

Visual Basic Review

SIMPLY C COMPUTERS

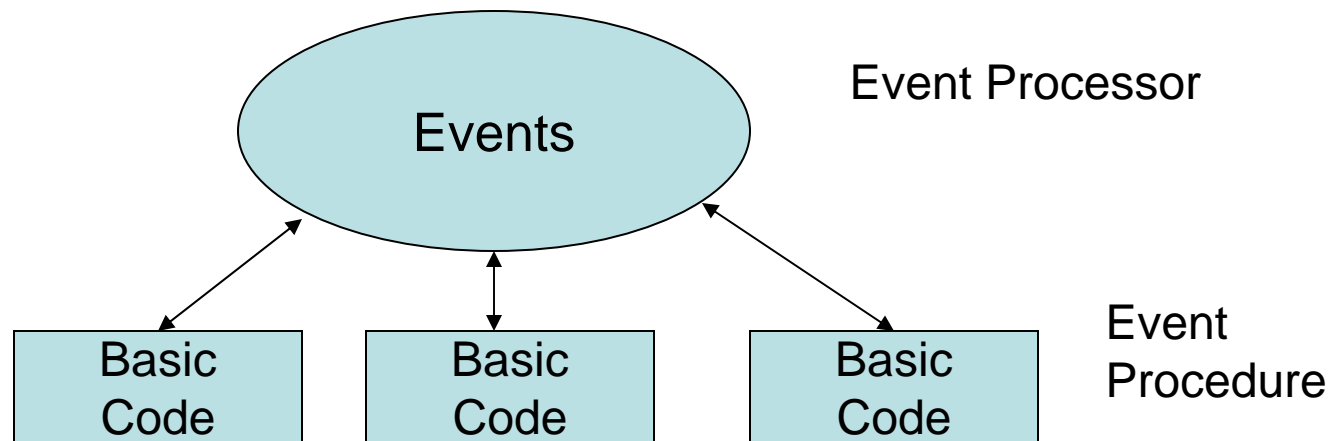
The Choice is Simple

SIMPLY C

IT Education
Web Design Company

What is Visual Basic?

- **Visual Basic** is a tool that allows you to develop Windows (Graphic User Interface - **GUI**) applications. The applications have a familiar appearance to the user.
- Visual Basic is **event-driven**, meaning code remains idle until called upon to respond to some event (button pressing, Mouse Click, Menu selection, ...). Visual Basic is governed by an event processor. Nothing happens until an event is detected. Once an event is detected, the code corresponding to that event (event procedure) is executed. Program control is then returned to the event processor.

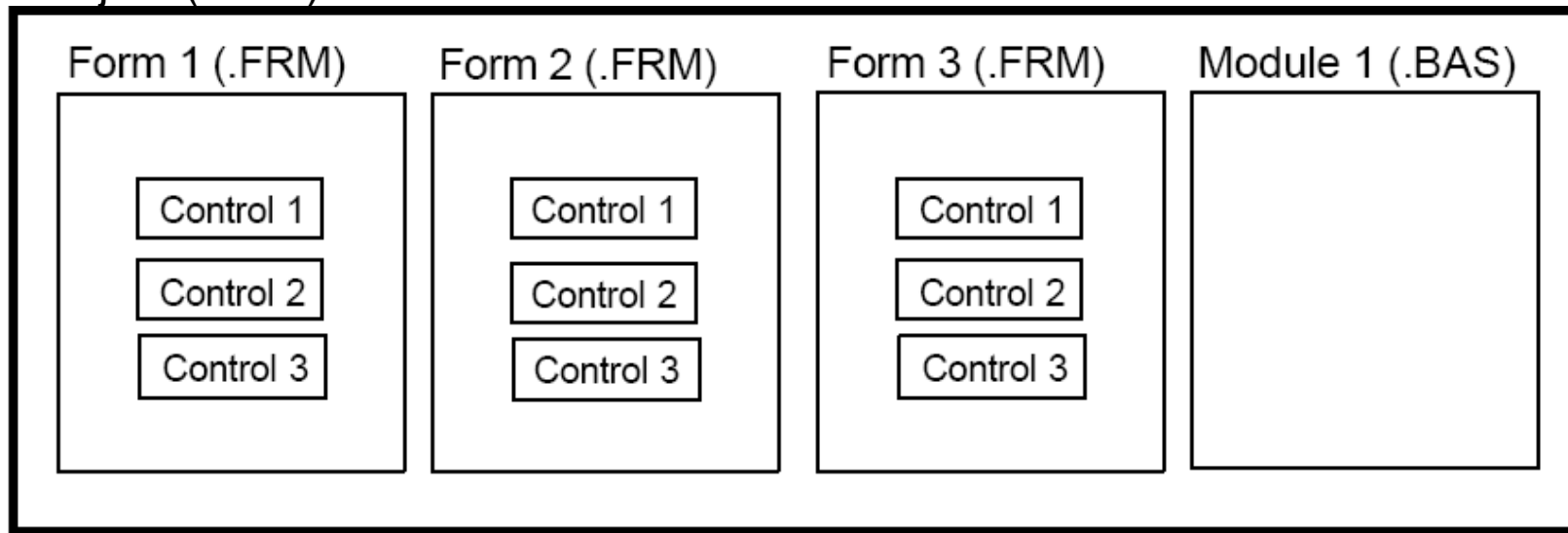


Some Important Features of Visual Basic

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

Structure of Visual Basic Application

Project (.VBP)



Application (Project) is made up of:

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

Steps in Developing Application

There are three primary steps involved in building a Visual Basic application:

1. Draw the user interface
2. Assign properties to controls
3. Attach code to controls

Drawing the User Interface and Setting Properties

Visual Basic operates in three modes.

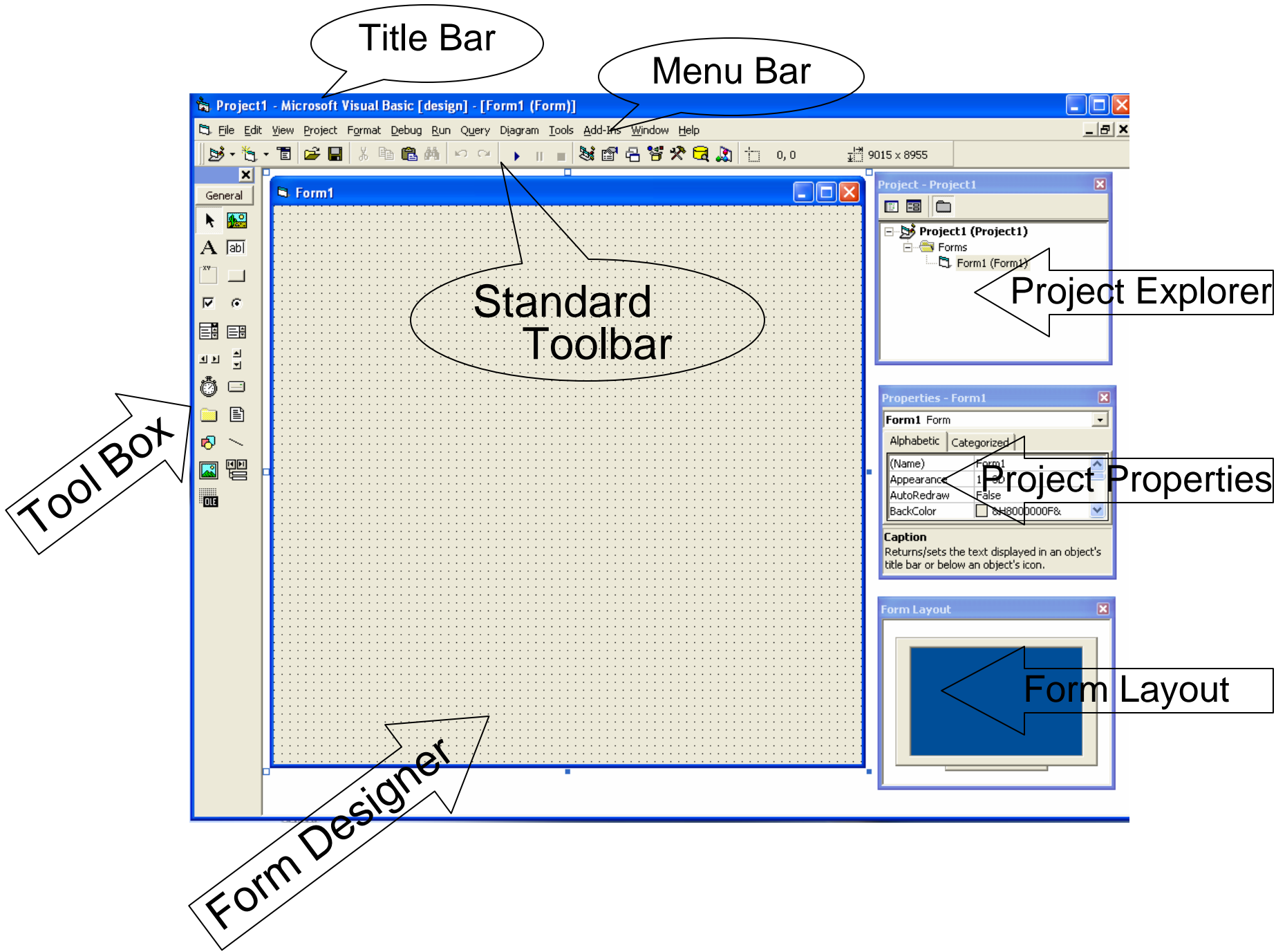
1. Design mode - used to build application
2. Run mode - used to run the application
3. Break mode - application halted and debugger is available

First We focus on the design mode.

Design Mode

Six windows appear when you start Visual Basic.

The Main Window consists of the title bar, menu bar, and toolbar. The title bar indicates the project name, the current Visual Basic operating mode, and the current form. The menu bar has drop-down menus from which you control the operation of the Visual Basic environment. The toolbar has buttons that provide shortcuts to some of the menu options. The main window also shows the location of the current form relative to the upper left corner of the screen (measured in twips) and the width and length of the current form.



