition, a Button object exposes the ButtonsMenus collection, where each ButtonMenu object is an item of a drop-down menu. (If the Button object isn't a drop-down menu, this collection is empty.)

Setting Design-Time Properties

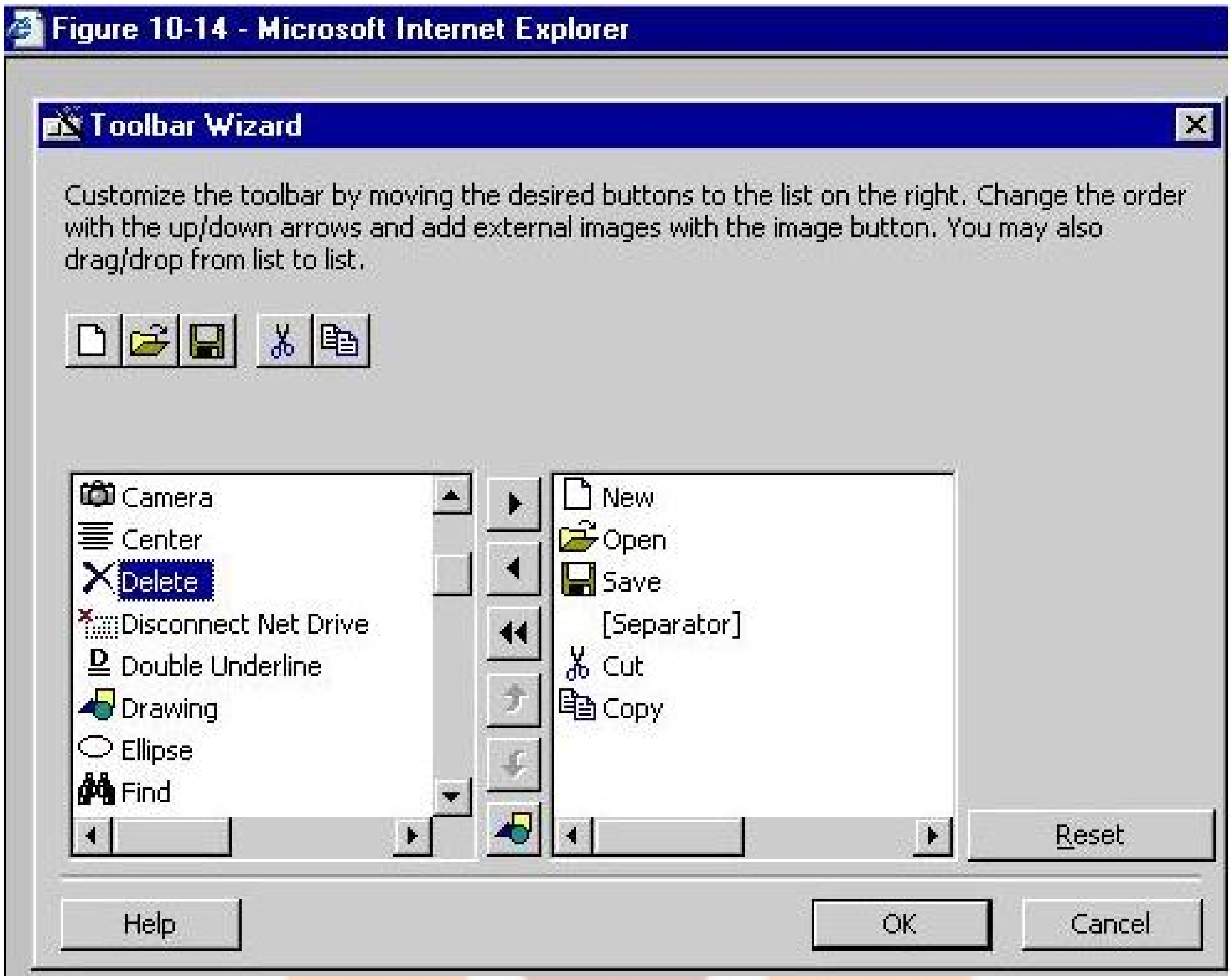
In most cases, you define the appearance of a Toolbar at design time

and then simply react to user's clicks on its buttons. You have two ways to work with a Toolbar at design time: by using the Toolbar Wizard or by manually setting properties. The two methods aren't mutually exclusive: In most cases, in fact, you might find it convenient to create a first version of a Toolbar control using the wizard and then refine it in its Properties dialog box.

The Toolbar Wizard

The Toolbar Wizard is a new add-in provided with Visual Basic 6. But you won't find this wizard in the list of installable add-ins in the Add-In Manager dialog box. Instead, you have to install the Application Wizard add-in: After you do this, you'll find the Toolbar Wizard command in the Add-In menu. If you select this command, the wizard adds a new Toolbar control to the current form and lets you customize it. Or you can place a Toolbar control on the form yourself, and the wizard will be automatically activated.

Using the Toolbar Wizard is simple. You have a list of buttons in the leftmost list box (see Figure 10-14) from which you select the buttons you want to add to the Toolbar control. You can move items between the two list boxes and change their order in the toolbar by using the provided push buttons, or you can use drag-and-drop. The wizard also creates the companion ImageList control on the form. When you complete a toolbar, you'll be asked whether you want to save it in an .rwp profile file, which lets you speed up the creation of similar toolbars in future applications.



Customize the toolbar by moving the desired buttons to the list on the right. Change the order with the up/down arrows and add external images with the image button. You may also drag/drop from list to list.

. Creating a toolbar using the Toolbar Wizard.

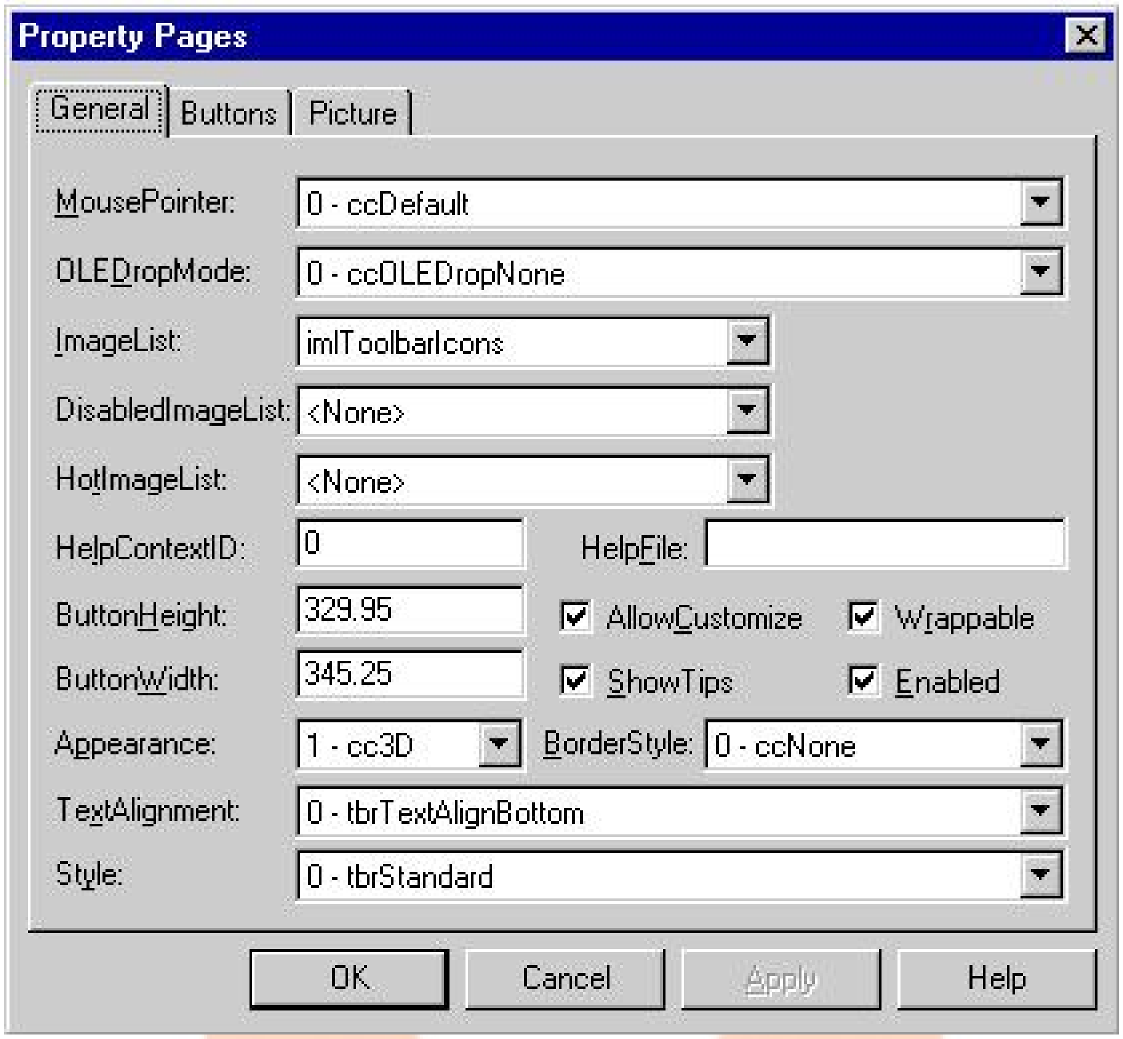
General properties

After you create a toolbar, you can access its Property Pages by right-clicking on it and choosing Properties. The General tab of the Property Pages dialog box includes most of the design-time properties that let you control the fine points of the appearance and behavior of a Toolbar control, as shown in Figures 10-15 and 10-16. For example, you make the following decisions: Whether the end user can customize the toolbar at run time (AllowCustomize property), whether the toolbar will wrap on multiple lines when the form is resized (Wrappable property), whether ToolTips are visible (ShowTips property), and what the default size of buttons (ButtonWidth and ButtonHeight properties) is. If necessary, buttons

are automatically enlarged to account for their caption or image, so in most cases you don't need to edit the default values of the latter two properties.

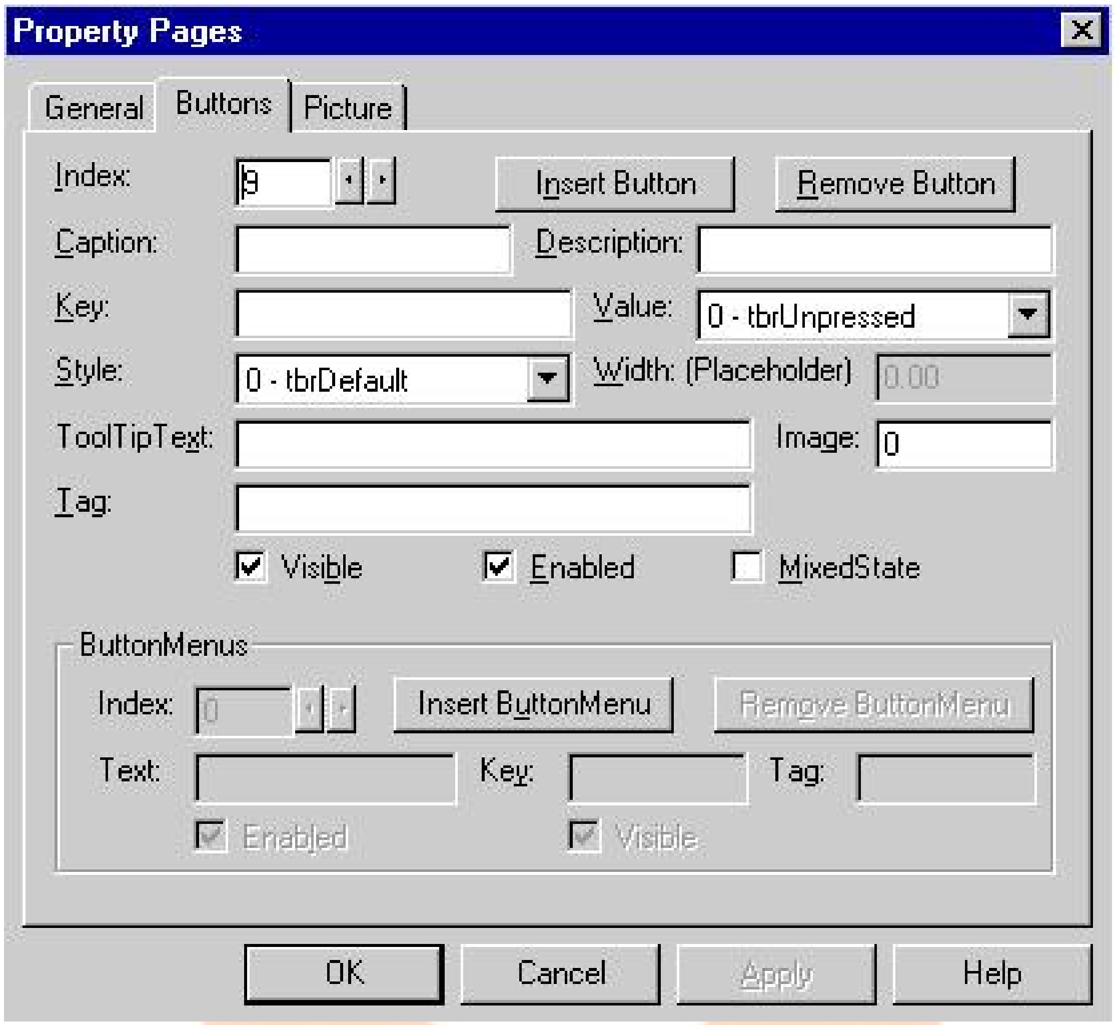
A few new properties let you access the most interesting features introduced in Visual Basic 6. You can create flat toolbars by setting the Style property to the value 1-tbrFlat, and you can use the TextAlignment property to modify the alignment of a button's caption with respect to the button's image (0-tbrTextAlignBottom or 1-tbrTextAlignRight).

A toolbar's button can be in three possible states: normal, disabled, or selected. (The selected state occurs when the mouse passes over the button if Style is 1-tbrFlat.) Instead of having three properties to point to different images of the same ImageList control, the Toolbar control uses a different approach: Each Button object exposes only one Image property—an index or a string key—and the state of the button implicitly affects which ImageList control will be used. You assign these three ImageList controls to the ImageList, DisabledImageList, and HotImageList properties either at design time or at run time. For example, you can mimic the behavior of Internet Explorer 4 by using a set of black-and-white icons for the normal state and a set of colorful icons for the selected state. If you don't assign the latter two properties, the Toolbar automatically creates a suitable image for the disabled or selected state.



MousePointer:	0 - ccDefault		
OLEDropMode:	0 - ccOLEDropNone		
ImageList:	imlToolbarIcons		
DisabledImageList:	<None>		
HotImageList:	<None>		
HelpContextID:	0	HelpFile:	
ButtonHeight:	329.95	✓ AllowCustomize	✓ Wrappable
ButtonWidth:	345.25	✓ ShowTips	✓ Enabled
Appearance:	1 - cc3D	BorderStyle:	0 - ccNone
TextAlignment:	0 - tbrTextAlignBottom		
Style:	0 - tbrStandard		

The General tab of the Property Pages dialog box of a Toolbar control.



Index: 9
Caption:
Description:
Key:
Value: 0 - tbrUnpressed
Style: 0 - tbrDefault
Width: (Placeholder) 0.00
ToolTipText:
Image: 0
Tag:
[✓] Visible [✓] Enabled [ ] MixedState

ButtonMenus
Index: 0
Text:
Key:
Tag:
[✓] Enabled [✓] Visible

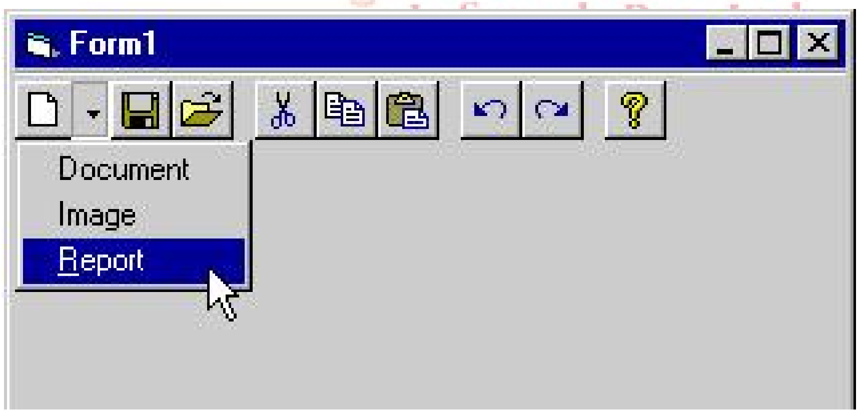
The Buttons tab of the Property Pages dialog box of a Toolbar control.

Button objects

A Toolbar control without any Button objects is useless. You can add Button objects using the Toolbar Wizard, as I explained previously, or you can do it in the Buttons tab of the Property Pages dialog box, as you can see in Figure 10-16. Each Button has a Caption property (use an empty string if you want to display only the icon), an optional Description that appears during a customization operation, a Tag property, a Key in the Buttons collection (optional, but use it to improve the readability of your code), a ToolTipText that appears if the Toolbar's ShowTips property is True, and an Image index or key to the associated ImageList controls.

Style is the most interesting property of a Button object. This property affects the appearance and behavior of the button and can be assigned one of the following values: 0-tbrDefault (a normal button, which behaves like a push button), 1-tbrCheck (a button that stays down when pressed, much like a CheckBox control), 2-tbrButtonGroup (a button that belongs to a group in which only one item can be in the selected state, similar to an OptionButton control), 3-tbrSeparator (a separator of fixed width), 4-tbrPlaceholder (a separator whose size depends on the Width property; this style is used to make room for another control placed on the toolbar), or 5-tbrDropDown (a button with a down arrow beside it, which displays a drop-down menu when clicked).

When the Style property is set to the value 5-tbrDropDown, you can add one or more ButtonMenu objects to the current Button. (You can create ButtonMenu items regardless of the button's style, but they're visible only when the style is tbrDropDown.) Each ButtonMenu object has a Text property (the caption of the menu line), an optional Key in the ButtonMenus collection, and a Tag property. Unfortunately, you can't associate an image with a ButtonMenu object: Drop-down menus are inherently text-only, which definitely contrasts with the graphical nature of the Toolbar control. See Figure 10-17 for an example of a Toolbar control whose first button has an associated drop-down menu.



A toolbar with a drop-down menu.

Run-Time Operations

Once you have added a Toolbar control to a form, you have to trap the user's actions on it. You might also need to programmatically build the control at run time or let the user customize it and save the new layout for subsequent sessions.

Creating Button and ButtonMenu objects

You can create new Button objects at run time using the Add method of the Buttons collection, which has the following

```
Add([Index], [Key], [Caption], [Style], [Image]) As Button
```

Index is the position at which the Button object will be inserted in the collection, Key is its optional string key, Caption is the text visible on the toolbar, Style determines the type of the button being added (0-tbrNormal, 1-tbrCheck, 2tbrButtonGroup, 3-tbrSeparator, 4-tbrPlaceholder, or 5-tbrDropDown), and Image is the index or the key of an image in the three companion ImageList controls.

If you create a Button object whose Style property is tbrDropDown, you can add one or more items to its ButtonMenus collection by using the collection's Add method:

```
Add ([Index], [Key], [Text]) As ButtonMenu
```

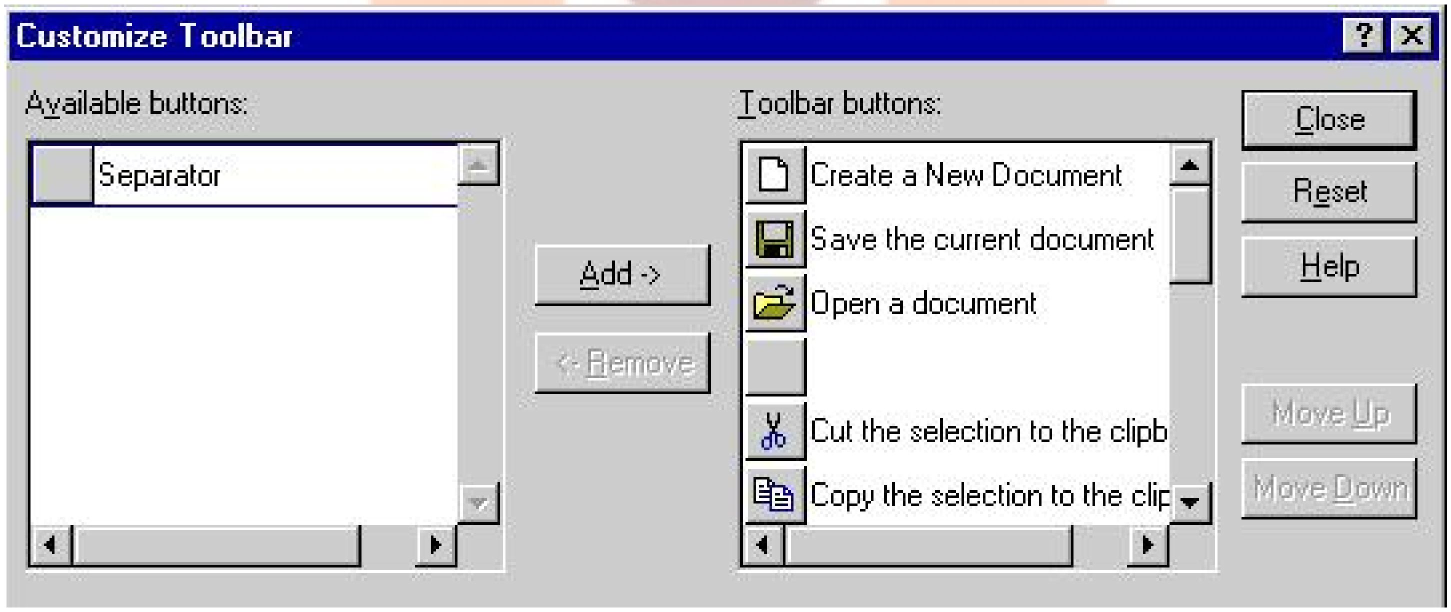
Customizing the Toolbar control

You can allow for users to customize the Toolbar control if you want. You can choose from two ways to achieve this: You set the AllowCustomization property to True to let users enter

customization mode by double-clicking on the toolbar, or you programmatically enter customization mode by executing the Toolbar's Customize method. The latter approach is necessary if you want to provide this capability only to a restricted group of users:

```
Private Sub Toolbar1_DblClick()

  If UserIsAdministrator Then Toolbar1.Customize

End Sub
```

Whatever method you choose, you end up displaying the Customize Toolbar dialog box



▫ Slider Control

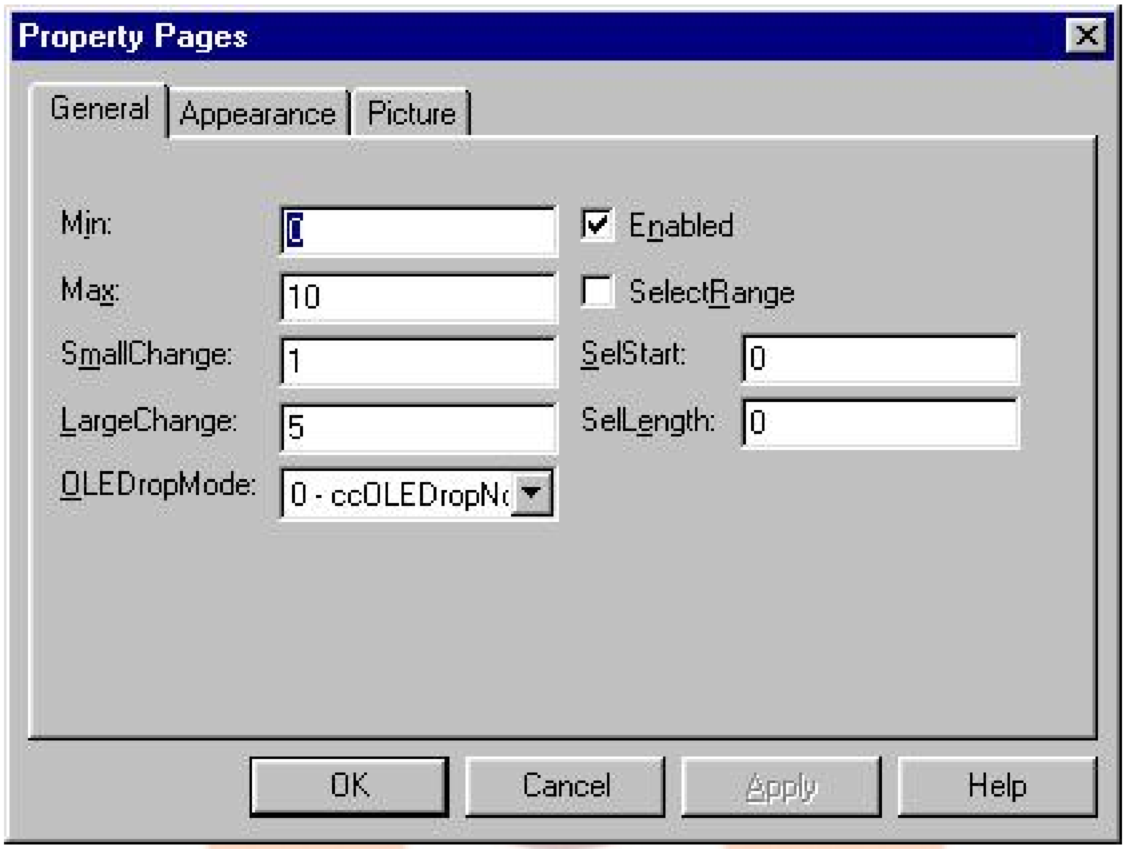
Introduction

The Slider control provides a way for end users to select a numerical value in a range. Conceptually, it's akin to the ScrollBar control, with which it shares many properties and events. A major difference is that there's only one kind of Slider control, which can create both horizontal and vertical sliders. The Slider control can also work in select-range mode, allowing your users to select a range rather than a single value.

Setting Design-Time Properties

Once you drop a Slider control on a form, you should right-click it and select the Properties menu command. In the General tab of the Properties custom dialog box, you can set the Min, Max, SmallChange, and LargeChange properties, which have the same meaning and effects as in HScrollBar and VScrollBar controls. In this tab, you can also set the SelectRange property, but this operation is most often performed at run time. (See "Employing the SelectRange Mode" later in this section.)

In the Appearance tab, you set a few properties that are peculiar to this control. The Orientation property lets you set the direction of the slider. The TickStyle property lets you select whether the slider has unit ticks and where they appear. (Valid values are 0-sldBottomRight, 1-sldTopLeft, 2-sldBoth, and 3-sldNoTicks.) The TickFrequency property indirectly determines how many ticks will be displayed. For example, if Min is 0 and Max is 10 (the default settings), a TickFrequency that equals 2 displays 6 ticks. The TextPosition property lets you decide where the ToolTip appears. (See "Showing the Value as a ToolTip" later in this section.)



Min: 0
Max: 10
SmallChange: 1
LargeChange: 5
OLEDropMode: 0 - ccOLEDropN
Enabled [✓]
SelectRange
SelStart: 0
SelLength: 0

Run-Time Operations

For most practical purposes, you can deal with a Slider control at run time as if it were a scroll bar control: Slider controls expose the Value property and the Change and Scroll events, exactly as scroll bars do. The following brief sections describe two features of the Slider control that are missing from the scroll bar controls.

Showing the value as a ToolTip

Slider controls can display ToolTip text that follows the indicator when it's being dragged by the user. You can control this new Visual Basic 6 feature using two properties, Text and TextPosition. The

former is the string that appears in the ToolTip window; the latter determines where the ToolTip appears with respect to the indicator. (Possible values are 0-sldAboveLeft and 1-sldBelowRight.) You can also set the TextPosition property at design time in the Appearance tab of the Property Pages dialog box.

You generally use these properties to show the current value in a ToolTip window near the indicator. To do so, you need just one statement in the Scroll event procedure:

```
Private Sub Slider1_Scroll()

  Slider1.Text = "Value = " & Slider1.Value

End Sub
```

Date Picker

The DateTimePicker control is a text box especially designed for Date or Time values. The text box is subdivided into subfields, one for each individual component (day, month, year, hour, minute, and second). This control supports all the usual date and time formats (including a custom format) and the ability to return a Null value (if the user doesn't want to select a particular date). You can even define your own custom subfields.

At run time, end users can advance through subfields using the Left and Right arrow keys and can increment and decrement their values using the Up and Down arrow keys. They can display a drop-down calendar (if the UpDown property is set to False) or modify the current value of the highlighted component using the companion spin buttons (if the value of UpDown is True).

Setting Design-Time Properties

By default, a Down arrow appears to the right of the control, much like a regular ComboBox control: A click on the arrow drops down a calendar, which lets users select a date without typing any keys. If you set the UpDown property to True, however, the Down arrow is replaced by a pair of spin buttons, which let users increment or decrement the value of individual subfields using only the mouse.

The CheckBox property, if True, displays a check box near the left border of the control: Users can deselect this check box if they don't intend to actually select any dates. (See Figure 11-7.)