Title Bar

Menu Bar

Standard
Toolbar

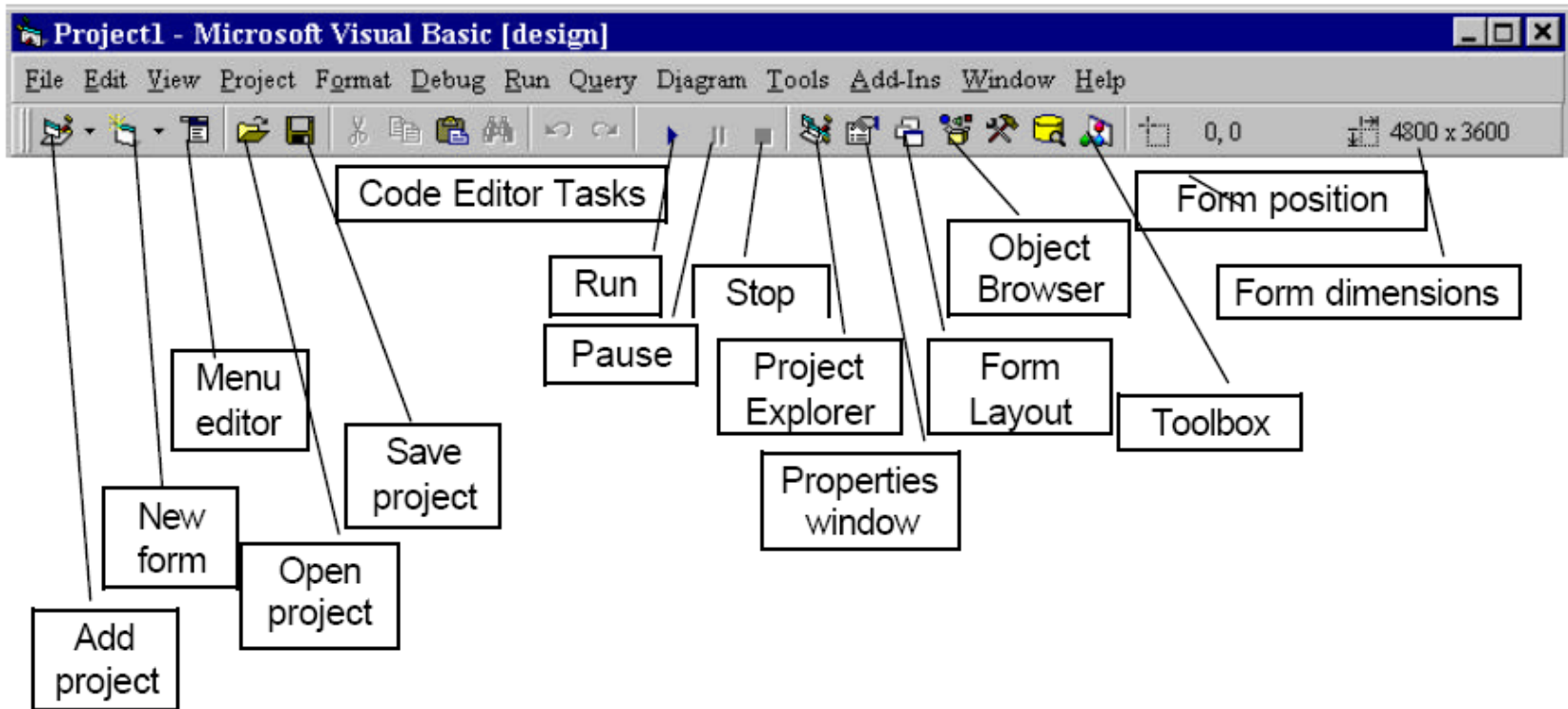
Project Explorer

Project Properties

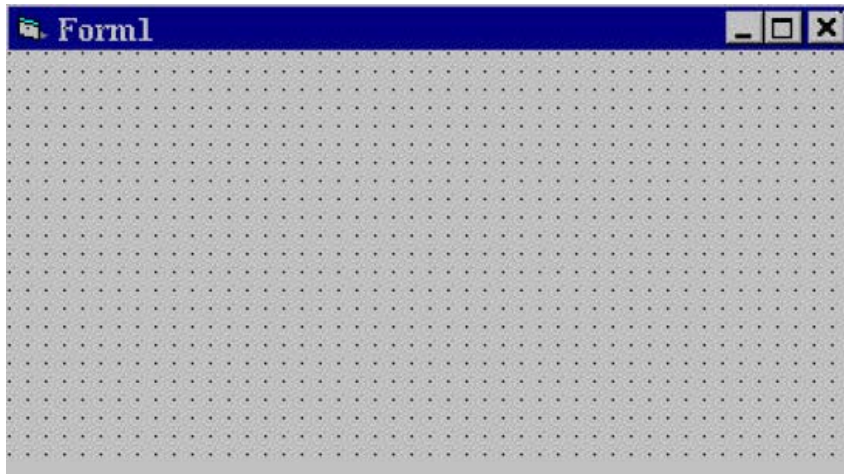
Form Layout

Tool Box

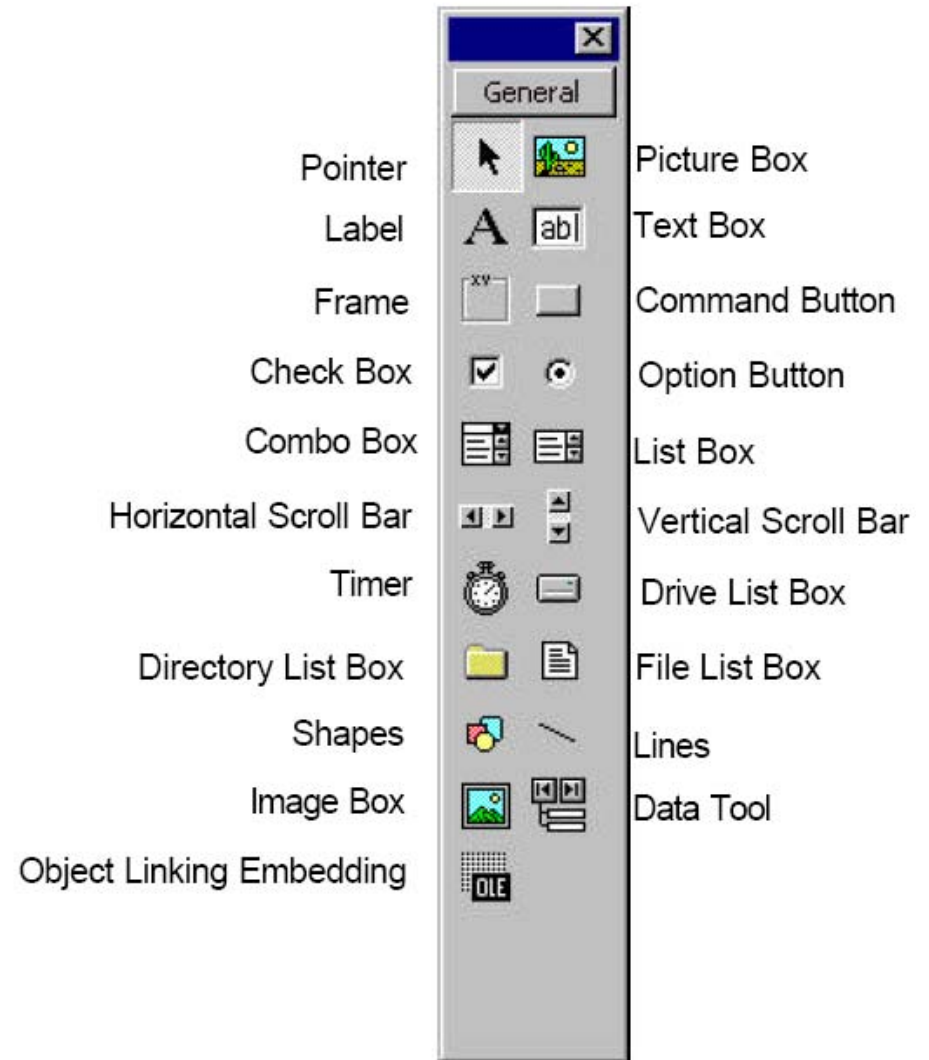
Form Designer

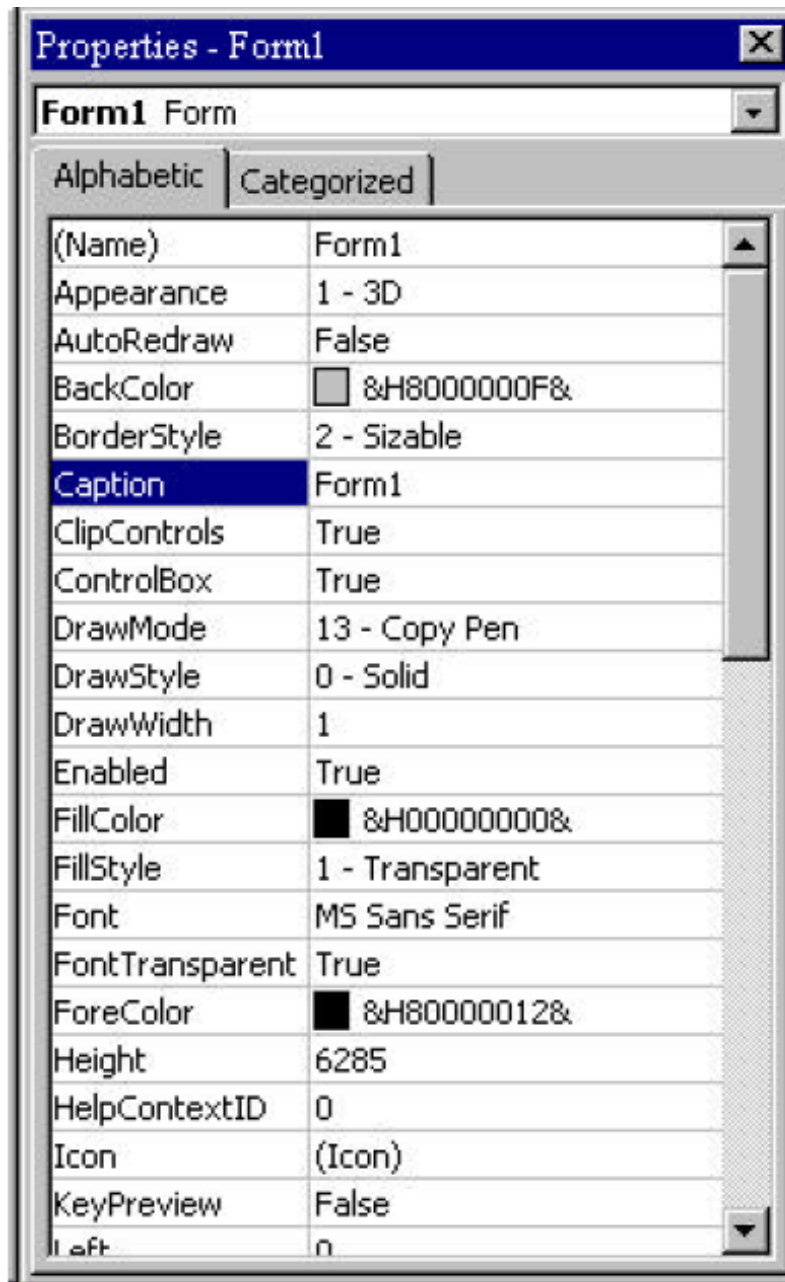


The Form Window is central to developing Visual Basic applications. It is where you draw your application.



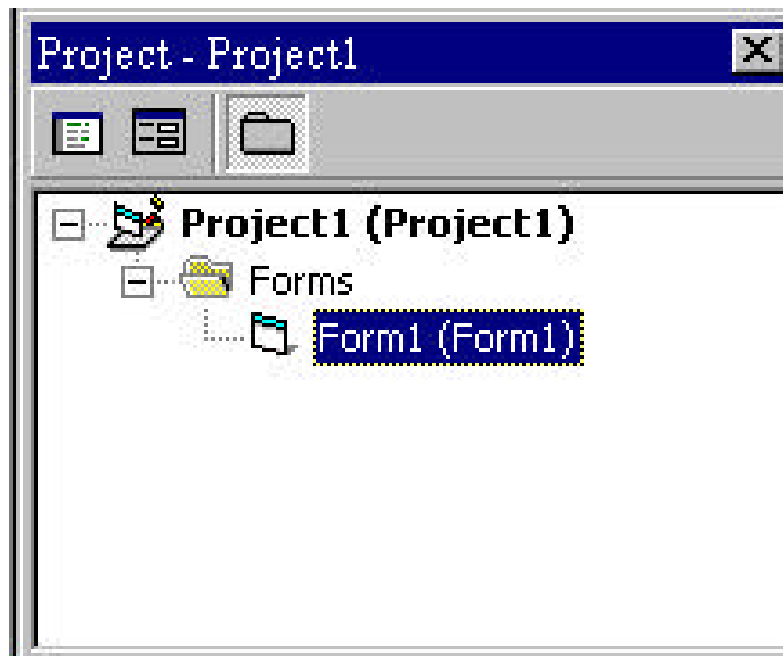
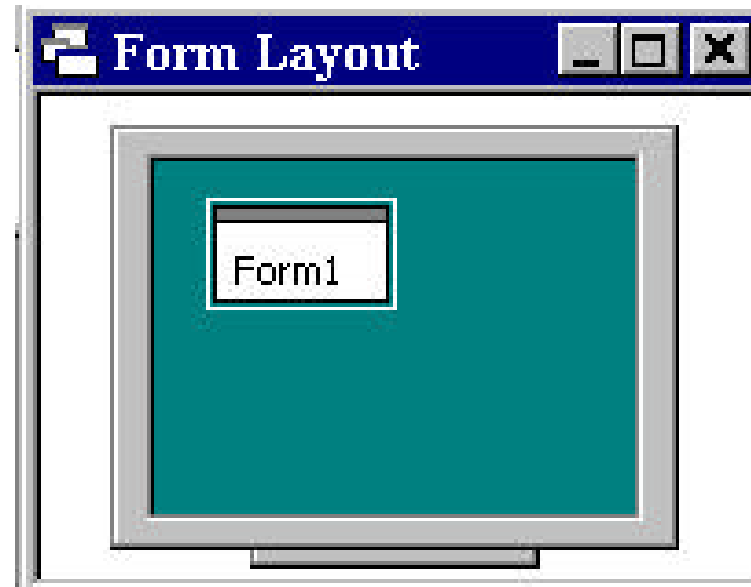
The Toolbox is the selection menu for controls used in your application.