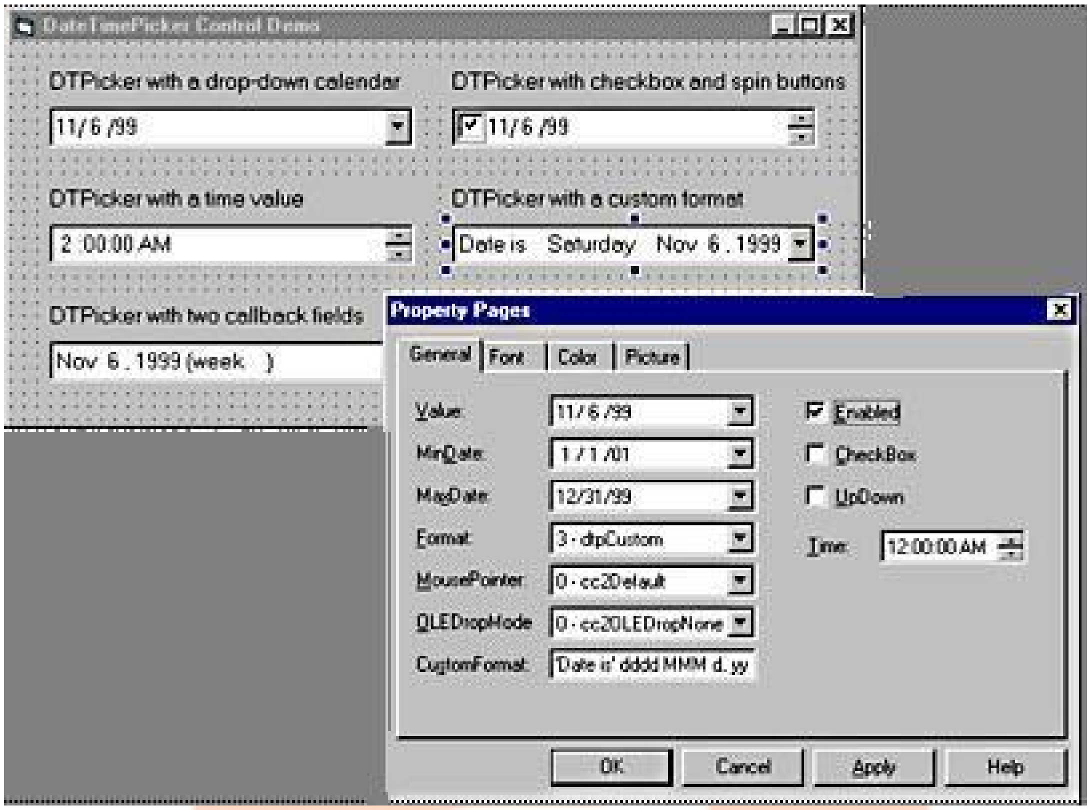
The DateTimePicker control shares a few properties with the MonthView control. For example, it exposes a Value property, which returns the Date value entered by the end user, and the MinDate and MaxDate properties, which define the interval of valid dates.

The drop-down calendar is nothing but a MonthView control that can show only one month at a time. Thus, the DateTimePicker control also exposes all the color properties of the MonthView control, even though each now has a different name: CalendarForeColor, CalendarBackColor, CalendarTitleForeColor, CalendarTitleBackColor, and CalendarTrailingForeColor. Oddly, the control doesn't expose the standard ForeColor and BackColor properties, so while you can modify the appearance of the drop-down calendar, you can't programmatically change the default colors of the edit portion of the control!

The Format property affects what's displayed in the control and can be one of the following values: 0-dtpLongDate, 1-dtpShortDate, 2-dtpTime, or 3-dtpCustom. If you select a custom format, you can assign a suitable string to the CustomFormat property. This property accepts the same formatting strings that you would pass to a Format function that works with date or time values. You can use this string:

```
'Date is' dddd MMM d, yyy

to display a value such as

Date is Friday Nov 5, 1999
```

As you see, you can include literal strings by enclosing them within single quotation marks. As I'll explain in a moment, the CustomFormat property can be used to create custom subfields too.


DateTimePicker Control Demo

DTPicker with a drop-down calendar: 11/6 /99

DTPicker with checkbox and spin buttons: ☑ 11/6 /99

DTPicker with a time value: 2 :00:00 AM

DTPicker with a custom format: Date is Saturday Nov 6 . 1999

DTPicker with two callback fields: Nov 6 . 1999 (week)

Property Pages — General | Font | Color | Picture

Value	11/6 /99
MinDate	1 / 1 /01
MaxDate	12/31/99
Format	3 - dtpCustom
MousePointer	0 - cc2Default
OLEDropMode	0 - cc2OLEDropNone
CustomFormat	'Date is' dddd MMM d, yy

☑ Enabled
☐ CheckBox
☐ UpDown
Time: 12:00:00 AM

. Different styles of the DateTimePicker control.

The DateTimePicker control can be bound to a data source, so it exposes the usual DataSource, DataMember, DataField, and DataFormat properties. The DataFormat property isn't supported when the control is bound to a standard Data or RemoteData control, but in either case you can modify the format of the displayed value using the Format and CustomFormat properties.

Run-Time Operations

At run time, you set and retrieve the contents in the DateTimePicker control through the Value property or by means of the Year, Month, Day, DayOfWeek, Hour, Minute, and Second properties. For example, you can programmatically increment the month portion of a date displayed in a DateTimePicker control with the following statements:

```
DTPicker1.Month = (DTPicker1.Month Mod 12) + 1

If DTPicker1.Month = 1 Then DTPicker1.Year = DTPicker1.Year + 1
```

If CheckBox is True and the user has deselected the check box, all date-related properties return Null.

The DateTimePicker control exposes many of the events supported by a standard TextBox control, including Change, KeyDown, KeyPress, KeyUp, MouseDown, MouseMove, MouseUp, Click, and DblClick. All keyboard and mouse events refer to the edit portion of the control and so don't fire when a calendar has been dropped down.

When the user clicks on the Down arrow, a DropDown event fires just before the drop-down calendar actually appears—that is, if the UpDown property is False (the default value). When the user selects a date in the drop-down calendar, a CloseUp event fires. These events aren't particularly useful, however, because you don't have much control over the calendar itself, apart from the colors it uses. When the user selects a date in the drop-down calendar, the Change event fires before the CloseUp event.

Managing callback fields

The most intriguing feature of the DateTimePicker control is the capability to define custom subfields, also known as callback fields. To define a callback field, you use a string of one or more X characters in the value assigned to the CustomFormat property. You can define multiple callback fields by using strings with different numbers of Xs. For example, the following format defines a date field with two callback fields:

```
DTPicker1.CustomFormat = "MMM d, yyy '(week 'XX')' XXX"
```

In the code sample that follows, the XX field is defined as the number of weeks since January 1, and the XXX field is the name of the holiday, if any, that occurs on the displayed date.

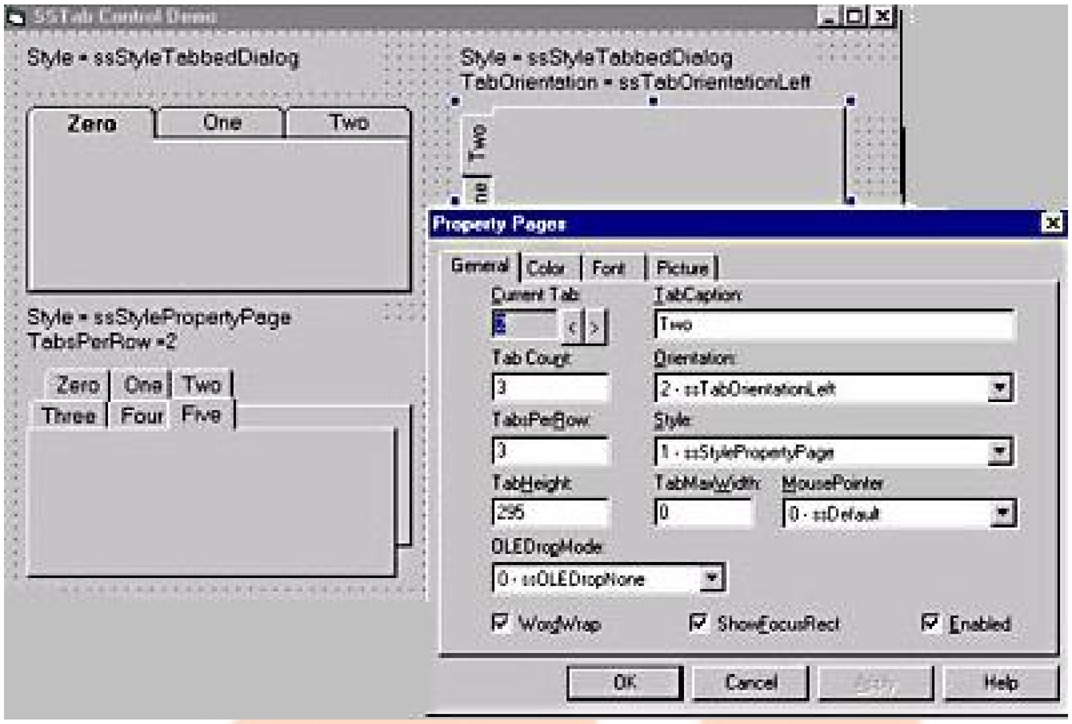
Tabbed Control

The SSTab control permits you to create tabbed dialog boxes almost the same way the TabStrip common control does it. The most important difference between the two controls is that the SSTab control is a real container, so you can place child controls directly on its surface. You can even switch among tabbed pages at design time, making the job of preparing the control much simpler and quicker than with the TabStrip control. Many programmers find it easier to work with the SSTab control because the control doesn't contain dependent objects, and the syntax of properties and events is more straightforward.

The SSTab control is embedded in the TabCtl32.ocx file, which must therefore be distributed with any Visual Basic application that uses this control.

Setting Design-Time Properties

The first thing to do after you drop an SSTab control on a form is to change its Style property from the default 0-ssStyleTabbedDialog value to the more modern 1-ssStylePropertyPage setting, which you can see in Figure 12-10. The tabs are usually displayed on the upper border of the control, but you can change this default setting by using the TabOrientation property.



Style = ssStyleTabbedDialog

Style = ssStyleTabbedDialog
TabOrientation = ssTabOrientationLeft

Style = ssStylePropertyPage
TabsPerRow =2

Current Tab: 2
TabCaption: Two
Tab Count: 3
Orientation: 2 - ssTabOrientationLeft
TabsPerRow: 3
Style: 1 - ssStylePropertyPage
TabHeight: 295
TabMaxWidth: 0
MousePointer: 0 - ssDefault
OLEDropMode: 0 - ssOLEDropNone
☑ WordWrap ☑ ShowFocusRect ☑ Enabled

The General tab of the Property Pages dialog box of an SSTab control.

You can add new tabs (or delete existing ones) by typing a value in the TabCount field (which corresponds to the Tabs property), and you can create multiple rows of tabs by setting a suitable value for the TabsPerRow property. After you have created enough tabs, you can use the spin buttons to move from tab to tab and modify each one's TabCaption property. (This property is the only field in the dialog box whose value depends on the Current Tab field.) Tab captions can include & characters to define hot keys for a quick selection.

The TabHeight property is the height in twips of all the tabs in the control. The TabMaxWidth property is the maximum width of a tab. (A zero width means that the tab is just large enough to accommodate its caption.) The WordWrap property must be True to let longer captions wrap around. If ShowFocusRect is True, a focus rectangle is displayed on the tab that has the focus.

Each tab can display a little image. To set it at design time, you first set the current tab in the General page of the Properties dialog box,

switch to the Picture tab, click on the Picture property in the leftmost listbox, and then select the bitmap or icon that you want to assign to the current tab. This bitmap can be referenced in code using the TabPicture property.

After you have created the tabs you need, you can place controls on each one of them. This operation is simple because you can select tabs even at design time. But you should be aware of an important detail: From Visual Basic's standpoint, all the controls you place on different tabs are contained in the SSTab control. In other words, the container is the SSTab control, not its tab pages. This has a number of implications—for example, if you have two groups of OptionButton controls on two different tab pages of the SSTab control, you should place each group in a separate Frame or another container, otherwise Visual Basic sees them as a single group.

Run-Time Operations

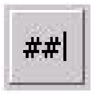
The main property of the SSTab control is Tab, which returns the index of the tab currently selected by the user. You can also set it to switch to another tab by means of code. The first tab has a 0 index.

Creating new tabs

You can create new tabs at run time by increasing the value of the Tabs property. You can append the new tab in one place only: following all the existing ones.

```
SSTab1.Tabs = SSTab1.Tabs + 1

SSTab1.TabCaption(SSTab1.Tabs - 1) = "Summary"
```

☐ Masked Edit Control



NOTE

The Masked Edit control can be added to Toolbox by selecting Microsoft Masked Edit Control 6.0 from the Components dialog box.

The Masked Edit control has several properties that assist in the validation of user input. Some of the frequently used properties are:

- **Mask**
- **Format**
- **Text and ClipText**
- **AutoTab**

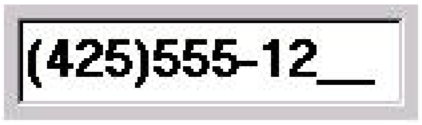
Mask Property

To define the Masked Edit control, use the Mask property. The Mask property forces data to be entered into a predefined template. You can set this property at design time or at run time. Although you can use standard formats at design time, and the control will distinguish between numeric and alphabetic characters, you may want to write

code to validate content such as the correct month or time of day. Each character position in the Masked Edit control corresponds to either a placeholder of a specified type or to a literal character.

The following code shows how to use the Mask property of the Masked Edit control to create an input mask for entering a United States telephone number, complete with placeholders for area code and local number:

mskPhone.Mask = "(###)###-####"


(425)555-12__

When the Mask property is an empty string (""), the control behaves like a standard TextBox control. When you define an input mask, underscores appear beneath every placeholder in the mask. You can replace a placeholder only with a character that is of the same type as the one specified in the input mask.

To clear the Text property when you have a mask defined, set the Text property to the default mask setting. For example:

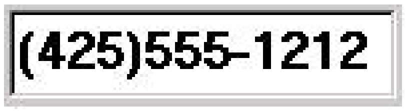
mskPhoneNumber.Text = "(___)___-____"

Format Property

Use the Format property to define the format for displaying and printing the contents of a control, such as numbers, dates, times, and text. You use the same format expressions as defined by the Visual Basic Format function, except that you cannot use named formats such as "Currency."

Text and ClipText

The Text property returns the data that the user has typed, along with the mask. The ClipText property returns only the data the user has typed. This is particularly important when implementing a Masked Edit control with a database. Figure 3.7 illustrates the text entered into the Masked Edit control, and the code that follows shows the use of these properties.


(425)555-1212

```
'The user entered 4255551212

Print "The user entered " & mskPhoneNumber.ClipText

'The control shows (425)555-1212

Print "The control shows " & mskPhoneNumber.Text
```

The ClipText property of the MaskEdit control shown in Figure 3.7 returns a value of 4255551212, and the Text property returns (425)555-1212.

AutoTab

When the AutoTab property is set to True, and the user enters the maximum number of characters specified by the Mask property for the control, the insertion point automatically moves to the control with the next TabIndex.