The Properties Window is used to establish initial property values for objects. The drop-down box at the top of the window lists all objects in the current form. Two views are available: Alphabetic and Categorized. Under this box are the available properties for the currently selected object.

The Form Layout Window shows where (upon program execution) your form will be displayed relative to your monitor's screen:



The Project Window displays a list of all forms and modules making up your application. You can also obtain a view of the Form or Code windows (window containing the actual Basic coding) from the Project window.

As mentioned, the user interface is 'drawn' in the form window.

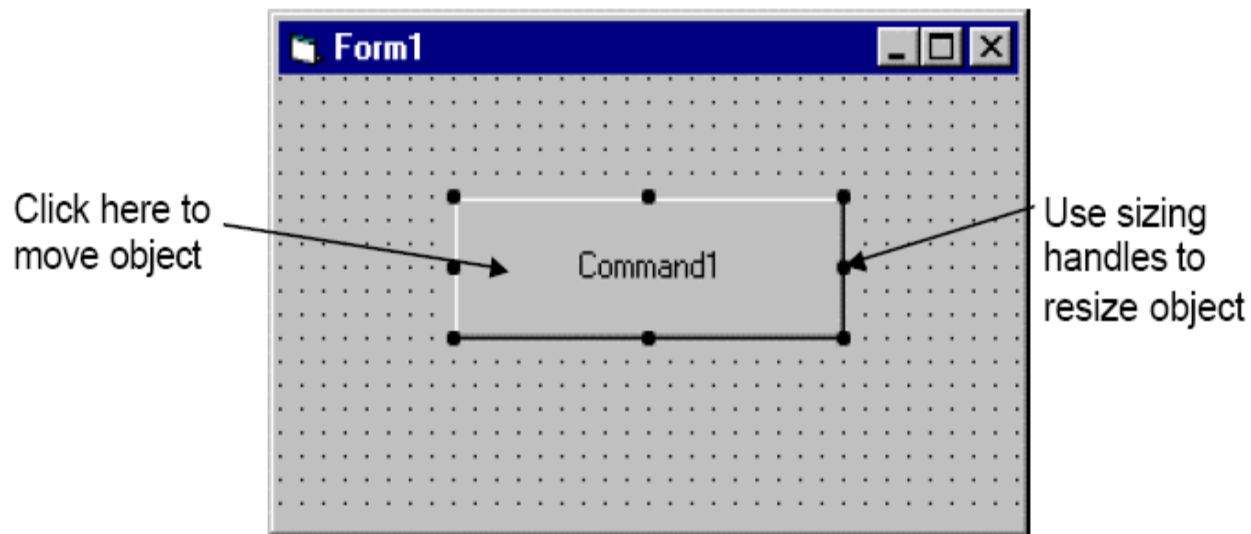
There are two ways to place controls on a form:

1. Double-click the tool in the toolbox and it is created with a default size on the form. You can then move it or resize it.

2. Click the tool in the toolbox, then move the mouse pointer to the form window. The cursor changes to a crosshair. Place the crosshair at the upper left corner of where you want the control to be, press the left mouse button and hold it down while dragging the cursor toward the lower right corner. When you release the mouse button, the control is drawn.

To move a control you have drawn, click the object in the form Window and drag it to the new location. Release the mouse button.

To resize a control, click the object so that it is selected and sizing handles appear. Use these handles to resize the object



Data Type – Numeric Type

<u>Type</u>	<u>Storage</u>	<u>Range of Values</u>
Byte	1 byte	0 to 255
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,648
Single	4 bytes	-3.402823E+38 to -1.401298E-45 for negative values 1.401298E-45 to 3.402823E+38 for positive values.
Double	8 bytes	-1.79769313486232e+308 to -4.94065645841247E-324 for negative values 4.94065645841247E-324 to 1.79769313486232e+308 for positive values.
Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	12 bytes	+/- 79,228,162,514,264,337,593,543,950,335 if no decimal is use +/- 7.9228162514264337593543950335 (28 decimal places).

Data Type – Non Numeric Type

Data Type	Storage	Range
String (fixed length)	Length of string	1 to 65,400 characters
String (variable length)	Length + 10 bytes	0 to 2 billion characters
Date	8 bytes	January 1, 100 to December 31, 9999
Boolean	2 bytes	True or False
Object	4 bytes	Any embedded object
Variant (numeric)	16 bytes	Any value as large as Double
Variant (text)	Length+22 bytes	Same as variable-length string

Data Type Suffix

Data Type	Suffix	Data Type	Suffix
Boolean	None	Integer	%
Long (Integer)	&	Single (Floating)	!
Double (Floating)	#	Currency	@
Date	None	Object	None
String	\$	Variant	None

Variables

We're now ready to attach code to our application. As objects are added to the form, Visual Basic automatically builds a framework of all event procedures. We simply add code to the event procedures we want our application to respond to. But before we do this, we need to discuss **variables**.

Variables are used by Visual Basic to hold information needed by your application. Rules used in naming variables:

- No more than 255 characters
- They not contain a space or period (.) or Hyphen (-)
- 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)
- It must be unique within the same scope. The **Scope** means the area within a program within which a variable can be used/accessed. For example within the same procedure, we can't declare two variable with same name.

Variable Declaration

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

- ❖ Default
- ❖ Implicit
- ❖ 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, while `Amount% = 300` creates an integer variable.

- There are many advantages to **explicitly** typing variables. Primarily, we insure all computations are properly done, mistyped variable names are easily spotted, and Visual Basic will take care of insuring consistency in upper and lower case Letters used in variable names. Because of these advantages, and because it is good programming practice, we will explicitly type all variables.

To **explicitly** type a variable, you must first determine its **scope**. There are four levels of scope:

- ❖ Procedure level
- ❖ Procedure level, static
- ❖ Form and module level
- ❖ Global level

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

```
Dim 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/Public** keyword:

```
Global MyInt as Integer  
Global MyDate as Date  
Public MyInt as Integer  
Public 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.

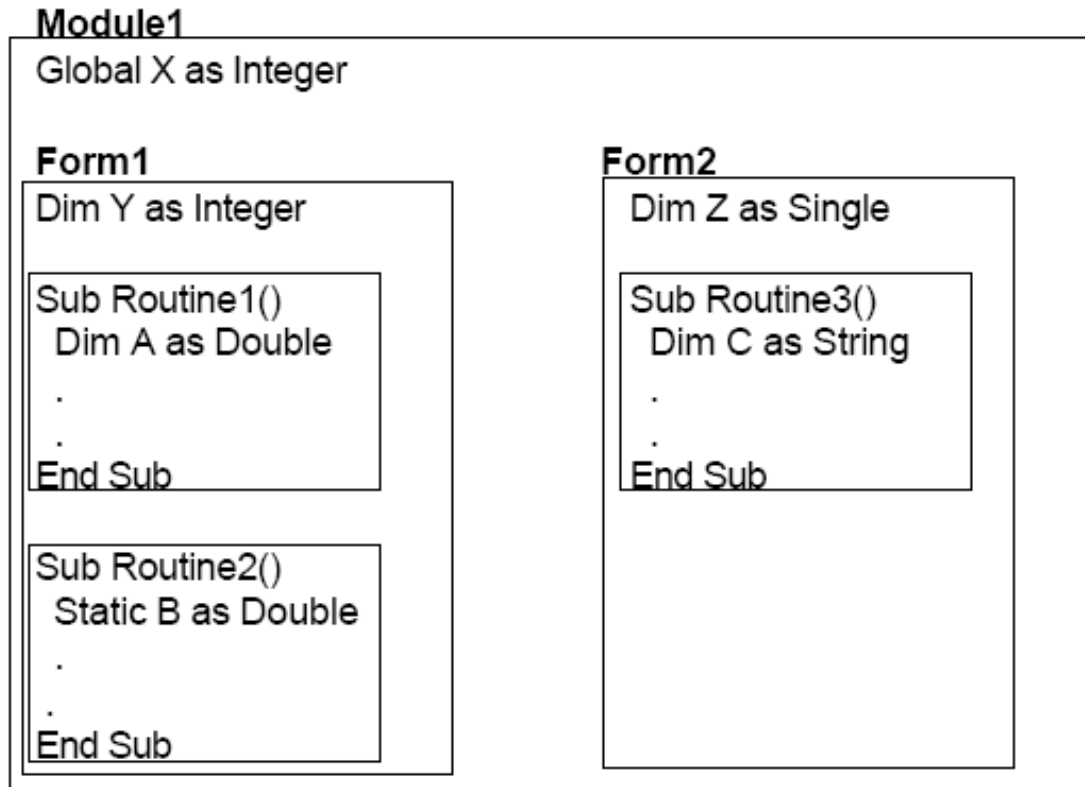
The **Option Explicit** statement forces us to declare all variables.

- Some time variable don't change their value during the execution of the program. These are constant that appear many time in our code. Declare the constant

Example –

```
Public Const pi as double=3.14159
```

Example of Variable Scope:



- Procedure Routine1 has access to X, Y, and A (loses value Upon termination)
- Procedure Routine2 has access to X, Y, and B (retains value)
- Procedure Routine3 has access to X, Z, and C (loses value)

Keyword Used to Declare The Variable: ↓	Where Declared →	General Declarations Section of a Form (.frm) Module	General Declarations Section of a Standard (.bas) Module	Sub or Function procedure of a Form or Standard Module
Dim (preferred keyword for local-, but not module-level variables)		module-level scope	module-level scope	local-level scope (value of the variable is NOT preserved between calls)
Static		not allowed	not allowed	local-level scope (value of the variable is preserved between calls)
Private (preferred keyword for module-level variables)		module-level scope	module-level scope	not allowed
Public		project-level scope (but references to the variable must be qualified with the form name; also there are some minor restrictions on the types of variables that can be declared as public in a form)	project-level scope	not allowed
Global (the use of this keyword is discouraged; it remains only for compatibility with older versions of VB)		not allowed	project-level scope	not allowed

Visual Basic Statements & Expression

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

Examples:

StartTime = Now

Explorer.Caption = "Captain Spaulding"

BitCount = ByteCount * 8

Energy = Mass * LIGHTSPEED ^ 2

NetWorth = Assets – Liabilities

The assignment statement stores information.

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

StartTime = Now : EndTime = StartTime + 10

(Be careful stacking statements, especially with If/End If structures. You may Not get the response you desire.)

- If a statement is very long, it may be continued to the next line using the **continuation** character, an underscore (_). Example:
Months = Log(Final * IntRate / Deposit + 1) _
/ Log(1 + IntRate)
-

Comment statements begin with the keyword **Rem** or a single quote ('). For example:

Rem This is a remark

' This is also a remark

x = 2 * y ' another way to write a remark or comment

Visual Basic Operator

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

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

- **Parentheses** around expressions can change precedence.
- To **concatenate** two strings, use the **&** symbol or the **+** symbol:
- There are six **comparison** operators in Visual Basic: **Operator**

Comparison

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to

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

Not	Logical NOT
And	Logical AND
Or	Logical OR
XOR	Logical XOR
- 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.

Control Statements:

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

- The simplest is the **If/Then** statement:

If condition then process

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

If condition then

Statement 1

Statement 2

End if

- Or, **If/Then/Else/End If** blocks:

If condition then

Statement 1

Else

Statement 2

End if

- Or, we can add the **Elseif** statement:

If condition then

Statement 1

Elseif condition

Statement 2

Else

Statement 3

End if

- **Select Case - Another Way to Branch**

Select Case Age

Case 5

Category = "Five Year Old"

Case 13 To 19

Category = "Teenager"

Case 20 To 35, 50, 60 To 65

Category = "Special Adult"

Case Is > 65

Category = "Senior Citizen"

Case Else

Category = "Everyone Else"

End Select

Visual Basic Looping

- 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  
Statement 1  
Counter = Counter + 1  
Loop
```

- **Do Until/Loop** Example:

```
Counter = 1  
Do Until Counter > 1000  
Statement 1  
Counter = Counter + 1  
Loop
```

- **Do/Loop While** Example:

```
Sum = 1  
Do  
statements  
Sum = Sum + 3  
Loop While Sum <= 50
```

- **Do/Loop Until** Example:

```
Sum = 1  
Do  
statements  
Sum = Sum + 3  
Loop Until Sum > 50
```

- Counting is accomplished using the

For/Next loop

For I = 1 to 50 Step 2

statements

Next I

The statement **Exit Do** will get you out of a loop and transfer program control to the statement following the Loop statement.

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

The Message Box

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

MsgBox Message, Type, Title

where

Message Text message to be displayed

Type Type of message box (discussed in a bit)

Title Text in title bar of message box

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

Style Values

Style Value	Named Constant	Buttons Displayed
0	vbOkOnly	Ok button
1	vbOkCancel	Ok and Cancel buttons
2	vbAbortRetryIgnore	Abort, Retry and Ignore buttons.
3	vbYesNoCancel	Yes, No and Cancel buttons
4	vbYesNo	Yes and No buttons
5	vbRetryCancel	Retry and Cancel buttons

We can use named constant in place of integers for the second argument to make the programs more readable. In fact, VB6 will automatically shows up a list of names constant where you can select one of them.

Example:

```
yourMsg=MsgBox( "Click OK to Proceed", 1, "Startup Menu")
```

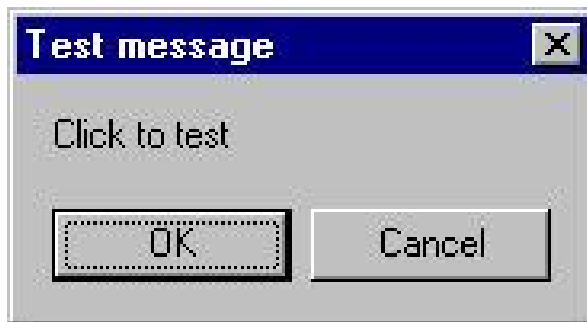
and

```
yourMsg=Msg("Click OK to Proceed". vbOkCancel,"Startup Menu")
```

yourMsg is a variable that holds values that are returned by the MsgBox () function. The values are determined by the type of buttons being clicked by the users. It has to be declared as Integer data type in the procedure or in the general declaration section. **Below Table** shows the values, the corresponding named constant and buttons.





Return Values and Command Buttons

Value	Named Constant	Button Clicked
1	vbOk	Ok button
2	vbCancel	Cancel button
3	vbAbort	Abort button
4	vbRetry	Retry button
5	vbIgnore	Ignore button
6	vbYes	Yes button
7	vbNo	No button



```
testmsg = MsgBox("Click to test", 1, "Test message")  
If testmsg = 1 Then  
    Statement 1  
Else  
    statement 2  
End If
```

- To make the message box look more sophisticated, you can add an icon besides the message. There are four types of icons available in VB as shown in [below Table](#)

Value	Named Constant	Icon
16	vbCritical	
32	vbQuestion	
48	vbExclamation	
64	vbInformation	



```
testMsg2 = MsgBox("Click to Test", vbYesNoCancel + vbExclamation,  
"Test Message")  
If testMsg2 = 6 Then  
    Statement 1  
Elseif testMsg2 = 7 Then  
    Statement 2  
Else  
    Statement 3  
End If
```

The InputBox() Function

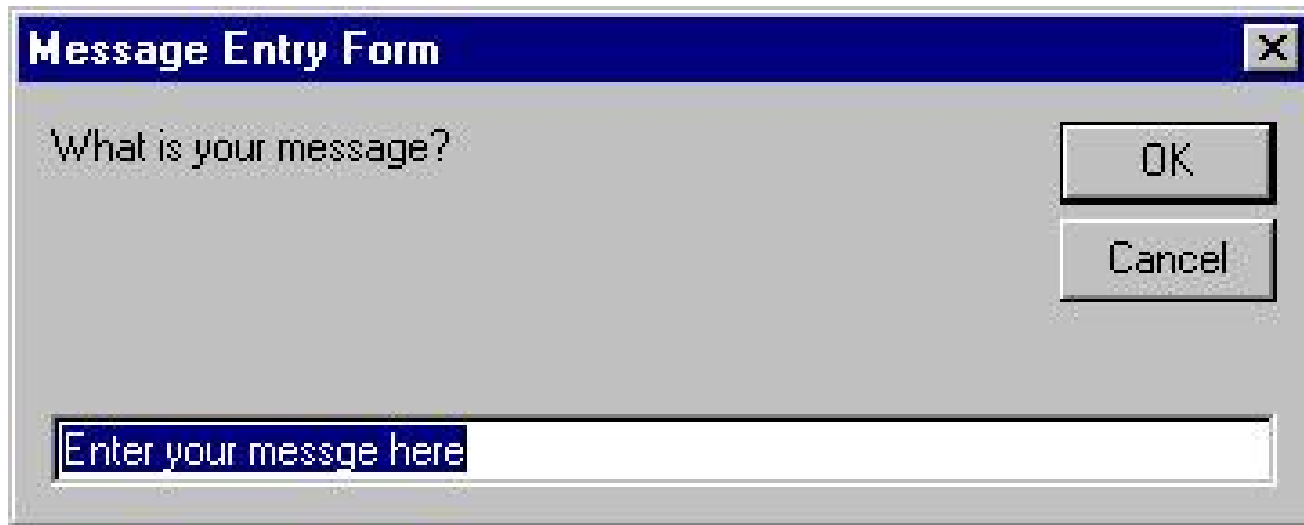
An InputBox() function will display a message box where the user can enter a value or a message in the form of text. The format is

myMessage=InputBox(Prompt, Title, default_text, x-position, y-position)

myMessage is a variant data type but typically it is declared as string, which accept the message input by the users. The arguments are explained as follows:

- Prompt - The message displayed normally as a question asked.
- Title - The title of the Input Box.
- default-text - The default text that appears in the input field where users can use it as his intended input or he may change to the message he wish to key in.
- x-position and y-position - the position or the coordinate of the input box.

```
userMsg = InputBox("What is your message?", "Message Entry Form",  
"Enter your messge here", 500, 700)
```



Array

By definition, an array is a list of variables, all with the same data type and name. When we work with a single item, we only need to use one variable. However, if we have a list of items which are of similar type to deal with, we need to declare an array of variables instead of using a variable for each item.

- ❖ An array can be one dimensional or multidimensional
- ❖ The format for a one dimensional array is `ArrayName(x)`
- ❖ The format for a two dimensional array is `ArrayName(x,y)`

We could use `Public` or `Dim` statement to declare an array just as the way we declare a single variable.

`Dim arrayName(subs) as dataType`

`Dim CusName(10) as String` - variable contain 11 elements from position 0 to 10

We can change the option base (starting value) by the given syntax

`Dim CusName(1 to 10) as String` – variable contain 10 elements from position 1 to 10

The general form of Two-Dimensional Array

`Dim arrayName(sub1,sub2) as dataType`

`Dim sname(10,10) as string`

`Dim sname(1 to 10, 1 to 10) as string`

- ❖ `Lbound (arrayname)` – gives the Lower bound of an array
- ❖ `Ubound(arrayname)` - gives the Upper bound of an array
- ❖ `Lbound(arrayname,Dimension)` - gives the Lower bound of 2 D array
Dimension Value - 1 for Row , 2 for column
- ❖ `Ubound(arrayname,Dimension)` - gives the Upper bound of 2 D array

❖ Dynamic Array

Some time we may not know how large to make an array, so we can declare the dynamic array

To create the dynamic array, declare its as usual with dim statement (or Private or Public)

`Dim dynarray()`

Later in the program, when we know how many elements we want to store in the array, use the `ReDim` statement to `redimension` the array

`ReDim dynarrya(size)`

- ❖ Redim can't change the type of data.

❖ Preserve Keyword

Each time we execute the Redim statement, all the values currently stored in the array are lost. By use Preserve keyword, we use the old data after the ReDim statements.

`ReDim Preserve Dynarray(size)`

Procedure

A large Application broken into small segments called procedures. In VB there are two types of procedures

1. Subroutine
2. Functions

Procedures are useful for implementing repeated task, such as frequently used calculation.

❖ Subroutine –

A subroutine is a block of statements that carry out a well defined task. The block of statement placed with a pair of **Sub/End Sub** statements and can be invoked by name. Example

```
Sub showmessage()  
Msgbox "Subroutine Invoked"  
End Sub
```

Calling of Subroutine – To call a procedures, use the call statement and supply the arguments

Call subroutinename (arg1, arg2, arg3.....)

Or

Subroutine name arg1, arg2, arg3.....

The number of arguments we supply to the subroutine and their type must match those in the procedure declaration.

❖ Function

A function is similar to a subroutine, but a function return a result.

Subroutine perform a task and don't return anything to the calling program; function commonly carry out calculation and return the result.

Function sum() as integer

Sum=5+2

End Function

Function sum return the value to the calling program. The value which return by function it must be stored into the name of **Function**

Calling of Function – Functions are called by name

Var=functionname(arg1,arg2.....)

❖ Arguments

Procedure and Function both accept the arguments from the calling program. So by the arguments the procedures and functions communicate with the rest of the application.

```
Sub Sum(a as integer, b as integer)
```

```
Dim s as integer
```

```
S=a + b
```

```
Msgbox S
```

```
End Sub
```

```
Call sum(10, 20)
```

```
Function Sum(a as integer, b as integer)
```

```
Sum=a + b
```

```
End Function
```

```
S=sum(10, 20)
```

❖ Argument – Passing Mechanism

The default mechanism of passing an arguments by Reference. The other mechanism is passing by value (use keyword **BYVal**)

- ❖ In case of passing by reference, pass the address of the variable in memory so that the procedure can change its value permanently.

Passing by Reference

```
Sub Sum(a as integer, b as integer)
```

```
Dim s as integer
```

```
S=a+b
```

```
a=0
```

```
b=0
```

```
End Sub
```

```
Call sum(X, Y)
```

After calling the value of variable X and Y as zero.

Passing by Value

```
Sub Sum(ByVal a as integer, ByVal b as integer)
```

```
Dim s as integer
```

```
S=a+b
```

```
a=0
```

```
b=0
```

```
End Sub
```

```
Call sum(X, Y)
```

After calling the value of variable X and Y retain same.

- ❖ When we pass the arguments to a procedures by reference, we are actually passing the variable itself. Any changes made to the arguments by the procedure. When we pass the arguments by value, the procedure gets a copy of the variable, which is discarded when the procedure ends.

❖ Using Optional Arguments

We also specify the optional argument i.e an arguments ar not compulsory to pass at the time of calling. This is done by using the keyword **Optional**

```
Sub Sum(a as integer, optional b as integer)
```

```
Dim s as integer
```

```
S=a+b
```

```
a=0
```

```
b=0
```

```
End Sub
```

Call sum(X, Y) or call sum (X)

After calling the value of variable X and Y as zero.

❖ Ismissing Function

By this function we can check the missing arguments. It return True when argument is missing otherwise False.

❖ Default value for an Optional Argument

We can also specify the default value for an optional argument.