Accessing File - An Overview

Visual Basic has always included many powerful commands for dealing with text and binary files. While Visual Basic 6 hasn't extended the set of built-in functions, it has nonetheless indirectly extended the potential of the language by adding a new and interesting FileSystemObject object that makes it very easy to deal with files and directories. In this section, I provide an overview of all the VBA functions and statements related to files, with many useful tips so that you can get as much as you can from them and stay away from the most recurrent problems.

Handling Files

In general, you can't do many things to a file without opening it. Visual Basic lets you delete a file (using the Kill command), move or rename it (using the Name ... As command), and copy it elsewhere (using the FileCopy command):

```
' Rename a file--note that you must specify the path in the target,

' otherwise the file will be moved to the current directory.

Name "c:\vb6\TempData.tmp" As "c:\vb6\TempData.Doc"



' Move the file to another directory, possibly on another drive.

Name "c:\vb6\TempData.Doc" As "d:\VS98\Temporary.Dat"



' Make a copy of a file--note that you can change the name during the copy

' and that you can omit the filename portion of the target file.

FileCopy "d:\VS98\Temporary.Dat", "d:\temporary.Dat"



' Delete one or more files--Kill also supports wildcards.

Kill "d:\temporary.*"
```

You can read and modify the attributes of a file using the GetAttr function and the SetAttr command, respectively. The GetAttr function returns a bit-coded value, so you need to test its individual bits using intrinsic constants provided by VBA. Here's a reusable

function that builds a descriptive string with all the attributes of the file:

```
    ' This routine also works with open files
    ' and raises an error if the file doesn't exist.
Function GetAttrDescr(filename As String) As String
     Dim result As String, attr As Long
     attr = GetAttr(filename)
     ' GetAttr also works with directories.
     If attr And vbDirectory Then result = result & " Directory"
     If attr And vbReadOnly Then result = result & " ReadOnly"
     If attr And vbHidden Then result = result & " Hidden"
     If attr And vbSystem Then result = result & " System"
     If attr And vbArchive Then result = result & " Archive"
     ' Discard the first (extra) space.
     GetAttrDescr = Mid$(result, 2)
End Function
```

Similarly, you change the attributes of a file or a directory by passing

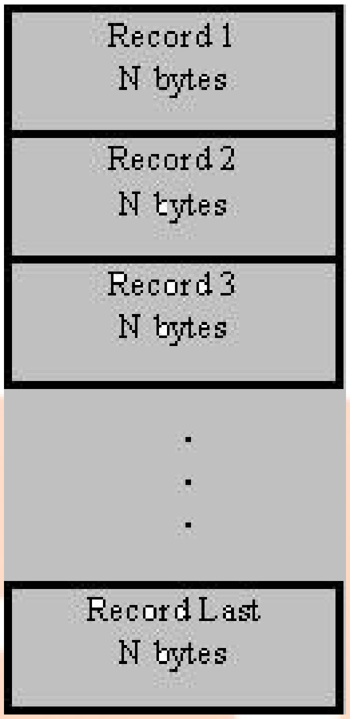
the SetAttr command a combination of values, as in the following code:

```
' Mark a file as Archive and Read-only.

filename = "d:\VS98\Temporary.Dat"

SetAttr filename, vbArchive + vbReadOnly

' Change a file from hidden to visible, and vice versa.

SetAttr filename, GetAttr(filename) Xor vbHidden
```

Random Access Files

- Note that to access a particular data item in a sequential file, you need to read in all items in the file prior to the item of interest. This works acceptably well for small data files of unstructured data, but for large, structured files, this process is time-consuming and wasteful. Sometimes, we need to access data in nonsequential ways. Files which allow nonsequential access are random access files.

- To allow nonsequential access to information, a random access file has a very definite structure. A random access file is made up of a number of records, each record having the same length (measured in bytes). Hence, by knowing the length of each record, we can easily determine (or the computer can) where each record begins. The first record in a random access file is Record 1, not 0 as used in Visual Basic arrays. Each record is usually a set of variables, of different types, describing some item. The structure of a random access file is:



- **A good analogy to illustrate the differences between sequential files and random access files are cassette music tapes and compact discs. To hear a song on a tape (a sequential device), you must go past all songs prior to your selection. To hear a song on a CD (a random access device), you simply go directly to the desired selection. One difference here though is we require all of our random access records to be the same length - not a good choice on CD's!**

- **To write and read random access files, we must know the record length in bytes. Some variable types and their length in bytes are:**

Type	Length(Bytes)
Integer	2
Long	4

Single	4
Double	8
String	1 byte per character

So, for every variable that is in a file's record, we need to add up the individual variable length's to obtain the total record length. To ease this task, we introduce the idea of user-defined variables.

User-Defined Variables

- **Data used with random access files is most often stored in user-defined variables. These data types group variables of different types into one assembly with a single, user-defined type associated with the group. Such types significantly simplify the use of random access files.**

- **The Visual Basic keyword Type signals the beginning of a user-defined type declaration and the words End Type signal the end. An example best illustrates establishing a user-defined variable. Say we want to use a variable that describes people by their name, their city, their height, and their weight. We would define a variable of Type Person as follows:**

```
Type Person

    Name As String

    City As String

    Height As Integer
```

```
        Weight As Integer

    End Type
```

These variable declarations go in the same code areas as normal variable declarations, depending on desired scope. At this point, we have not reserved any storage for the data. We have simply described to Visual Basic the layout of the data.

- To create variables with this newly defined type, we employ the usual Dim statement. For our Person example, we would use:

```
    Dim Lou As Person

    Dim John As Person

    Dim Mary As Person
```

And now, we have three variables, each containing all the components of the variable type Person. To refer to a single component within a user-defined type, we use the dot-notation

VarName.Component

As an example, to obtain Lou's Age, we use:

```
Dim AgeValue as Integer

.

.

AgeValue = Lou.Age
```

Writing and Reading Random Access Files

- **We look at writing and reading random access files using a user-defined variable. For other variable types, refer to Visual Basic on-line help. To open a random access file named RanFileName, use:**

  Open RanFileName For Random As #N Len = RecordLength

where N is an available file number and RecordLength is the length of each record. Note you don't have to specify an input or output mode. With random access files, as long as they're open, you can write or read to them.

- **To close a random access file, use:**

  Close N

- **As mentioned previously, the record length is the sum of the lengths of all variables that make up a record. A problem arises with String type variables. You don't know their lengths ahead of time. To solve this problem, Visual Basic lets you declare fixed lengths for strings. This allows you to determine record length. If we have a string variable named StrExample we want to limit to 14 characters, we use the declaration:**

  Dim StrExample As String * 14

  Recall each character in a string uses 1 byte, so the length of such a variable is 14 bytes.

- **Recall our example user-defined variable type, Person. Let's**

revisit it, now with restricted string lengths:

```
Type Person
    Name As String * 40
    City As String * 35
    Height As Integer
    Weight As Integer
End Type
```

The record length for this variable type is 79 bytes (40 + 35 +2 + 2). To open a file named PersonData as File #1, with such records, we would use the statement:

```
Open PersonData For Random As #1 Len = 79
```

- **The Get and Put statements are used to read from and write to random access files, respectively. These statements read or write one record at a time. The syntax for these statements is simple:**

```
Get #N, [RecordNumber], variable
Put #N, [RecordNumber], variable
```

The Gets statement reads from the file and stores data in the variable, whereas the Put statement writes the contents of the specified variable to the file. In each case, you can optionally specifiy the record number. If you do not specify a record number, the next sequential position is used.

- **The variable argument in the Get and Put statements is usually a single user-defined variable. Once read in, you obtain the component parts of this variable using dot-notation. Prior to writing a user-defined variable to a random access file, you 'load' the component parts using the same dot-notation.**

- **There's a lot more to using random access files; we've only looked at the basics. Refer to your Visual Basic documentation and on-line help for further information. In particular, you need to do a little cute programming when deleting records from a random access file or when 'resorting' records.**

Using the Open and Save Common Dialog Boxes

- **Note to both write and read sequential and random access files, we need a file name for the Open statement. To ensure accuracy and completeness, it is suggested that common dialog boxes (briefly studied in Class 4) be used to get this file name information from the user. I'll provide you with a couple of code segments that do just that. Both segments assume you have a common dialog box on your form named cdlFiles, with the CancelError property set equal to True. With this property True, an error is generated by Visual Basic when the user presses the Cancel button in the dialog box. By trapping this error, it allows an elegant exit from the dialog box when canceling the operation is desired.**

- **The code segment to obtain a file name (MyFileName with default extension Ext) for opening a file to read is:**

```
Dim MyFileName As String, Ext As String

.
```

```
.

cdlFiles.Filter = "Files (*." + Ext + ")|*." + Ext

cdlFiles.DefaultExt = Ext

cdlFiles.DialogTitle = "Open File"

cdlFiles.Flags = cdlOFNFileMustExist +
cdlOFNPathMustExist

On Error GoTo No_Open

cdlFiles.ShowOpen

MyFileName = cdlFiles.filename

.

.

Exit Sub

No_Open:

Resume ExitLIne

ExitLine:

Exit Sub

End Sub
```

A few words on what's going on here. First, some properties are set such that only files with Ext (a three letter string variable) extensions are displayed (Filter property), the default extension is Ext (DefaultExt property), the title bar is set (DialogTitle property), and some Flags are set to insure the file and path exist (see Appendix II for more common dialog flags). Error trapping is enabled to trap the Cancel button. Finally, the common dialog box is displayed and the filename property returns with the desired name. That name is put in the string variable MyFileName. What you do after obtaining the file name depends on what type of file you are dealing with. For sequential files, you would open the file, read in the information, and close the file. For random access files, we just open the file here. Reading and writing to/from the file would be handled elsewhere in your coding.

- **The code segment to retrieve a file name (MyFileName) for writing a file is:**

```
Dim MyFileName As String, Ext As String

.

.

cdlFiles.Filter = "Files (*." + Ext + ")|*." + Ext

cdlFiles.DefaultExt = Ext

cdlFiles.DialogTitle = "Save File"

cdlFiles.Flags = cdlOFNOverwritePrompt +
cdlOFNPathMustExist

On Error GoTo No_Save
```

```
cdlFiles.ShowSave

MyFileName = cdlFiles.filename

.

.

Exit Sub

No_Save:

Resume ExitLine

ExitLine:

EndSub

EndSub
```

Note this code is essentially the same used for an Open file name. The Flags property differs slightly. The user is prompted if a previously saved file is selected for overwrite. After obtaining a valid file name for a sequential file, we would open the file for output, write the file, and close it. For a random access file, things are trickier. If we want to save the file with the same name we opened it with, we simply close the file. If the name is different, we must open a file (using a different number) with the new name, write the complete random access file, then close it. Like I said, it's trickier.

- **We use both of these code segments in the final example where we write and read sequential files.**

Example :

Note Editor - Reading and Saving Text Files

1. We now add the capability to read in and save the contents of the text box in the Note Editor application from last class. Load that application. Add a common dialog box to your form. Name it cdlFiles and set the CancelError property to True.

2. Modify the File menu (use the Menu Editor and the Insert button) in your application, such that Open and Save options are included. The File menu should now read:

File

New

Open

Save

Exit

Properties for these new menu items should be:

Caption	Name	Shortcut
&Open	mnuFileOpen	[None]
&Save	mnuFileSave	[None]

3. The two new menu options need code. Attach this code to the

mnuFileOpen_Click event. This uses a modified version of the code segment seen previously. We assign the extension ned to our note editor files.

```
 Private Sub mnuFileOpen_Click()

cdlFiles.Filter = "Files (*.ned)|*.ned"

cdlFiles.DefaultExt = "ned"

cdlFiles.DialogTitle = "Open File"

cdlFiles.Flags = cdlOFNFileMustExist + cdlOFNPathMustExist

On Error GoTo No_Open

cdlFiles.ShowOpen

Open cdlFiles.filename For Input As #1

txtEdit.Text = Input(LOF(1), 1)

Close 1

Exit Sub

No_Open:

Resume ExitLine

ExitLine:

Exit Sub

End Sub
```

And for the mnuFileSave_Click procedure, use this code. Much of this can be copied from the previous procedure.

```
Private Sub mnuFileSave_Click()

cdlFiles.Filter = "Files (*.ned)|*.ned"

cdlFiles.DefaultExt = "ned"

cdlFiles.DialogTitle = "Save File"

cdlFiles.Flags = cdlOFNOverwritePrompt + cdlOFNPathMustExist

On Error GoTo No_Save

cdlFiles.ShowSave

Open cdlFiles.filename For Output As #1

Print #1, txtEdit.Text

Close 1

Exit Sub

No_Save:

Resume ExitLine

ExitLine:

Exit Sub

End Sub
```

Each of these procedures is similar. The dialog box is opened and, if a filename is returned, the file is read/written. If Cancel is pressed, no action is taken. These routines can be used as templates for file operations in other applications.

4. Save your application. Run it and test the Open and Save functions. Note you have to save a file before you can open one. Check for proper operation of the Cancel button in the common dialog box.

5. If you have the time, there is one major improvement that should be made to this application. Notice that, as written, only the text information is saved, not the formatting (bold, italic, underline, size). Whenever a file is opened, the text is displayed based on current settings. It would be nice to save formatting information along with the text. This can be done, but it involves a fair amount of reprogramming. Suggested steps:

A. Add lines to the mnuFileSave_Click routine that write the text box properties FontBold, FontItalic, FontUnderline, and FontSize to a separate sequential file. If your text file is named TxtFile.ned, I would suggest naming the formatting file TxtFile.fmt. Use string functions to put this name together. That is, chop the ned extension off the text file name and tack on the fmt extension. You'll need the Len() and Left() functions.

B. Add lines to the mnuFileOpen_Click routine that read the text box properties FontBold, FontItalic, FontUnderline, and FontSize from your format sequential file. You'll need to define some intermediate variables here because Visual Basic won't allow you to read properties directly from a file. You'll also need logic to set/reset any check marks in the menu structure to correspond to these input properties.

C. Add lines to the mnuFileNew_Click procedure that, when the user wants a new file, reset the text box properties FontBold, FontItalic, FontUnderline, and FontSize to their default values and set/reset the corresponding menu check marks.

D. Try out the modified application. Make sure every new option works as it should.

Actually, there are 'custom' tools (we'll look at custom tools in Class 10) that do what we are trying to do with this modification, that is save text box contents with formatting information. Such files are called 'rich text files' or rtf files. You may have seen these before when transferring files from one word processor to another.

6. Another thing you could try: Modify the message box that appears when you try to Exit. Make it ask if you wish to save your file before exiting - provide Yes, No, Cancel buttons. Program the code corresponding to each possible response. Use calls to existing procedures, if possible

Sequential Access Files

- In many applications, it is helpful to have the capability to read and write information to a disk file. This information could be some computed data or perhaps information loaded into a Visual Basic object.

- Visual Basic supports two primary file formats: sequential and random access. We first look at sequential files.

- A sequential file is a line-by-line list of data. You can view a sequential file with any text editor. When using sequential files, you must know the order in which information was written to the file to allow proper reading of the file.