```
Sub Sum(a as integer, optional b as integer=10)
```

❖ Passing an Unknown number of arguments

Sometime we don't know the number of arguments, so we use the

ParamArray Keyword which allow us to pass a variable number of arguments

```
Sub Sum (ParamArray a() )
```

```
Dim s as variant
```

```
For each x in a
```

```
s=s + a
```

```
Next
```

```
Msgbox s
```

```
End Sub
```

```
Call sum (10,20,30.....)
```

Form Object

- The **Form** is where the user interface is drawn. It is central to the development of Visual Basic applications.
- **Some Basic Properties –**
 - **Appearance** Selects 3-D or flat appearance.
 - **BackColor** Sets the form background color.
 - **BorderStyle** Sets the form border to be fixed or sizeable.
 - **Caption** Sets the form window title.
 - **Enabled** If True, allows the form to respond to mouse and keyboard events; if False, disables form.
 - **Font** Sets font type, style, size.
 - **ForeColor** Sets color of text or graphics.
 - **Picture** Places a bitmap picture in the form.
 - **Visible** If False, hides the form.

- **Form Events–**

Activate Form_Activate event is triggered when form becomes the active window.

Click Form_Click event is triggered when user clicks on form.

DbIcIck Form_DbIcIck event is triggered when user double clicks on form.

Load Form_Load event occurs when form is loaded. This is a good place to initialize variables and set any runtime properties.

- **Form Methods-**

Cls Clears all graphics and text from form. Does not clear any objects.

Print Prints text string on the form.

Show Show the Form (vbModeless / vbmodal)

Hide Hide the form

Use the any property / method or event of any object or control
Nameofobject.property/method nameofobject_event

- Discuss some common event

Mouse Move	Mouse Up	Mouse Down
Click	DbIClick	getfocus
lostfocus	keypress	keyup
keydown		

- Common Constant

Vbleftbutton	-	1
Vbrightbutton	-	2
Vbmiddlebutton	-	4
Vbshiftmask	-	1
Vbctrlmask	-	2
Vbaltmask	-	4

Label

A **label box** is a control you use to display text that a user can't edit directly. We've seen, though, in previous examples, that the text of a label box can be changed at run-time in response to events.

Label Properties:

Alignment	Aligns caption within border.
Appearance	Selects 3-D or flat appearance.
AutoSize	If True, the label is resized to fit the text specified by the caption property. If False, the label will remain the size defined at design time and the text may be clipped.
BorderStyle	Determines type of border.
Caption	String to be displayed in box.
Font	Sets font type, style, size.

Label Events:

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

Text Box

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

Text Box Properties:

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

Text Box Events:

Change

Triggered every time the **Text** property changes.

LostFocus

Triggered when the user leaves the text box. This is a good place to examine the contents of a text box after editing.

KeyPress

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

Text Box Methods:

SetFocus

Places the cursor in a specified text box.

** Discuss Some Problem Based on Text Box

Some Important String Function

Asc	Determine the ASCII Value of a String character
Chr	Translate the ASCII Value in to a String Character
Len	Determine the length of String
Ltrim	Truncate any leading space from a string
Rtrim	Truncate any following space from a string
Trim	Truncate any leading and trailing space from a string
Left	Return a Specified number of character of a string, starting from the leftmost – <code>left (String, number of character)</code>
Right	Return a specified number of character of a string, starting from the rightmost - <code>right (String, number of character)</code>
Mid	Return the specified portion of a string expression – <code>mid(String,start , [length])</code>
Lcase	Translate all upper case character in a string to lowercase
Ucase	Translate all lower case character in a string to uppercase
Str	Convert the numeric value into String
Strcomp	Return a value indicating the result of a string comparison. values are -1, 0, 1
Strreverse	Returns a string in which the character order of a specified string is reversed – <code>string=strreverse(string)</code>

- Instr** Find the character position of one string into another
`var=instr([start position],string1,string2,[compare])`
value of var =0 means Not Found otherwise return the position
- Instrrev** Find the character position of one string into another, from the end of the string to the beginning
`var=instrrev(string1,string2,[startpos],[compare])`
value of var =0 means Not Found otherwise return the position
- Replace** Its substitute a specified string value within a given string for a specified number of times
`var=replace(String, Find, Replace, [start],[count],[compare])`
- Join** Its returns a string created by concatenating all the values within an array – `join (listarray,[delimiter])`
- Filter** Its search through a given string array for a specified filter criterion and returns the value of zero based array
`Filter(inputstring,findvalue,[include/exclude],compare)`
- Split** its divide into substring based on a specified delimiter.
`split (string, [delimiter])`

Some Important Maths Function

Abs	Return the absolute value of a numeric expression
Exp	Return the natural logarithmic base (e) to the specified power e^x
Log	Return the natural Logarithm
Cos	Return the cosine of a numeric expression
Sin	Return the sine of a numeric expression
Tan	Return the tangent of a number
Atan	Return the arctangent of a numeric expression
Sqr	Return the square root of a positive number
Rnd	Return a randomly generated number between 0 to 1
Randomize	its statement, seed the random number generator. Its must be used before the Rnd function. Although Randomize ca be used several times throughout the program, it is best if it is used only once. By default its use the system time.
Hex	Convert into Hexadecimal Number
Oct	Convert into Octal Number
Val	Return the number containing string into numeric value
Int	Return the largest integer that is less than or equal to a number
Fix	Truncate the fractional part of the numeric expression
CInt	Convert to Integer

Value	Fix	Int	Cint
2.7	2	2	3
2.2	2	2	2
2	2	2	2
-2	-2	-2	-2
-2.2	-2	-3	-2
-2.7	-2	-3	-3

Format Its format the numeric Expression
format(Expression, [Format])

- 0 - Display a digit or a zero
- # - Display a digit or nothing

Format(123, "00000.00")	00123.00
Format(123, "#####.##")	123. / 123.45
Format(123, ""##,##0.00")	1,234.00
Format(12345, "###e+")	12e+3

Some Important Date Time Function

Now	Return the current date and time of the system
Date	Return / Set the current date of the system
Time	Return / Set the current time of the system
Month	Return the integer between 1 and 12
Day	Return the integer between 1 and 31
Year	Return the integer between 100 and 9999
Monthname	Return a string indicating the specified month
Weekday	Return an integer between 1 and 7 (represent the day of week by default Sunday is the first day of the week)
Weekdayname	Returns a string indicating the specified day of the week
Dateserial	Convert the numeric value in to date format Dateserial(year%,month%,day%)
Datevalue	Convert the String into date format
Hour	Return an integer between 0 and 23
Min	Return an integer between 0 to 59
Second	Return an integer between 0 to 59
Timeserial	Convert the numeric value in to time format timeserial(hour%,min%,sec%)
Timevalue	Convert the string in to time format
Formatdatetime	its used to format the date and time

Command Button

It is probably the most widely used control. It is used to begin, interrupt, or end a particular process.

Command Button Properties:

Cancel	Allows selection of button with Esc key (only one button on a form can have this property True).
Caption	String to be displayed on button.
Default	Allows selection of button with Enter key (only one button on a form can have this property True).
Enable	Control Respond or Not (True / False)
Font	Sets font type, style, size.
Name	The string value used to refer to the control in code
Picture	Specifies the graphic to be displayed in the control
TabIndex	Specifies the tab order of a control within its parent form
Tabstop	Specifies whether or not the user can use the Tab key to give focus to the control.
Visible	Specifies whether the control is visible or hidden,

Command Button Events:

- Click** Event triggered when button is selected either by clicking on it or by pressing the access key.
- Gotfocus** An object receives focus
- Lostfocus** An object loses focus.

Command Button Method:

- Setfocus** Gives focus to the object specified in the method call

Check Box

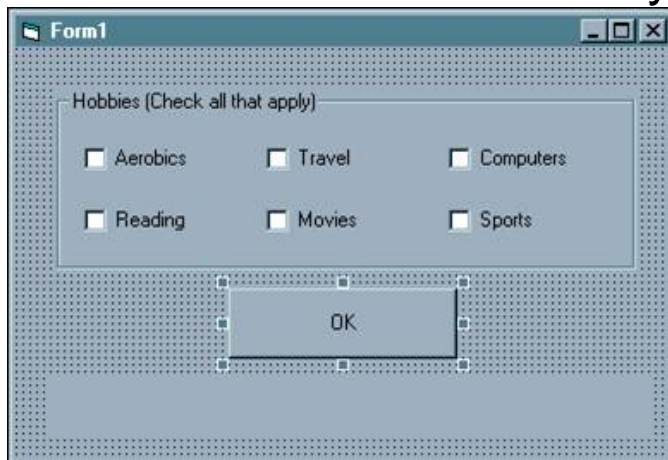
Check boxes provide a way to make choices from a list of potential candidates. Some, all, or none of the choices in a group may be selected.

Check Box Properties:

Caption	Identifying text next to box.
Font	Sets font type, style, size.
Value	Indicates if unchecked (0, vbUnchecked), checked (1, vbChecked), or grayed out (2, vbGrayed).

Check Box Events:

Click	Triggered when a box is clicked. Value property is automatically changed by Visual Basic.
--------------	---



Option Button

Option buttons provide the capability to make a mutually exclusive choice among a group of potential candidate choices. Hence, option buttons work as a group, only one of which can have a True (or selected) value.

Option Button Properties:

Caption	Identifying text next to button.
Font	Sets font type, style, size.
Value	Indicates if selected (True) or not (False). Only one option button in a group can be True. One button in each group of option buttons should always be initialized to True at design time.
Style	Normal / Graphical

Option Button Events:

Click	Triggered when a button is clicked. Value property is automatically changed by Visual Basic.
--------------	---

Control Arrays

- With some controls, it is very useful to define **control arrays** - it depends on the application. For example, option buttons are almost always grouped in control arrays.
- Control arrays are a convenient way to handle groups of controls that perform a similar function. All of the events available to the single control are still available to the array of controls, the only difference being an argument indicating the index of the selected array element is passed to the event. Hence, instead of writing individual procedures for each control (i.e. not using control arrays), you only have to write one procedure for each array.
- Once a control array has been created and named, elements of the array are referred to by their name and index.

Example – If u create the control array of text box –