- Sequential files can handle both text data and variable values. Sequential access is best when dealing with files that have lines with mixed information of different lengths. I use them to transfer data between applications.

Sequential File Output (Variables)

- We first look at writing values of variables to sequential files. The first step is to Open a file to write information to. The syntax for opening a sequential file for output is:

Open SeqFileName For Output As #N

where SeqFileName is the name of the file to open and N is

an integer file number. The filename must be a complete path to the file.

- **When done writing to the file, Close it using:**

Once a file is closed, it is saved on the disk under the path and filename used to open the file.

- **Information is written to a sequential file one line at a time. Each line of output requires a separate Basic statement.**

- **There are two ways to write variables to a sequential file. The first uses the Write statement:**

Write #N, [variable list]

where the variable list has variable names delimited by commas. (If the variable list is omitted, a blank line is printed to the file.) This statement will write one line of information to the file, that line containing the variables specified in the variable list. The variables will be delimited by commas and any string variables will be enclosed in quotes. This is a good format for exporting files to other applications like Excel.

Example

Dim A As Integer, B As String, C As Single, D As Integer

.

.

Open TestOut For Output As #1

Write #1, A, B, C

Write #1, D

Close 1

After this code runs, the file TestOut will have two lines. The first will have the variables A, B, and C, delimited by commas, with B (a string variable) in quotes. The second line will simply have the value of the variable D.

- **The second way to write variables to a sequential file is with the Print statement:**

Print #N, [variable list]

This statement will write one line of information to the file, that line containing the variables specified in the variable list. (If the variable list is omitted, a blank line will be printed.) If the variables in the list are separated with semicolons (;), they are printed with a single space between them in the file. If separated by commas (,), they are spaced in wide columns. Be careful using the Print statement with string variables. The Print statement does not enclose string variables in quotes, hence, when you read such a variable back in, Visual Basic may have trouble knowing where a string ends and begins. It's good practice to 'tack on' quotes to string variables when using Print.

Example

Dim A As Integer, B As String, C As Single, D As Integer

- .

▪

Open TestOut For Output As #1

Print #1, A; Chr(34) + B + Chr(34), C

Print #1, D

Close 1

After this code runs, the file TestOut will have two lines. The first will have the variables A, B, and C, delimited by spaces. B will be enclosed by quotes [Chr(34)]. The second line will simply have the value of the variable D.

Quick Example: Writing Variables to Sequential Files

1. Start a new project.

2. Attach the following code to the Form_Load procedure. This code simply writes a few variables to sequential files.

```
 Private Sub Form_Load()

Dim A As Integer, B As String, C As Single, D As Integer

A = 5

B = "Visual Basic"

C = 2.15

D = -20

Open "Test1.Txt" For Output As #1
```

Open "Test2.Txt" For Output As #2

Write #1, A, B, C

Write #1, D

Print #2, A, B, C

Print #2, D

Close 1

Close 2

End Sub

3. Run the program. Use a text editor (try the Windows 95 Notepad) to examine the contents of the two files, Test1.Txt and Test2.Txt. They are probably in the Visual Basic main directory. Note the difference in the two files, especially how the variables are delimited and the fact that the string variable is not enclosed in quotes in Test2.Txt. Save the application, if you want to.

Sequential File Input (Variables)

- **To read variables from a sequential file, we essentially reverse the write procedure. First, open the file using:**

Open SeqFileName For Input As #N

where N is an integer file number and SeqFileName is a complete file path. The file is closed using:

Close N

- **The Input statement is used to read in variables from a sequential file. The format is:**

Input #N, [variable list]

The variable names in the list are separated by commas. If no variables are listed, the current line in the file N is skipped.

- **Note variables must be read in exactly the same manner as they were written. So, using our previous example with the variables A, B, C, and D, the appropriate statements are:**

```
Input #1, A, B, C

Input #1, D
```

These two lines read the variables A, B, and C from the first line in the file and D from the second line. It doesn't matter whether the data was originally written to the file using Write or Print (i.e. commas are ignored).

Quick Example: Reading Variables from Sequential Files

1. Start a new project or simply modify the previous quick example.

2. Attach the following code to the Form_Load procedure. This code reads in files created in the last quick example.

```
Private Sub Form_Load()

Dim A As Integer, B As String, C As Single, D As Integer

Open "Test1.Txt" For Input As #1
```

```
Input #1, A, B, C

Debug.Print "A="; A

Debug.Print "B="; B

Debug.Print "C="; C

Input #1, D

Debug.Print "D="; D

Close 1

End Sub
```

Note the Debug.Print statements and how you can add some identifiers (in quotes) for printed information.

3. Run the program. Look in the debug window and note the variable values. Save the application, if you want to.

4. Rerun the program using Test2.Txt as in the input file. What differences do you see? Do you see the problem with using Print and string variables? Because of this problem, I almost always use Write (instead of Print) for saving variable information to files. Edit the Test2.Txt file (in Notepad), putting quotes around the words Visual Basic. Rerun the program using this file as input - it should work fine now.

Writing and Reading Text Using Sequential Files

- **In many apllications, we would like to be able to save text information and retrieve it for later reference. This information could be a text file created by an application or the contents of a**

Visual Basic text box.

- **Writing Text Files:**

To write a sequential text file, we follow the simple procedure: open the file, write the file, close the file. If the file is a line-by-line text file, each line of the file is written to disk using a single Print statement:

```
Print #N, Line
```

where Line is the current line (a text string). This statement should be in a loop that encompasses all lines of the file. You must know the number of lines in your file, beforehand.

If we want to write the contents of the Text property of a text box named txtExample to a file, we use:

```
Print #N, txtExample.Text
```

Example

We have a text box named txtExample. We want to save the contents of the Text property of that box in a file named MyText.ned on the c: drive in the \MyFiles directory. The code to do this is:

```
Open "c:\MyFiles\MyText.ned" For Output As #1

Print #1, txtExample.Text

Close 1
```

The text is now saved in the file for later retrieval.

- **Reading Text Files:**

To read the contents of a previously-saved text file, we follow similar steps to the writing process: open the file, read the file, close the file. If the file is a text file, we read each individual line with the Line Input command:

```
Line Input #1, Line
```

This line is usually placed in a Do/Loop structure that is repeated untill all lines of the file are read in. The EOF() function can be used to detect an end-of-file condition, if you don't know, a prioiri, how many lines are in the file.

To place the contents of a file opened with number N into the Text property of a text box named txtExample we use the Input function:

```
txtExample.Text = Input(LOF(N), N)
```

This Input function has two arguments: LOF(N), the length of the file opened as N and N, the file number.

Example

We have a file named MyText.ned stored on the c: drive in the \MyFiles directory. We want to read that text file into the text property of a text box named txtExample. The code to do this is:

```
Open "c:\MyFiles\MyText.ned" For Input As #1

txtExample.Text = Input(LOF(1), 1)

Close 1
```

The text in the file will now be displayed in the text box.

Data Environment Designer

In short, it's a design-time representation of the ADO objects that you would otherwise create at run time. When you use a form designer, you're actually defining at design time the forms and controls Visual Basic will create at run time. You make your choices in a visual manner, without worrying about what Visual Basic actually does when the program runs. Similarly, you can use the DataEnvironment designer to define the behavior of ADO Connections, Commands, and Recordset objects. You can set their properties at design time by pressing the F4 key to bring up the Properties window or by using their custom property pages, exactly as you would do with forms and controls.

It works with any local and remote ADO connection and even supports multiple connections. Moreover, it qualifies as an ADO data source, so you can bind fields to it. To add a DataEnvironment designer to the current project, you can choose the Add Data Environment command from the Project menu.

Connection Objects

The main object in a DataEnviroment designer is the Connection object. It broadly corresponds to the form object in the Form designer in the sense that it's the top-level object. Unlike forms, however, a DataEnvironment designer instance can contain multiple Connection objects. You can create a Connection in many ways. When you create a DataEnvironment, it already contains a default

Connection object, so you simply need to set its properties. You do this either by pressing F4 to display the standard Properties window, or (better) by right-clicking on the object and selecting the Properties menu command to display its custom property pages. (You get the same effect by clicking on the Properties button on the DataEnviroment toolbar.) The Provider, Connection, Advanced, and All pages are exactly the same ones that you encountered when setting data link's properties in the DataView window or when creating the ConnectionString property of an ADO Data control.

The DesignUserName and DesignPassword properties let you set the user name and password you want to use when you're creating the DataEnvironment object, while RunUserName and RunPassword are the user name and password you want to use when the program is executing. For example, you might develop the application using an Administrator identity and then check how the application behaves at run time when a guest user logs in. You can decide whether you want to see the prompt when the connection opens, and you can use different settings for design time and run time. You usually set DesignPromptBehavior to adPromptComplete and RunPromptBehavior to adPromptNever; the latter prevents malicious users from logging on to other data sources or entering random user names and passwords until they manage to get into the system.

Command Objects

A Command object in the DataEnvironment designer represents an action performed on a database. A command object is always a child of a Connection object, in much the same way a control is always a child of a form. More precisely, you can create a stand-alone Command object, but you can't use it until you make it a child of a Connection object.

Creating a Command object

The easiest way to create a Command object is by dragging a table, a view, or a stored procedure from the DataView window into the DataEnvironment window. Visual Basic then creates the Command object that corresponds to that table, view, or stored procedure, and it also creates a parent Connection, if necessary. A Command object can be a child only of a Connection that refers to its own database. You can also create one or more Command objects that map to stored procedures in a database by clicking on the Insert Stored Procedures button on the DataEnvironment toolbar.

There are two kinds of Command objects: ones that return Recordsets and ones that don't. The former are SQL queries, stored procedures, tables, or views that return a Recordset (which can be empty, if no records in the database meet the selection criteria). The latter are SQL commands or stored procedures that insert, delete, or modify values in the database but don't return a set of records. For example, you can create a Command named AuthorsInCA that returns all the authors that live in California by using the following SQL query:

```
SELECT * FROM Authors WHERE State = 'CA'
```

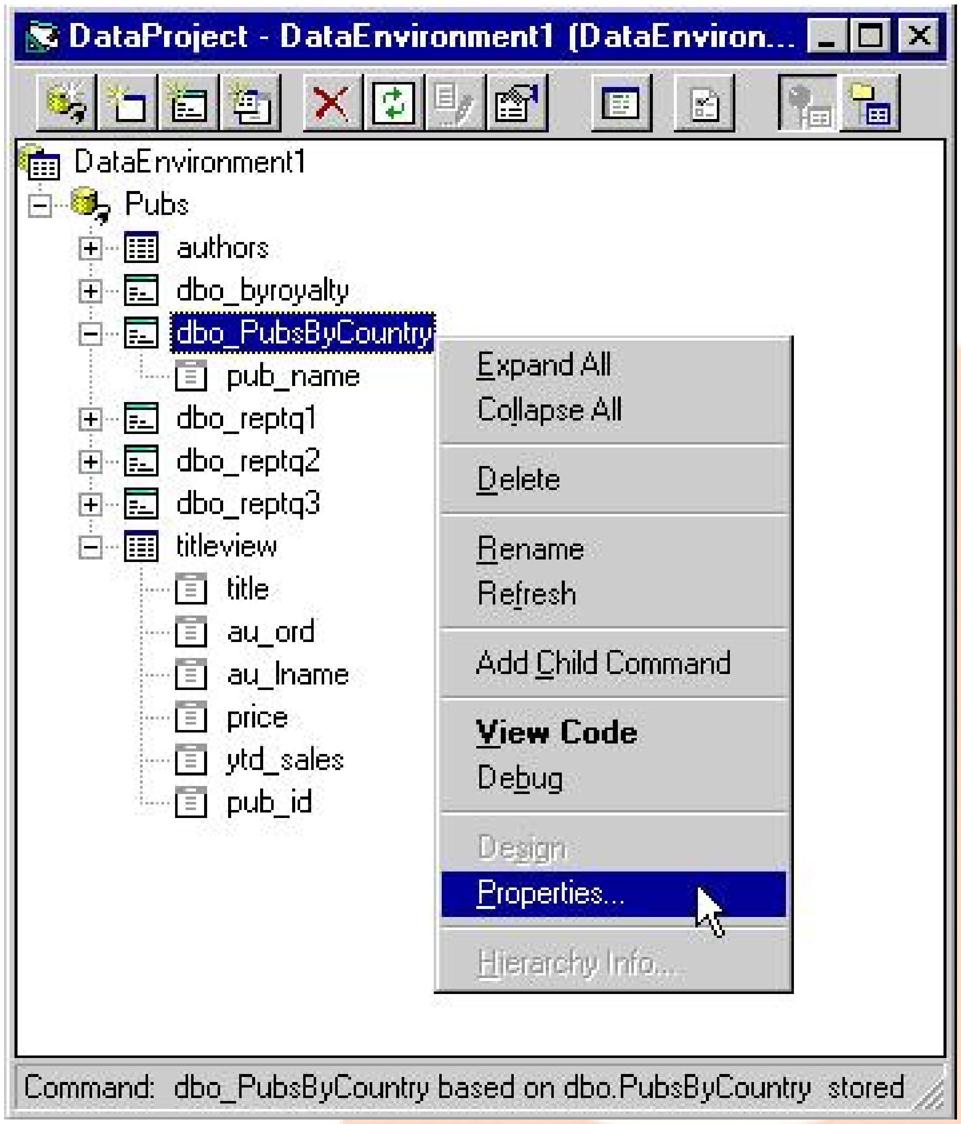
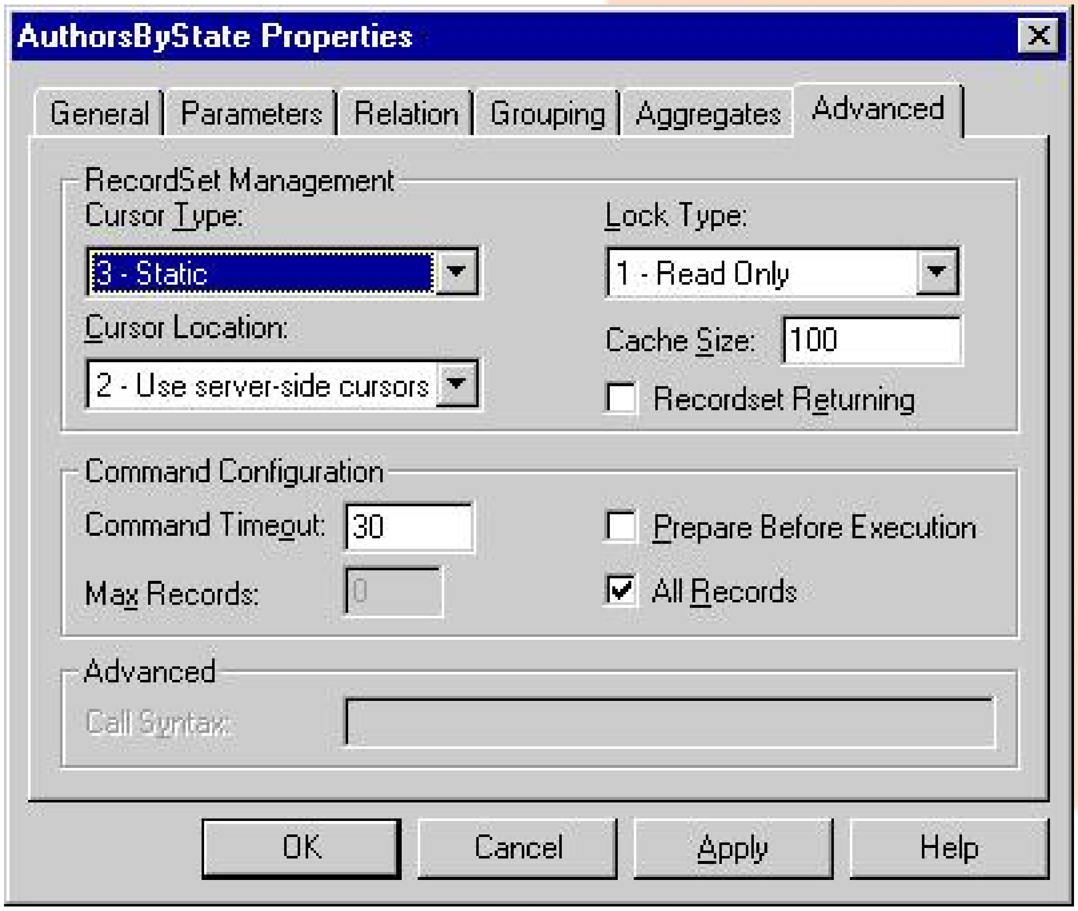


Figure : You can drag tables, views, and stored procedures from the DataView window to the DataEnvironment designer to create Command objects, and right-click on them to display the custom property pages.