```
Private Sub Text1_Change(Index As Integer)
```

```
End Sub
```

```
Text1(index).property
```

Frame Control

- **Frames** provide a way of grouping related controls on a form. And, in the case of option buttons, frames affect how such buttons operate.
- To group controls in a frame, you first draw the frame. Then, the associated controls must be drawn in the frame. This allows you to move the frame and controls together. And, once a control is drawn within a frame, it can be copied and pasted to create a control array within that frame.
- Option buttons within a frame work as a **group**, independently of option buttons in other frames. Option buttons on the form, and not in frames, work as another independent group. Remember physical location, and physical location only, dictates independent operation of option button groups.

Frame Properties:

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

Pizza Order

Size

- Small
- Medium
- Large

Crust Type

- Thin Crust
- Thick Crust

Toppings

- Extra Cheese
- Onions
- Mushrooms
- Green Peppers
- Black Olives
- Tomatoes

Eat In Take Out

Build Pizza Exit

List Box

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

List Box Properties:

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

List Box Events:

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

List Box Methods:

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

Combo Box

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

Combo Box Properties:

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

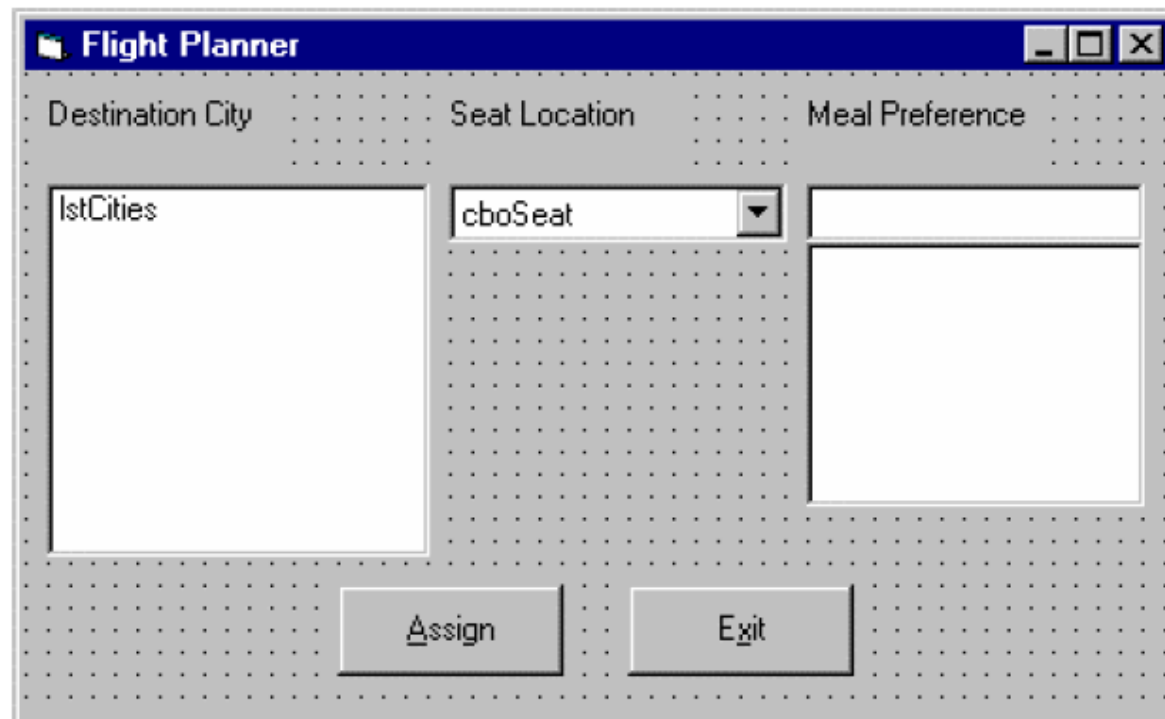
Appearance	Selects 3-D or flat appearance.
List	Array of items in list box portion.
ListCount	Number of items in list.
ListIndex	The number of the most recently selected item in list. If no item is selected, ListIndex = -1.
Sorted	True means items are sorted in 'Ascii' order, else items appear in order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

Combo Box Events:

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

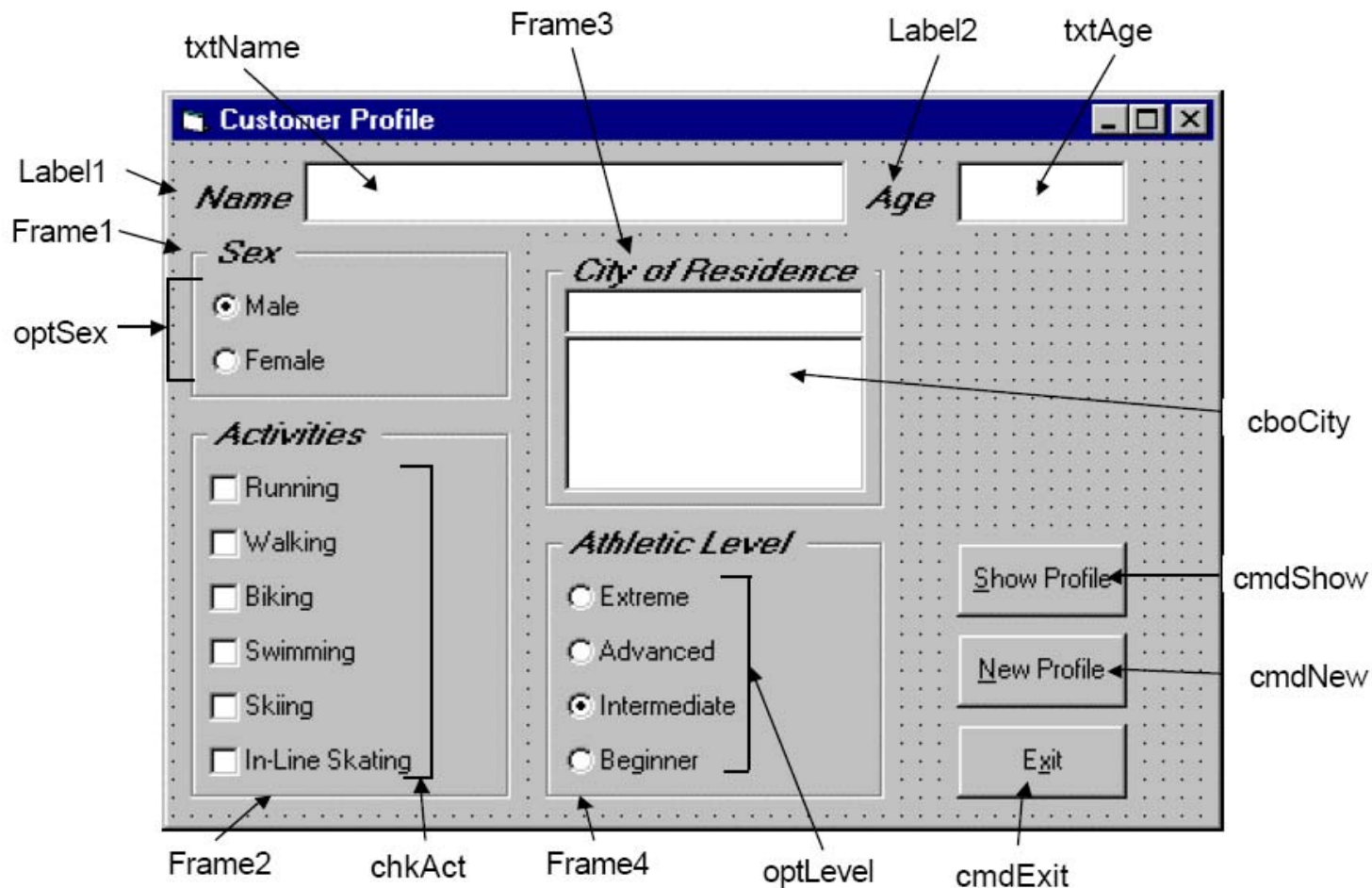
Combo Box Methods:

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



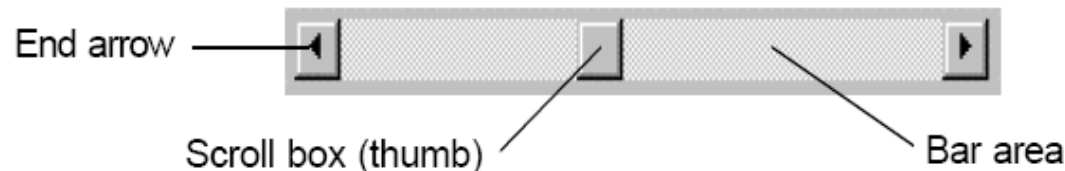
Problem - A new sports store wants you to develop an input screen for its customer database. The required input information is:

1. Name
2. Age
3. City of Residence
4. Sex (Male or Female)
5. Activities (Running, Walking, Biking, Swimming, Skiing and/or In-Line Skating)
6. Athletic Level (Extreme, Advanced, Intermediate, or Beginner)



Horizontal & Vertical Scroll Bar

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



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

Scroll Bar Properties:

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

Note that although the extreme values are called **Min** and **Max**, they do not necessarily represent minimum and maximum values. There is nothing to keep the Min value from being greater than the Max value.

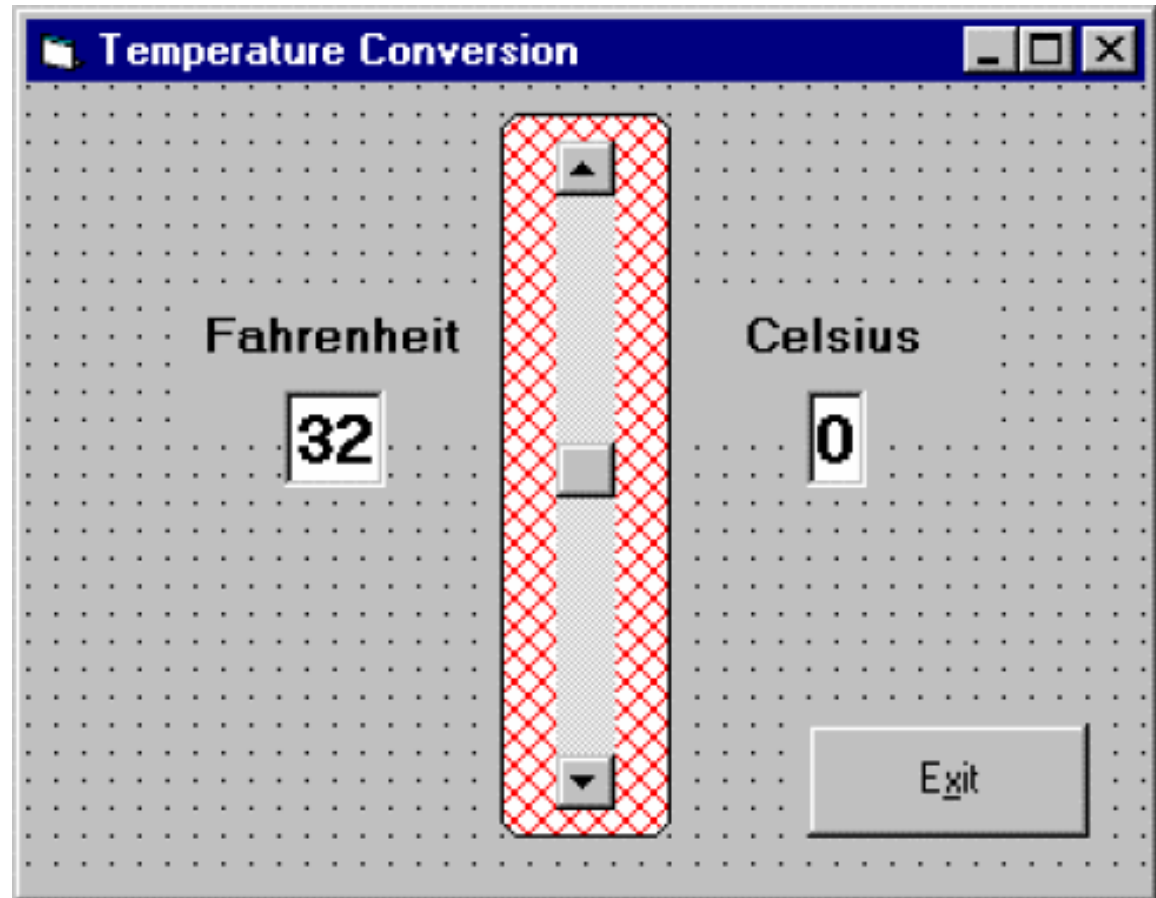
Scroll Bar Events:

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

Problem - Temperature Conversion Application

Shape1:

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



Timer Control

- Many times, especially in using graphics, we want to repeat certain operations at regular intervals. The **timer tool** allows such repetition. The timer tool does not appear on the form while the application is running.
- Timer tools work in the background, only being invoked at time intervals you specify. This is multi-tasking - more than one thing is happening at a time.

Timer Properties:

- Enabled** Used to turn the timer on and off. When on, it continues to operate until the Enabled property is set to False.
- Interval** Number of milliseconds between each invocation of the Timer Event.

Timer Events:

The timer tool only has one event, **Timer**. It has the form:

```
Sub TimerName_Timer()  
Statement 1  
Statement 2  
End Sub
```

** STOP Watch Application

Picture Box

- The **picture box** allows you to place graphics information on a form. It is best suited for dynamic environments - for example, when doing animation.

Picture Box Properties:

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

Picture Box Events:

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

Picture Box Methods:

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

** loadpicture – it is a function to load the picture by code
`picExample.Picture = LoadPicture("c:\pix\sample.bmp")`

Image Box

- An **image box** is very similar to a picture box in that it allows you to place graphics information on a form. Image boxes are more suited for static situations that is, cases where no modifications will be done to the displayed graphics.
- Image box graphics can be resized by using the **Stretch** property.

Image Box Properties:


Picture	Establishes the graphics file to display in the image box.
Stretch	If False, the image box resizes itself to fit the graphic. If True, the graphic resizes to fit the control area.

Image Box Events:

Click	Triggered when a image box is clicked.
DbClick	Triggered when a image box is double-clicked.

Form1

IMAGE GALLERY



FIRST PREVIOUS NEXT LAST

This is a screenshot of a Windows-style application window titled "Form1". The window has a blue title bar with standard minimize, maximize, and close buttons. The main content area has a light beige background with a dotted pattern. At the top, the text "IMAGE GALLERY" is centered. Below this is a large, empty rectangular area with a light beige background and a green cross-hatched border. At the bottom, there are four buttons labeled "FIRST", "PREVIOUS", "NEXT", and "LAST" arranged horizontally.

Drive List Box

- The **drive list box** control allows a user to select a valid disk drive at run time. It displays the available drives in a drop-down combo box. No code is needed to load a drive list box; Visual Basic does this for us. We use the box to get the current drive identification.

Drive List Box Properties:

Drive Contains the name of the currently selected drive.

Drive List Box Events:

Change Triggered whenever the user or program changes the drive selection.

** To change the drive physically use the function – chdrive, which change the drive

ChDrive Drive1.Drive

Directory List Box

- The **directory list box** displays an ordered, hierarchical list of the user's disk directories and subdirectories. The directory structure is displayed in a list box. Like, the drive list box, little coding is needed to use the directory list box – Visual Basic does most of the work for us.

Directory List Box Properties:

Path	Contains the current directory path.
List	Array of items in list box.
ListCount	Number of items in list.
ListIndex	The number of the most recently selected item in list. If no item is selected, ListIndex = -1.

Directory List Box Events:

Change	Triggered when the directory selection is changed.
---------------	--

File List Box

- The **file list box** locates and lists files in the directory specified by its Path property at run-time. You may select the types of files you want to display in the file list box.

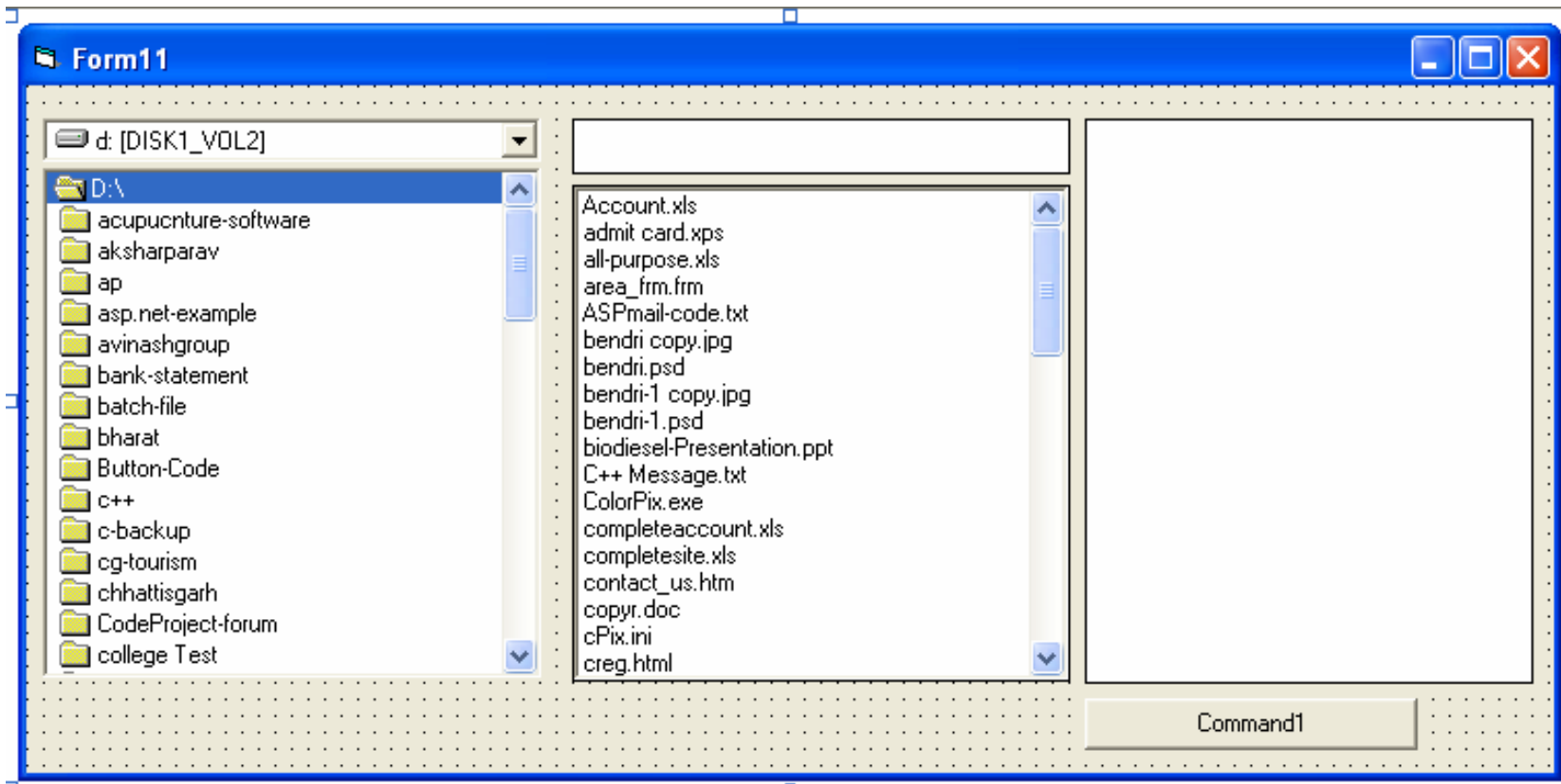
File List Box Properties:

FileName	Contains the currently selected file name.
Path	Contains the current path directory.
Pattern	Contains a string that determines which files will be displayed. It supports the use of * and ? Wildcard characters. For example, using *.dat only displays files with the .dat extension.
List	Array of items in list box.
ListCount	Number of items in list.
ListIndex	The number of the most recently selected item in list. If no item is selected, ListIndex = -1.

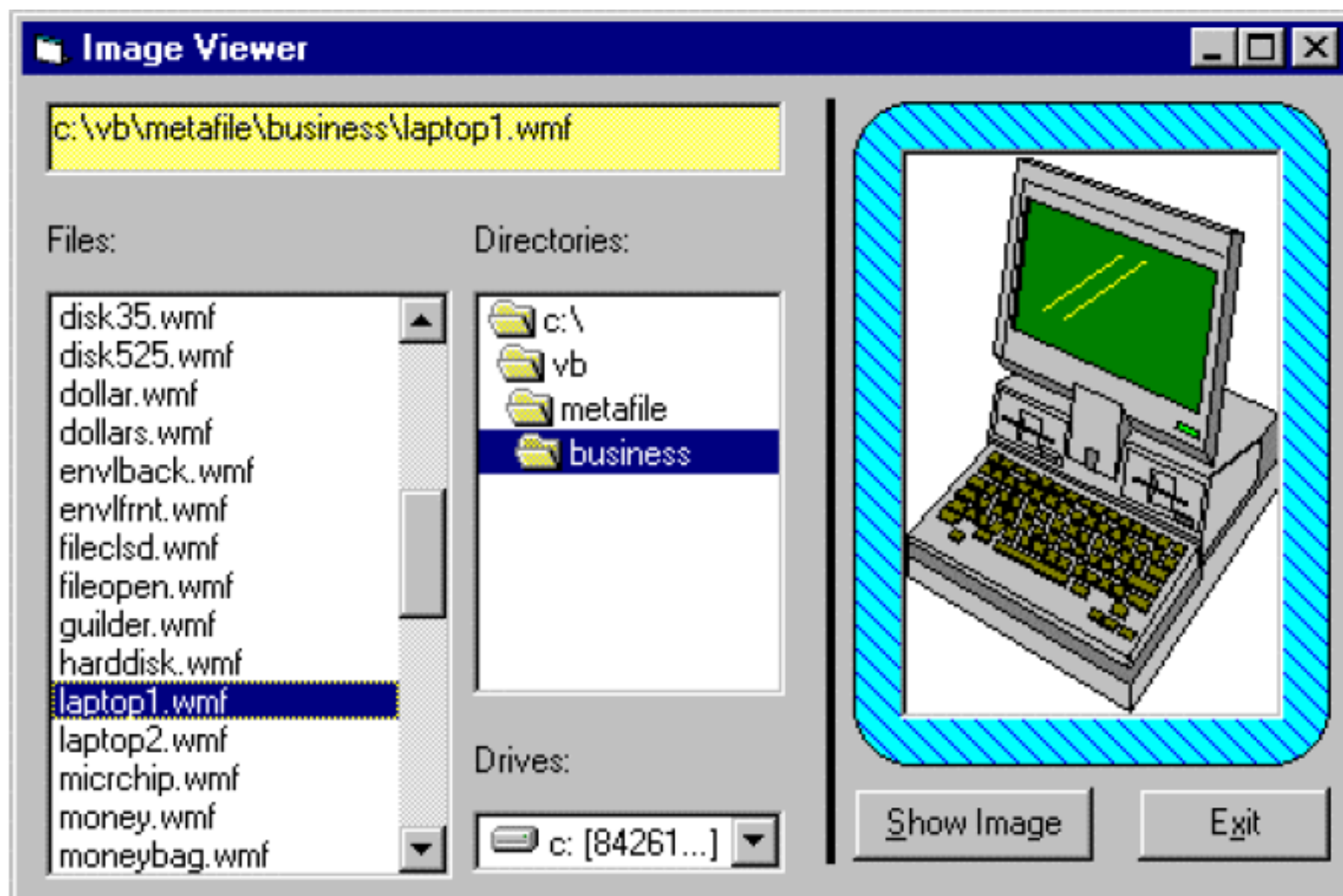
File List Box Events:

DbClick	Triggered whenever a file name is double-clicked.
PathChange	Triggered whenever the path changes in a file list box.

- Image Show the Simply Synchronization of Drive / Directory / List Box



- Image Viewer Application -



Creating Menus

- Menus are used to provide a user with choices that control the application. Menus are easily incorporated into Visual Basic programs using the **Menu Editor**.
- A good way to think about elements of a menu structure is to consider them as a hierarchical list of command buttons that only appear when pulled down from the menu bar.
- To add menu to a VB application, we need to invoke **Menu Editor (Tool-> Menu Editor)**. The menu editor of VB is an interactive way to create and modify menu and that too with minimal coding

The image shows the Visual Basic Menu Editor dialog box. The title bar reads "Menu Editor". The dialog is divided into two main sections. The top section contains property boxes for the selected menu item, including fields for Caption, Name, Index, Shortcut, HelpContextID, and NegotiatePosition, along with checkboxes for Checked, Enabled, Visible, and WindowList. The bottom section is a list box containing the menu structure, with "&File" selected. Callouts with lines pointing to specific parts of the dialog provide the following descriptions:

- Menu Controls Property Boxes**: Points to the top section of the dialog containing the Caption, Name, Index, Shortcut, HelpContextID, NegotiatePosition, and checkboxes.
- Buttons for moving to next item or inserting or deleting items**: Points to the "Next", "Insert", and "Delete" buttons located below the list box.
- Buttons for changing levels or position of menu items**: Points to the four small arrow buttons (left, right, up, down) located to the left of the list box.
- Menu Control List Box**: Points to the list box containing the menu items: "&File", "&New", "&Open", "&Save", "E&xit", "&Edit", "Cu&t Ctrl+X", "&Copy Ctrl+C", "&Paste Ctrl+V", and "F&ormat".

- The **Caption** box is where you type the text that appears in the menu bar. **Access keys (its allow the user to open a menu by pressing the ALT key and typing a designated letter)** are defined in the standard way using the ampersand (&). Separator bars (a horizontal line used to separate menu items) are defined by using a Caption of a single hyphen (-).
- The **Name** box is where you enter a control name for each menu item. This is analogous to the Name property of command buttons and is the name used to set properties and establish the **Click** event procedure for each menu item. Each menu item must have a name, even separator bars!
- The **Index** box is used for indexing any menu items defined as control arrays.
- The **Shortcut** dropdown box is used to assign shortcut keystrokes to any item in a menu structure. The shortcut keystroke will appear to the right of the caption for the menu item. An example of such a keystroke is using **Ctrl+X** to cut text.
- At the bottom of the Menu Editor form is a list box displaying the hierarchical list of menu items. Sub-menu items are indented to their level in the hierarchy. The right and left arrows adjust the levels of menu items, while the up and down arrows move items within the same level. The **Next**, **Insert**, and **Delete** buttons are used to move the selection down one line, insert a line above the current selection, or delete the current selection, respectively.

- Each menu item has four properties associated with it. These properties can be set at design time using the Menu Editor or at run-time using the standard dot notation. These properties are:

Checked	Used to indicate whether a toggle option is turned on or off. If True, a check mark appears next to the menu item.
Enabled	If True, menu item can be selected. If False, menu item is grayed and cannot be selected.
Visible	Controls whether the menu item appears in the structure.
WindowList	Used with Multiple Document Interface (MDI) – not discussed here.

Using Pop-Up Menus

- **Pop-up menus** can show up anywhere on a form, usually being activated by a single or double-click of one of the two mouse buttons. Most Windows applications, and Windows itself, use pop-up menus.
- Adding pop-up menus to your Visual Basic application is a two step process. First, you need to create the menu using the **Menu Editor** (or, you can use any existing menu structure with at least one sub-menu). If creating a unique pop-up menu (one that normally does not appear on the menu bar), it's **Visible** property is set to be **False** at design time. Once created, the menu is displayed on a form using the **PopupMenu** method.

The **PopupMenu** method syntax is:

ObjectName.**PopupMenu** MenuName, Flags, X, Y

The ObjectName can be omitted if working with the current form. The Arguments are:

MenuName	Full-name of the pop-up menu to display.
Flags	Specifies location and behavior of menu (optional).
X, Y (X, Y)	coordinate of menu in twips (optional; if either value is omitted, the current mouse coordinate is used).

- The **Flags** setting is the sum of two constants. The first constant specifies location:

Value	Meaning	Symbolic Constant
0	Left side of menu is at X coordinate	vbPopupMenuLeftAlign
4	Menu is centered at X coordinate	vbPopupMenuCenterAlign
8	Right side of menu is at X coordinate	vbPopupMenuRightAlign

The second specifies behavior:

Value	Meaning	Symbolic Constant
0	Menu reacts only to left mouse button	vbPopupMenuLeftButton
2	Menu reacts to either mouse button	vbPopupMenuRightButton

Common Dialog Box

- The common dialog control provide a standard set of dialog boxes for operations such as opening and saving files, setting print option, selecting colors and font etc.
- The common dialog tool, although it appears on your form, is invisible at run-time. You cannot control where the common dialog box appears on your screen. The tool is invoked at run-time using one of five '**Show**' methods. These methods are:

Method	Common Dialog Box
ShowOpen	Open dialog box
ShowSave	Save As dialog box
ShowColor	Color dialog box
ShowFont	Font dialog box
ShowPrinter	Printer dialog box

[Commdialogcontrol.method](#)

- The **Open** common dialog box provides the user a mechanism for specifying the name of a file to open / to save. The box is displayed by using the **ShowOpen / ShowSave** method.

Open / Save Dialog Box Properties:

CancelError	If True, generates an error if the Cancel button is clicked. Allows you to use error-handling procedures to recognize that Cancel was clicked.
DialogTitle	The string appearing in the title bar of the dialog box. Default is Open. In the example, the DialogTitle is Open Example.
FileName	Sets the initial file name that appears in the File name box. After the dialog box is closed, this property can be read to determine the name of the selected file.
Filter	Used to restrict the filenames that appear in the file list box. In the example, the Filter was set to allow Bitmap (*.bmp), Icon (*.ico), Metafile (*.wmf), GIF (*.gif), and JPEG (*.jpg) types.
FilterIndex	Indicates which filter component is default.
Flags	Values that control special features of the Open dialog box.
Initdir	This property set the initial directory whose files are displayed the first time the Open and Save dialog boxes are opened.

Initdir This property set the initial directory whose files are displayed the first time the Open and Save dialog boxes are opened.

Color Dialog Box

- Its is one of the simple dialog box. It has single property, **Color**, which returns the color selected by the user or sets the initially selected color when the user open the color dialog box.

Flag	Description
Cdlccfullopen(&H2)	Display the full dialog box
Cdlcchelpbutton(&H8)	Show the help button
cdlCCPreventFullOpen(&h4)	Prevent the Custom Color Section

Font Dialog Box

The user select the font, size and style.

Flag	Description
Cdlcfboth(&H3)	Display both printer and screen font
Cdlcfeffects	Show the effects option

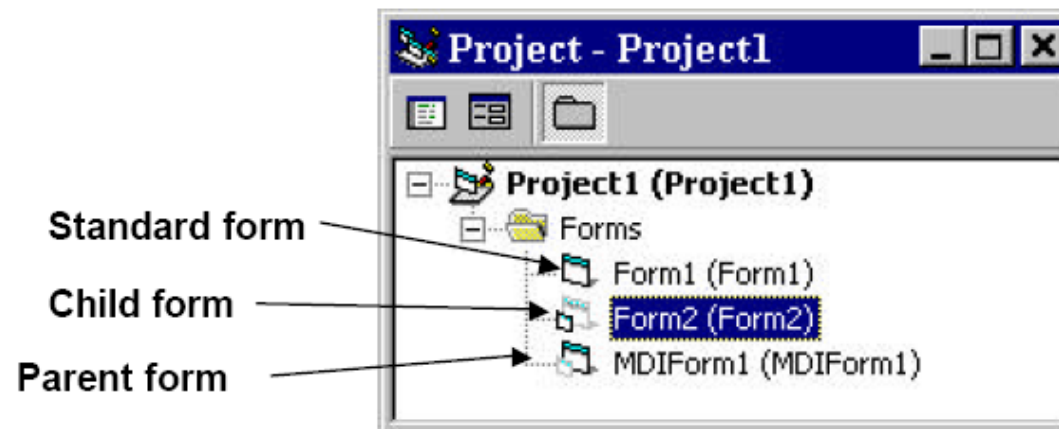
Print Dialog Box

The user select a printer nad set the certain property.

Copies	Specifies the number of copies
From Page	Specifies the page to start the printing
To Page	Specifies the page to stop printing

MDI Application

- Visual Basic actually provides a system for maintaining multiple-form applications, known as the **Multiple Document Interface (MDI)**. MDI allows you to maintain multiple forms within a single container form. Examples of MDI applications are Word, Excel, and the Windows Explorer program.
- An MDI application allows the user to display many forms at the same time. The container window is called the **parent** form, while the individual forms within the parent are the **child** forms. Both parent and child forms are modeless, meaning you can leave one window to move to another. An application can have only one parent form.
- You can determine whether a form is a child by examining its **MDIChild** property, or by examining the project window. The project window uses special icons to distinguish **standard** forms, MDI **child** forms, and MDI **parent** forms:



- At run-time all child forms are displayed within the parent form's internal area. The user can move and size child forms like any other form, but they must stay in this internal area.\
- When a child is minimized, its icon appears on the MDI parent form instead of the user's desktop.
- When a child form is maximized, its caption is combined with the parent form's caption and displayed in the parent title bar.
- The active child form's menus (if any) are displayed on the parent form's menu bar, not the child form.
- The **Arrange** command can be used to determine how the child forms and their icons (if closed) are displayed. The syntax is:

Arrange *style*

where *style* can take on these values:

Style	Symbolic Constant	Effect
0	vbCascade	Cascade all nonminimized MDI child .
1	vbTileHorizontal	Horizontally tile all nonminimized MDI child.
2	vbTileVertical	Vertically tile all nonminimized MDI child.
3	vbArrangeIcons	Arrange icons for minimized MDI child.

File Input / Output

- In many applications, it is helpful to have the capability to read and write information to a disk file.
- 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)

- The first step is to **Open** a file to write information to . The syntax for opening a sequential file for output is:

Open FileName For Output As #N

where **FileName** 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:

Close N

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.

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.

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.

Sequential File Input (Variables)

To **read variables** from a sequential file, we essentially reverse the write procedure. First, open the file using:

```
Open FileName For Input As #N
```

where **N** is an integer file number and **FileName** 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).

- The **EOF(Filenumber)** function can be used to detect an end-of-file condition, its return **True** means reach the end of file otherwise **False**.

- The another important function to read the content from file is Input function. The Syntax is :

`input (number, file number)`

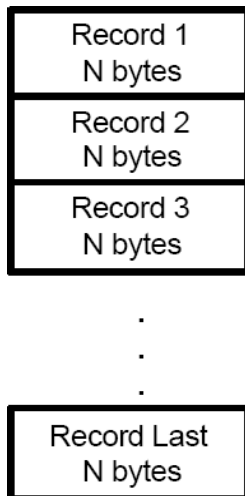
`input (LOF (file number), File Number)`

number – how many number of character to read from which file (file number)

or may be use first argument as LOF(N), the length of the file opened as N and N, the file number.

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 non sequential ways. Files which allow non sequential access are **random access files**.
- To allow non sequential 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:



- 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 lengths to obtain the total record length. To ease this task, we introduce the idea of 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*15  
City As String*20  
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 S1 As Person  
Dim S2 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  
S1.Name
```

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

- 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 **Get** 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 specify the record number. If you do not specify a record number, the next sequential position is used.

Error Handling

- How to handle the errors in programs, using both run-time error trapping and debugging techniques.

- **Error Types**

No matter how hard we try, **errors** do creep into our programs. These errors can be grouped into three categories:

1. **Syntax** errors
2. **Run-time** errors
3. **Logic** errors

- **Syntax errors** occur when you mistype a command or leave out an expected phrase or argument.
- **Run-time errors** are usually beyond your program's control.
- **Logic errors** are the most difficult to find. With logic errors, the program will usually run, but will produce incorrect or unexpected results.
- **Some ways to minimize errors:**
 - ❖ Design your application carefully. More design time means less debugging time.
 - ❖ Use comments where applicable to help you remember what you were trying to do.
 - ❖ Use consistent and meaningful naming conventions for your variables, objects, and procedures.

Run Time Error Trapping & Handling

- How to handle the errors in programs, using both run-time error trapping and debugging techniques.

- **Error Types**

No matter how hard we try, **errors** do creep into our programs. These errors can be grouped into three categories:

1. **Syntax** errors
2. **Run-time** errors
3. **Logic** errors

- **Syntax errors** occur when you mistype a command or leave out an expected phrase or argument.
- **Run-time errors** are usually beyond your program's control.
- **Logic errors** are the most difficult to find. With logic errors, the program will usually run, but will produce incorrect or unexpected results.
- **Some ways to minimize errors:**
 - ❖ Design your application carefully. More design time means less debugging time.
 - ❖ Use comments where applicable to help you remember what you were trying to do.
 - ❖ Use consistent and meaningful naming conventions for your variables, objects, and procedures.

- Run-Time Error Trapping and Handling

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

- 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 Error() function takes this error number as its argument and returns a string description of the error.
Error(Err.Number)
- 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*.
- 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.

- Here we develop 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.

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

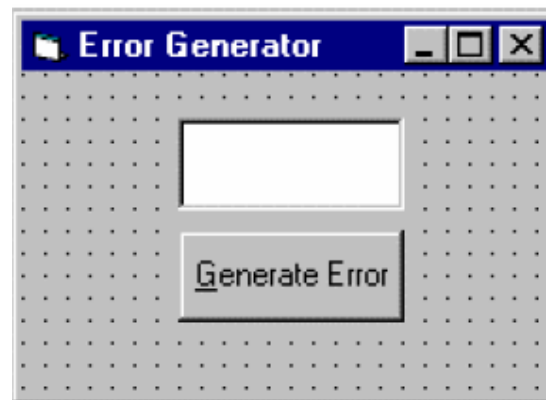
End Sub

- 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

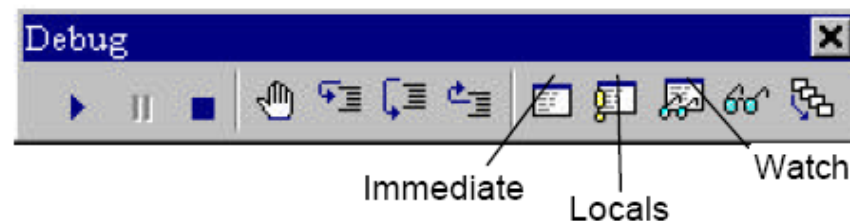


- Some common error

Error Number	Error Description
6	Overflow
9	Subscript out of range
11	Division by zero
13	Type mismatch
16	Expression too complex
20	Resume without 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

Debugging

- We now consider the search for, and elimination of, logic errors. These are errors that don't prevent an application from running, but cause incorrect or unexpected results. Visual Basic provides an excellent set of debugging tools to aid in this search.
- Debugging a code is an art, not a science. There are no prescribed processes that you can follow to eliminate all logic errors in your program.
- What we'll do here is present the debugging tools available in the Visual Basic environment (several of which appear as buttons on the toolbar) and describe their use with an example. You, as the program designer, should select the debugging approach and tools you feel most comfortable with.
- The interface between your application and the debugging tools is via three different debug windows: **the Immediate Window**, **the Locals Window**, and the **Watch Window**. These windows can be accessed from the View menu (the Immediate Window can be accessed by pressing Ctrl+G). Or, they can be selected from the Debug Toolbar (accessed using the Toolbars option under the View menu):



- All debugging using the debug windows is done when your application is in break mode. You can enter break mode by setting breakpoints, pressing **Ctrl+Break**, or the program will go into break mode if it encounters an untrapped error or a Stop statement.
- Once in break mode, the debug windows and other tools can be used to:
 1. Determine values of variables
 2. Set breakpoints
 3. Set watch
 4. variables and expressions
 5. Manually control the application
 6. Determine which procedures have been called
 7. Change the values of variables and properties

Method –

1. By Immediate Window – use the following command
Debug.Print X; Y
2. By Break Point – Set a Break Point on statement and use the print command to print the value of variable on immediate window or use question mark (?variable name)
3. The locals window shows the value of any variables within the scope of the current procedure. As execution switches from procedure to procedure, the contents of this window changes to reflect only the variables applicable to the current procedure.
4. By Watch Window - Values of watch expressions are displayed in the watch window. Add all the variable or expression on watch window.
5. By CALL Stack – That will display all active procedures, that is those that have not been exited.