If you have a normal, nonparameterized and nonhierarchical command, you can skip all the intermediate tabs and go to the Advanced page, shown in Figure 8-20. Here you decide the cursor type and location, the type of locking to be enforced, the size of the local cache (that is, the number of records read from the server

when necessary), the timeout for the command, and the maximum number of records that the query should return. You can use this last value to prevent a query from returning hundreds of thousands of records and so bringing your workstation and your network to their knees.



RecordSet Management
Cursor Type: 3 - Static
Cursor Location: 2 - Use server-side cursors
Lock Type: 1 - Read Only
Cache Size: 100
☐ Recordset Returning

Command Configuration
Command Timeout: 30
Max Records: 0
☐ Prepare Before Execution
☑ All Records

Advanced
Call Syntax:

Figure : The Advanced tab of the Command's property pages.

Parameterized commands

Using parameters adds a lot of flexibility to Command objects. You can create two types of parameterized Command objects: those based on a SQL query and those based on a stored procedure with

parameters. For the first kind, you must enter a parameterized SQL query, using question marks as placeholders for parameters. For example, you can create a Command object named AuthorsByState, which corresponds to the following query:

```
SELECT * FROM Authors WHERE State = ?
```

After you've entered this query in the General tab of the Properties dialog box, switch to the Parameters tab and check that the DataEnvironment has correctly determined that the query embeds one parameter. In this tab, you can assign a name to each parameter, set its data type and size, and so on. All parameters in this type of query are input parameters.

To create a Command object that maps a stored procedure, you can click on the Insert Stored Procedure button and select the stored procedure you're interested in. The DataEnvironment is usually able to retrieve the stored procedure syntax and correctly populate the Command's Parameters collection. You should pay attention to the direction of the parameters because sometimes the DataEnvironment doesn't correctly recognize output parameters and you have to manually fix their Direction attribute. Also, double-check that all string parameters have nonzero sizes.

Data Report

Before using the DataReport designer, you must make it accessible from the IDE, which you do by issuing the Components command from the Project menu, switching to the Designer tab, and ticking the Data Report check box. Alternatively, you can create a new Data Project and let Visual Basic create an instance of the DataReport designer for you.

The DataReport designer works in bound mode only, in the sense that it's able to automatically retrieve the data to be sent to the printer or simply displayed in the preview window. It can export a report to a text file or an HTML file and also supports custom format layouts. The DataReport designer comes with a set of custom controls that you can drop on its surface in the same way as you do with forms and other designers. These controls include lines, shapes, images, and also function fields, which you can use to create summary fields in your reports. Another intriguing feature of this designer is its ability to print in asynchronous mode, which lets the user perform other tasks while the printing proceeds.

Design-Time Operations

The simplest way to create a report using the DataReport designer is in conjunction with the DataEnvironment designer. The DataReport designer supports drag-and-drop operations of DataEnvironment's Command objects, including hierarchical Command objects. The only limitation is that the report can account for just one child Recordset at each nesting level. We'll use a hierarchical Command object based on the Orders and Order Details tables in the NWind.mdb database.

Binding to a Command object

Here are the steps you should follow to create a report based on the sample hierarchical Command object:

i. **Create a hierarchical Command, named Orders, that contains a child Command named Order Details. Ensure that it retrieves the information you're interested in.**

ii. **Create a new instance of the DataReport designer.**

iii. **Bring up the Properties window, let the DataReport's DataSource property point to DataEnvironment1 (or whatever the name of your DataEnvironment is), and then set its DataMember property to Orders.**

iv. **Right-click on the Report Header of the DataReport designer, and select the Retrieve Structure menu command; this will create a Group Header and Group Footer section labeled Orders_Header and Orders_Footer, respectively; between them is a Detail section labeled Order_Details_Detail.**

v. **A section represents a block of data that will be repeated for each record in the parent Command object. The first section corresponds to the parent Command object, the second section to its child Command, and so on until you reach the Detail section, which corresponds to the innermost Command object. All the sections except the Detail section are divided into a header section and a footer section, which are printed before and after the information related to the sections pertaining to objects at an inner level. The DataReport designer also includes a Report section (which prints information at the beginning and end of the report) and a Page section (which prints information**

at the beginning and end of each page). If you don't see these two sections, right-click anywhere on the DataReport designer and select the appropriate menu command.

vi. Drag the fields you need from the Orders Command object in the DataEnvironment to the Orders_Header section of the DataReport. Whenever you release the mouse button, a pair of controls, RptLabel and a RptTextBox, appear in the DataReport. When the report is eventually displayed, the RptLabel control produces a constant string with the name of the field (or whatever you assigned to its Caption property), while the RptTextBox control is replaced by the actual contents of the corresponding database field. You can then arrange the fields in the Orders_Header section and delete the RptLabel controls that you don't want to display.

vii. Click on the Order Details Command object and drag it onto the DataReport; Visual Basic creates one RtpLabel-RptTextBox control pair for each field in the corresponding Recordset. You can then delete the OrderID field and arrange the others in a row, as displayed in Figure given below.

viii. Adjust each section's height so that it doesn't take more room than strictly necessary. This is especially important for the Detail section, because it will be repeated for each single record in the Order Detail table. You can also reduce all the sections that don't contain any fields to a null height.

ix. What you've done so far is sufficient to see the DataReport in action. Bring up the Project Property Pages dialog box, select DataReport1 as the startup object, and then run the program. Or you can run it from any event with the use of .Show method just like we do with standard form.

x. Before moving on to another topic, a couple of notes about the placement of controls are in order. First, you can drop any control in the section that corresponds to the Command object it belongs to, as well as in any section with a deeper nesting level. For example, you can drop the OrderID field from the Orders Command in both the Orders section and the Order_Details section. You can't, however, move the UnitPrice field from the inner Order_Details section to the Order section. Second, you shouldn't drop binary fields or fields containing images from the DataEnvironment onto the DataReport designer; Visual Basic won't generate an error, but it will create a RptTextBox control that contains meaningless characters at run time.

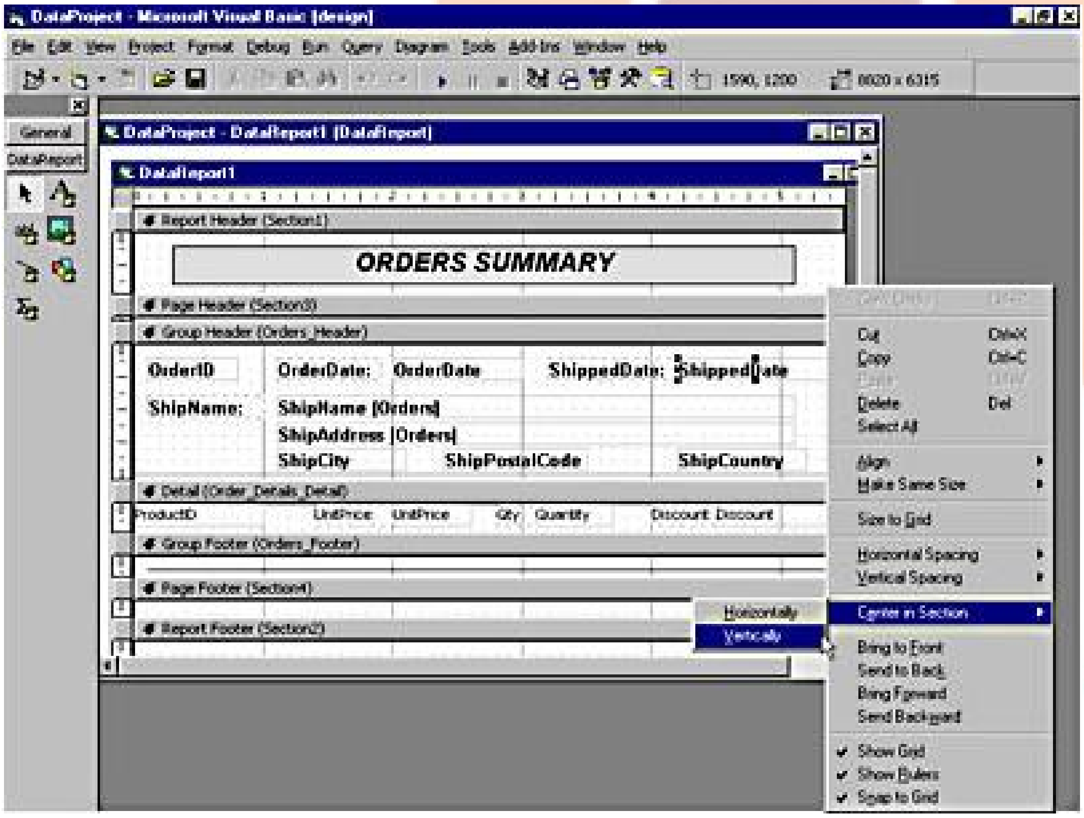


Figure : The DataReport designer at design time, with the pop-up

menu that appears when you right-click on a control.

Setting control properties

The controls you have dropped on the DataReport's surface are similar to the standard controls you place on a form, but they belong to a different control library. In fact, you can't drop a standard intrinsic control on a DataReport designer, nor can you place a control from the DataReport control library on a form or another designer. But you can move DataReport controls and align them as you would do with any regular control. you should already be familiar with most of the properties you find in this window. For example, you can change the DataFormat properties of the txtOrderDate and txtShippedDate controls so that they display their values in long date format. Or you can change the txtOrderID control's BackStyle property to 1-rptBkOpaque and its BackColor property to gray (&HE0E0E0) so that order identifiers are highlighted in the report. RptLabel controls don't expose any Dataxxxx property; they're just cosmetic controls that insert fixed strings in the report. The only custom property that we haven't seen yet is CanGrow, which applies to both the RptLabel and RptTextBox controls. If this property is True, the control is allowed to expand vertically when its content exceeds the control's width. The default value for this property is False, which causes longer strings to be truncated to the control's width.

Displaying calculated fields

The first way, which is suitable for calculated values that depend on other values in the same record, requires that you modify the SELECT command to include the calculated field in the list of fields to be retrieved.

SELECT OrderID, ProductID, UnitPrice, Quantity, Discount, ((UnitPrice*Quantity)*(1-Discount)) AS Total FROM Order_Details

Then you might add a Total field in the Detail section that lists the total price for each record from the Order Details table. Remember to align the field to the right and allow for the correct number of digits after the decimal point.

The second technique for adding a calculated field is based on RptFunction controls and is suitable for summary fields. For example, let's add a field that evaluates the total value of each order. This requires calculating the sum of the values of the Total field in the Order_Details Command. To do this, you must drop a RptFunction control into the Orders_Footer section—that is, the first footer after the section where the data to be summed is displayed. Then set the new control's DataMember property to Order_Details, its DataField property to Total, its FunctionType to 0rptFuncSum, and its DataFormat property to Currency. Using the same approach, you can add a summary field with the total number of distinct products in the order, by setting DataField to ProductID and FunctionType to 4-rptFuncRCnt.

You're not forced to place a RptFunction control in the footer section that immediately follows the section where the data field is. For example, to evaluate the sum of the Total fields from the Order_Details Command, you can add a RptFunction control in the Report Footer section, and you can add another RptFunction control to calculate the sum of the Freight fields from the Orders section. In any case, you only have to set these controls' DataMember properties to point to the correct Command object. Unfortunately, you can't place a RptFunction control in a Page Footer section, so you can't have totals at the end of each page.

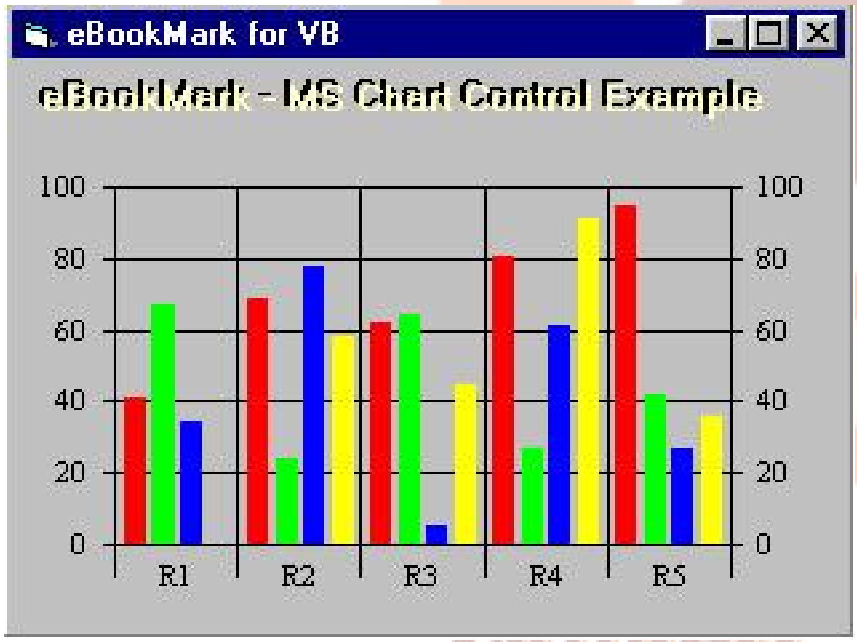
It is easy to generate report that groups records. For example, to display a list of customers grouped by country, all you have to do is create a Command object linked to the Customers table, switch to the Grouping tab of its Property Pages dialog box, and group the Command object by its Country field. This operation creates a new Command object with two folders. You can then assign this Command to the DataMember property of a DataReport designer and issue the Retrieve Structure command to let the designer automatically create the necessary sections. The sample application on the companion CD includes a report built using this technique.

Managing page footers and page breaks

You can place controls in a Page Header or Page Footer section, typically to display information about the current page number, the total number of pages, the date and time of the report, and so forth. To do this, right-click in the section of interest, select the Insert Control menu command, and then from a pop-up menu select the information you want to display. A control created in this way is a RptLabel, which contains special characters in its Caption property.

Graphs with (MS Chart Control)

The MSChart control is an external ActiveX control that lets you add charting capabilities to your applications. You can create two- and three-dimensional charts in different styles, including bars, lines, and pies. You have complete control over all the items in the chart, such as title, legends, footnotes, axes, data point series, and so on. You can even rotate the graph, add backdrop images to virtually any element of the chart, set up your own light sources, and place them where you want. At run time, users can select portions of the chart and move and resize them at will, if you want to provide them with this capability. Sample of chart control on the form is shown below figure



Setting Design-Time Properties

The MSChart control property pages dialog box is shown in Figure

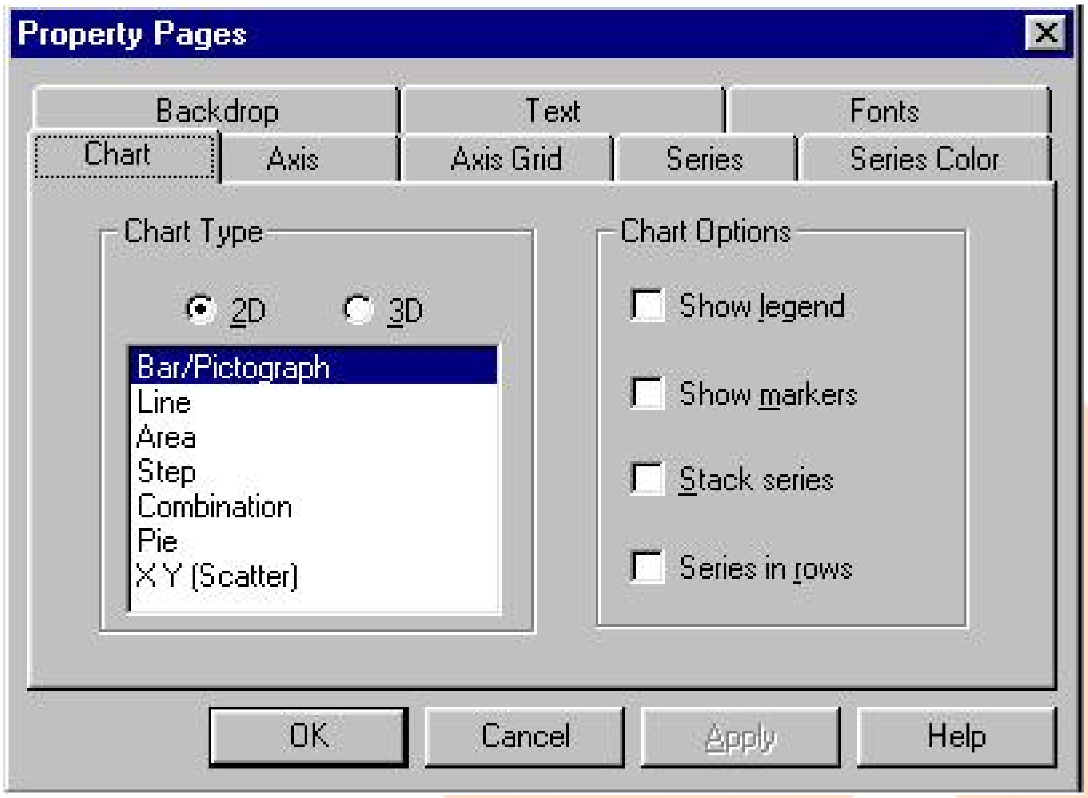


Chart Type

[✓ 2D] 3D

Bar/Pictograph
Line
Area
Step
Combination
Pie
X Y (Scatter)

Chart Options

☐ Show legend

☐ Show markers

☐ Stack series

☐ Series in rows

The Chart tab is where you decide which type of graphic you want to display, whether you want to stack series, and whether you show legends that explain what each data series is. These settings correspond to the ChartType, Chart3d, Stacking, and ShowLegend properties of the MSChart object.

The Axis tab is where you select the attributes of the axis of the chart: line width and color, whether the scale is displayed, and whether the scale is determined automatically by the control (the recommended setting) or manually by the programmer. In the latter case, you have to set minimum and maximum values and the frequency of divisions. Two-dimensional charts have three axes (x-axis, y-axis, and secondary y-axis), while three-dimensional charts have an additional fourth axis (z-axis). Your code can modify these properties using the Axis object, a child of the Plot object.

The AxisGrid tab lets you modify the style lines of axis grids; these settings correspond to the properties of the AxisGrid object, a child

of the Axis object.

In the Series tab, you define how each data series should be displayed. You can hide a series (but reserve the space for it on the chart), exclude it (this also reuses its space on the chart), show its markers, and draw it on the secondary y-axis. If you are drawing a two-dimensional Line chart, you can also display statistical data, such as minimum and maximum values, mean, standard deviation, and regression. You can modify these features through code by acting on the SeriesCollection and the Series objects.

You refine the appearance of each data series in the SeriesColor tab, where you select the color and the style of the edge and the interior of each series. (The latter isn't available for Line and X-Y charts.) Your code can manipulate these properties through the DataPoint object.

All the main objects in the control—MsChart, Plot, Title, Legend, and Footnote—can have a backdrop pattern. You define the color and style of each backdrop in the Backdrop tab of the Property Pages window. The title, the legends, and the axis in your graph expose a Title, and you can set its properties in the Text and the Font tabs.

Run-Time Operations

Unless you want to give users the ability to modify some key properties of your charts, you can define all the key properties at design time using the Property Pages dialog box so that at run time you only have to feed the MSChart control the actual data to be displayed. You achieve this using the DataGrid object.

You can think of the DataGrid object as a multidimensional array that

holds both data and its associated labels. You define the size of the array by assigning a value to the DataGrid's RowCount and ColumnCount properties, and you define the number of labels with the RowLabelCount and ColumnLabelCount properties. For example, you might have 12 rows of data to which you add a label at every third data point:

```
' 12 rows of data, with a label every third row

MSChart1.DataGrid.RowCount = 12

MSChart1.DataGrid.RowLabelCount = 4

' 10 columns of data, with a label on the 1st and 6th column

MSChart1.DataGrid.ColumnCount = 10

MSChart1.DataGrid.ColumnLabelCount = 2
```

Alternatively, you can set these four properties in one operation using the SetSize method:

```
' Syntax is: SetSize RowLabelCount, ColLabelCount,
RowCount, ColCount

MSChart1.DataGrid.SetSize 4, 2, 12, 10
```

You define the label text using the RowLabel and ColumnLabel properties, which accept two arguments: the row or column number and the number of the label you want to assign.

```
' Set a label every three years.

MSChart1.DataGrid.RowLabel(1, 1) = "1988"

MSChart1.DataGrid.RowLabel(4, 2) = "1991"

MSChart1.DataGrid.RowLabel(7, 3) = "1994"

' And so on.
```

You can set the value of individual data points using the SetData method, which has the following syntax:

MSChart.DataGrid.SetData Row, Column, Value, NullFlag

where Value is a Double value and NullFlag is True if the data is Null. You can easily (and quickly) insert or delete rows or columns using a number of methods exposed by the DataGrid object. Among these are InsertRows, DeleteRows, InsertColumns, DeleteColumns, InsertRowLabels, DeleteRowLabels, InsertColumnLabels, and DeleteColumnLabels. You can also fill the grid with random values (useful for providing the user with visual feedback even without actual data values) with the method RandomDataFill.

☐ Win API - An Overview

The Windows operating system is heavily based on messages. For example, when the user closes a window, the operating system sends the window a WM_CLOSE message. When the user types a key, the window that has the focus receives a WM_CHAR message, and so on. (In this context, the term window refers to both top-level windows and child controls.) Messages can also be sent to a window or a control to affect its appearance or behavior or to retrieve the information it contains. For example, you can send the WM_SETTEXT message to most windows and controls to assign a string to their contents, and you can send the WM_GETTEXT message to read their current contents. By means of these messages you can set or read the caption of a top-level window or set or read the Text property of a TextBox control, just to name a few common uses for this technique.

Broadly speaking, messages belong to one of two families: They're control messages or notification messages. Control messages are sent by an application to a window or a control to set or retrieve its contents, or modify its behavior or appearance . Notification messages are sent by the operating system to windows or controls as the result of the actions users perform on them.

Visual Basic greatly simplifies the programming of Windows applications because it automatically translates most of these messages into properties, methods, and events. Instead of using

WM_SETTEXT and WM_GETTEXT messages, Visual Basic programmers can reason in terms of Caption and Text properties. Nor do they have to worry about trapping WM_CLOSE messages sent to a form because the Visual Basic runtime automatically translate them into Form_Unload events. More generally, control messages map to properties and methods, whereas notification messages map to events.

Not all messages are processed in this way, though. For example, the TextBox control has built-in undo capabilities, but they aren't exposed as properties or methods by Visual Basic and therefore they can't be accessed by "pure" Visual Basic code. (In this chapter, pure Visual Basic means code that doesn't rely on external API functions.) Here's another example: When the user moves a form, Windows sends the form a WM_MOVE message, but the Visual Basic runtime traps that message without raising an event. If your application needs to know when one of its windows moves, you're out of luck.

By using API functions, you can work around these limitations. In this section, I' show you how you can send a control message to a window or a control to affect its appearance or behavior, while in the "Callback and Subclassing" section of this chapter, I' illustrate a more complex programming technique, called window subclassing, which lets you intercept the notification messages that Visual Basic doesn't translate to events.

Before you can use an API function, you must tell Visual Basic the name of the DLL that contains it and the type of each argument. You do this with a Declare statement, which must appear in the declaration section of a module. Declare statements must be declared as Private in all types of modules except BAS modules (which also accept Public Declare statements that are visible from the entire application). For additional information about the Declare

statement, see the language documentation.

The main API function that you can use to send a message to a form or a control is SendMessage, whose Declare statement is this:

```
Private Declare Function SendMessage Lib "user32" Alias
"SendMessageA" _

  (ByVal hWnd As Long, ByVal wMsg As Long, _

  ByVal wParam As Long, lParam As Any) As Long
```

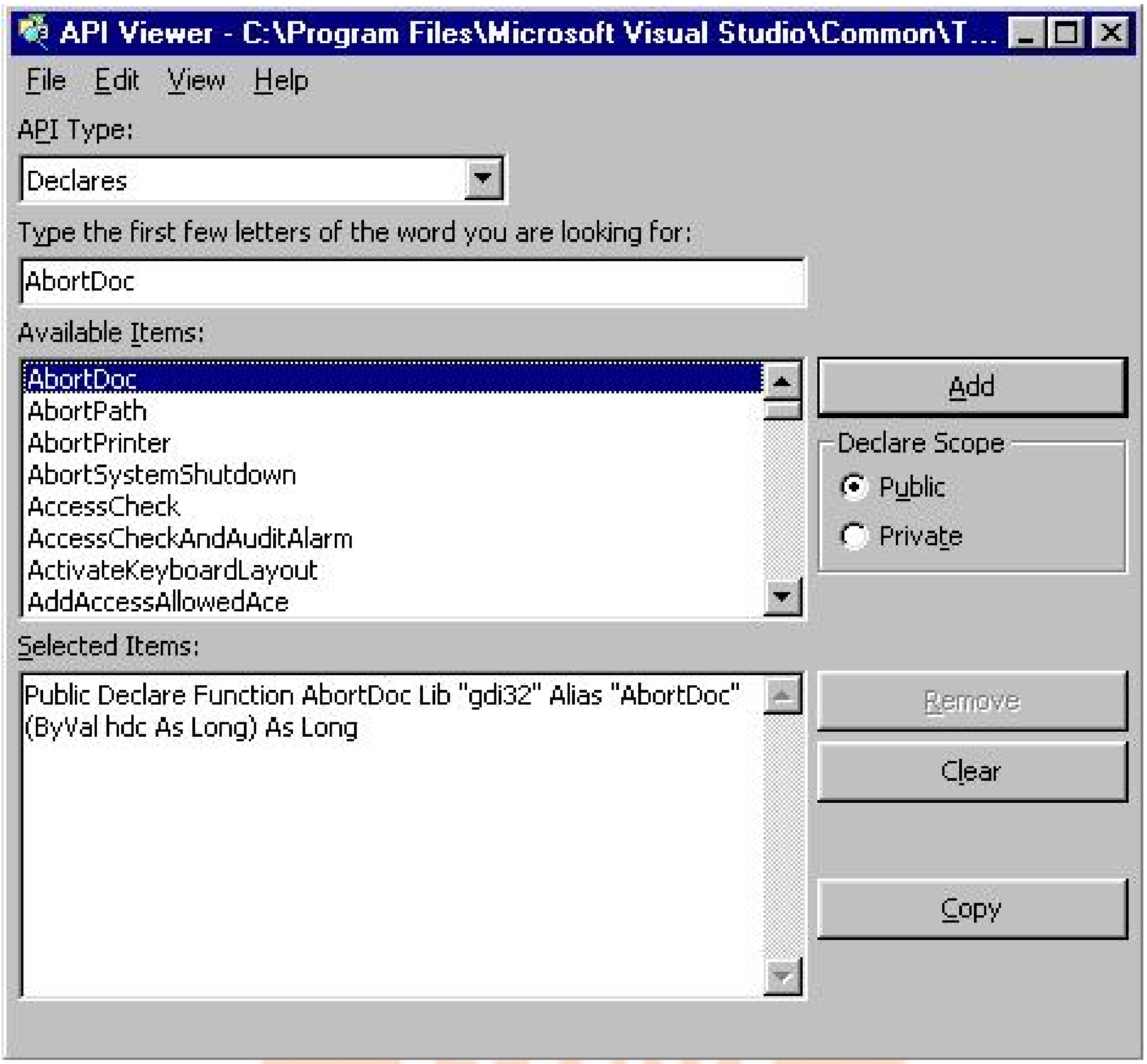
The hWnd argument is the handle of the window to which you're sending the message (it corresponds to the window's hWnd property), wMsg is the message number (usually expressed as a symbolic constant), and the meaning of the wParam and lParam values depend on the particular message you're sending. Notice that lParam is declared with the As Any clause so that you can pass virtually anything to this argument, including any simple data type or a UDT. To reduce the risk of accidentally sending invalid data, I've prepared a version of the SendMessage function, which accepts a Long number by value, and another version that expects a String passed by value. These are the so called type-safe Declare statements:

```
Private Declare Function SendMessageByVal Lib "user32" _
  Alias "SendMessageA" (ByVal hWnd As Long, ByVal wMsg As Long, _
  ByVal wParam As Long, Byval lParam As Long) As Long
Private Declare Function SendMessageString Lib "user32" _
  Alias "SendMessageA" ByVal hWnd As Long, ByVal wMsg As Long, _
  ByVal wParam As Long, ByVal lParam As String) As Long
```

Apart from such type-safe variants, the Declare functions used in this chapter, as well as the values of message symbolic constants, can be obtained by running the API Viewer utility that comes with Visual Basic.

CAUTION

When working with API functions, you're in direct touch with the operating system and aren't using the safety net that Visual Basic offers. If you make an error in the declaration or execution of an API function, you're likely to get a General Protection Fault (GPF) or another fatal error that will immediately shut down the Visual Basic environment. For this reason, you should carefully double-check the Declare statements and the arguments you pass to an API function, and you should always save your code before running the project.



Declare Scope: [✓ Public] Private

```
Public Declare Function AbortDoc Lib "gdi32" Alias "AbortDoc"
(ByVal hdc As Long) As Long
```

OLE

Introduction

An impottant festure of Microsoft Wndows Operating system is its ability for applications to share information. OLE is a means of communication, which gives applications the power to directly use and manipulate other windows applications.OLE is an important Windows topic in Visual Basic. Any object that supports OLE can be linked.OLE specification permits the user to link and embed objects, and also edit the object with in the container application. This chapter cover the basics of using other applications' objects in Visual Basic, featuring the OLE container control and OLE automation.

OLE Basics

DDE is an acronym for Dynamic Data Exchange.It is the basic foundation for inter process communication between applications.In DDE, the application creating a link is known as the destination application, and the application that responds is the source application.Although there are functional similarities between OLE and DDE there are some differences. Using DDE, unformatted data is exchanged. The Visual basic application has to format it appropriately. For e.g in case of an Excel Spreadsheet, the formula calculating the result is not fetched, but only the resultant number is fetched.

OLE is an abbreviation for Object Linking and Embedding.OLE

actually transfers control to the original application.Object linking and embedding(OLE) is a technology that enables the programmer of a windows-based application to create an application that can display data from many different applications and enables the user to edit that data from within the application in which it was created.When a spreadsheet is edited in Visual Basic program, actually the original application is called.OLE contains the correct underlying objects so the information is fully ediatble.In some cases, the user can even edit the data from within the Visual Basic application.The following terms and concepts are the fundamentals for understanding the methodology to use OLE in Visual Basic.

OLE Automation

Some application provide objects that support OLE Automation. We can use Visual Basic to programatically manipulate the data in these objects. Some objects that support OLE Automation also support linking and embedding. If an object in an OLEcontainer control supports OLE Automation, we can access its properties and methods using the object property.

Objects Linked

Data associated with a linked object is stored by the application that supplied the object.The application only stores link refrences that display a snapshot of the source data. When we link an object, any application containing a link to that object can access the object's data and change it. For example, if we link a text file to a Visual Basic application, the text file can be modified by the application linked to it. The modified version appears in all documents linked to this text file. We can use the OLE container control to create a linked object in our Visual Basic application.

Objects Embedded

When an embedded object is created, all the data associated with the object is contained in the object.For example, if a spreadsheet is an embedded object, all the data associated with the cells would be contained in the OLE container control or insertable object, including any necesary formulate.The name of the application that created the object is saved along with the data.If we select the embedded object while working with the Visual Basic application, the spreadsheet application can be started automatically so that we can edit those cells. When an object is embedded in an application, no other application has access to the data in the embedded object.We can use embedded objects when we want only the application to maintain data that is produced and edited in another application.

Meaning and Use of OLE container Control

An application that receices and displays an objct's data is a container application.

The OLe container control allows adding from the other application.An OLE control can have only one object at a time.Using OLE control we can

Advantages of OLE control:

- Create a placeholder in our application for an object
- Create a linked object in our application
- Bind the OLE container control to a database.
- Create objects from data that was copied onto the clipboard.
- Display objects as icons.
- Perform an action if the user moves,sizes or updates the objects in the OLE control.

- Provide backward if the user moves,sizes or updates the objects in the OLE control.

Creating Linked objects at Design Time

Each time an OLE control is drawn on a Form,an Insert Object dialogue box appears as shown in below figure which presents a list of the available objects that can be linked to or embedded in the appliucations.When an object is inserted into the OLE control at design time, the class,SourceDoc and SourceItem properites that identify the application that supplies the object, the source filename and any specific file that is linked from within that file are automatically set.The following example creates a linked object using Insert Object dialogue box:

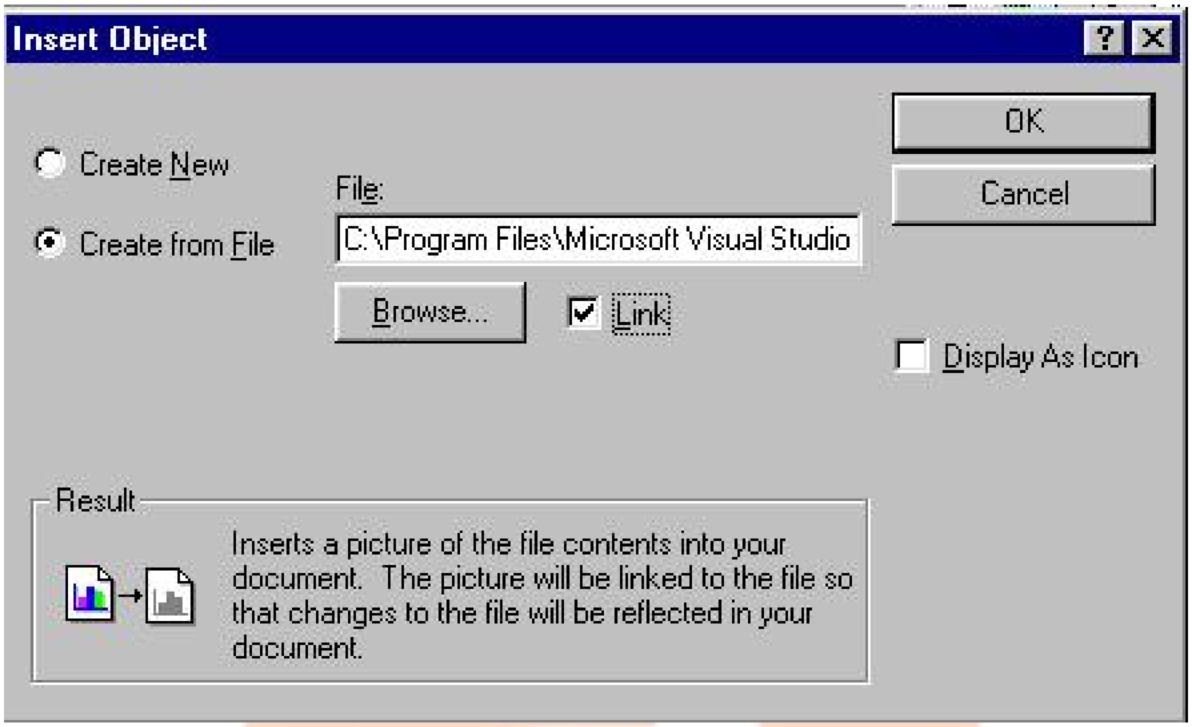
- The following example creates link from an existing application.

- Drag an OLE control on the form.The Insert object dialogue box is appeared.

- Click on the create from file option is clicked and the Browse button is chosen.A Browse dialogue box appears.

- Select the desired file from the directory and Click on OK Button. The Insert object dialogue box is displayed.

- To Link the object Click on **Link** Check Box and Click on OK button



File: C:\Program Files\Microsoft Visual Studio

Create from File [✓ Link]

Inserts a picture of the file contents into your document. The picture will be linked to the file so that changes to the file will be reflected in your document.

When a linked object is created, the data displayed in the OLE control exists in one place which is the source file. The file can be edited in its original applications and saved.The object's data can be accessed from any of the other application that contain links to that data, and the data in the source file can be changed from within any application.

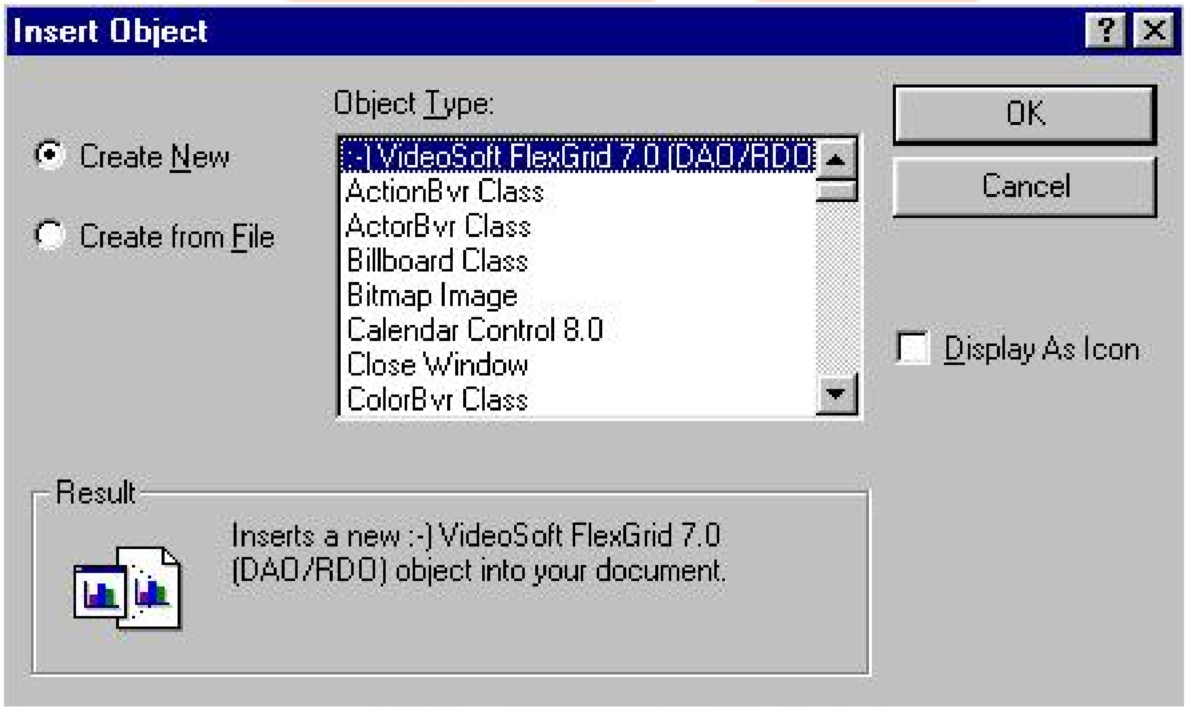
Creating embedded object at design time

The following are the steps of creating embedded object

- Draw an OLE control on the Form.
- Insert object dialogue box is appeared.
- Click on Create from File Option.And Click on browse Option.
- The desired file is selected from the directory and click on OK button.
- Click on Ok button after chossing the OK button.

When an embedded object is created, we can either embed data from a file or create a new empty object that can be filled later with data.When data is embedded from a file, a copy of the specified file's data is displayed in the OLE container control.When a new object is created, the application that created the object is invoked and data can be entered into the object.

The following Examples shows the how new file is created using OLE container:

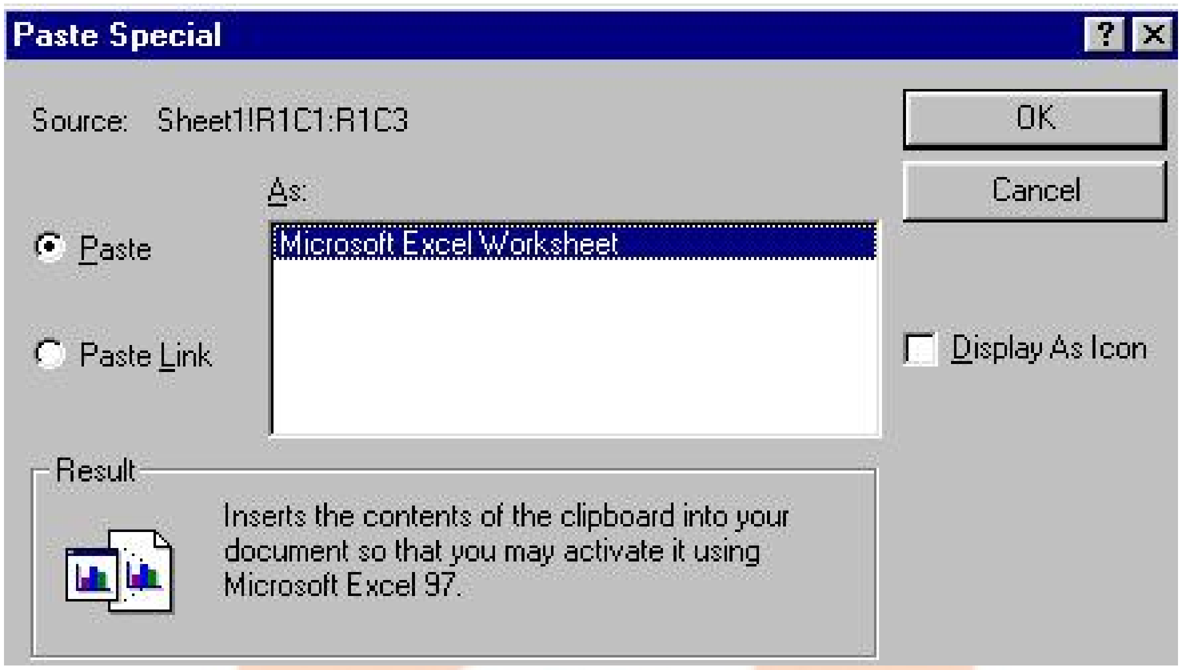


[● Create New] ○ Create from File

Inserts a new :-) VideoSoft FlexGrid 7.0 (DAO/RDO) object into your document.

To Create Objects by Paste Special Dialogue Box

Paste special Dialogue box can be used to create an object during design time.This dialogue box is useful if only a portion of a file is to be used.For e.g a paragraph from a word document, or a range of cells from an Excel Spreadsheet.



Inserts the contents of the clipboard into your document so that you may activate it using Microsoft Excel 97.

Creating Objects at Run Time

To create a linked or embedded object at run time, various methods and properties are used in the code.We can create a linked object at run time using the SourceDoc property and Create Link method.The following code fragment creates linked object at run time.

Example:

OLE1.CreateLink " C:\OLE.xls"

Empty Embedded Object Creation at run time

The CreateEmbed method can be used to specify an empty embedded object at run time.The following code fragment inserts a file for a Micorsoft Excel Worksheet in the OLE container control.

OLE1.CreateEmbed " " , "Excel.sheet"