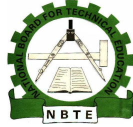
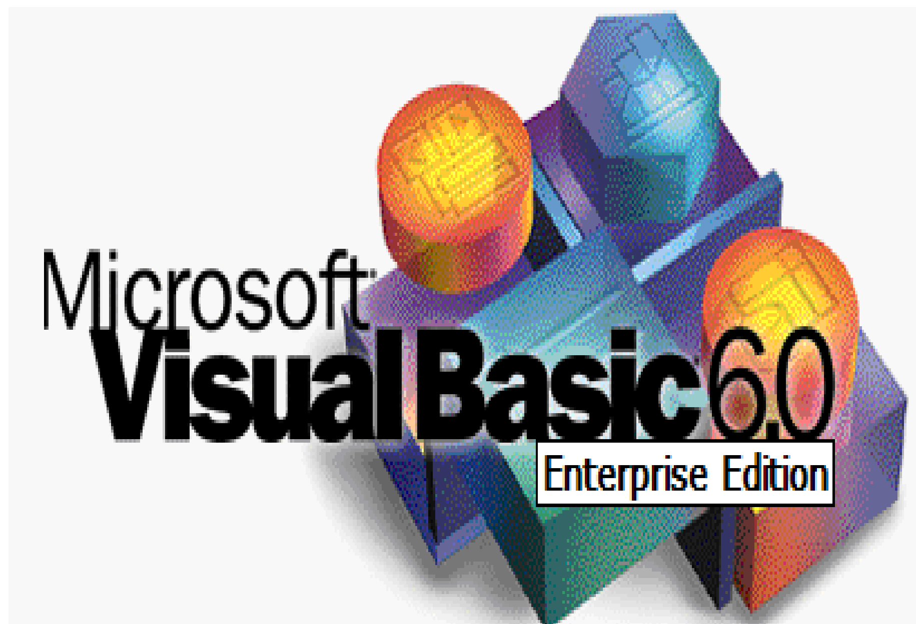


UNESCO-NIGERIA TECHNICAL & VOCATIONAL EDUCATION REVITALISATION PROJECT-PHASE II

NATIONAL DIPLOMA IN COMPUTER TECHNOLOGY



OOBASIC/VISUAL BASIC PROGRAMMING

COURSE CODE: COM 211

YEAR I SEMESTER II
THEORY
Version 1: December,2008

Table Of Contents

WEEK 1 Concept of programming 4
Systems Development Cycle 4
How windows work 5
WEEK 2 The Visual Basic environment 5
Starting Visual Basic 7
Stopping Visual Basic 8
Getting online help 8
Opening Application 12
Creating Simple application (Wizard) 12
Running your application 18
Creating Executable File 19
Saving your application 20
VB Character set 22
Relational Operators 22
Arithmetic Operators 22
Blank character 22
Data Types 22
WEEK 3 Declaring Variables 26
Assigning Values to Variables 28
Mathematical Expressions 30
Conditional Operators 33
WEEK 4 Properties of the Form 34
Intrinsic Controls 37
Label control 38
Text Box 40
Focus In on Controls 42
The Command Button 42
The PictureBox Control 44
The Image Control 46
WEEK 5 Creating non-wizard applications 47
Adding Controls to applications 47
Managing Controls 48
Change the control properties 48
Handling Control Event 57
Writing Code 63
WEEK 6 If...Then...Else Statement 65
Select Case Statement 67
OptionButton Control 69
CheckBox Control 72
Frame Control 78
WEEK 7 Do...Loop Statement 80
For...Next Statement 81
Sub Statement 84
WEEK 8 Supplied Numeric Functions (Math Functions) 88
Abs Function 88
Int Functions 88
Rnd Function 89
Sqr Function 90
Supplied String Function 90
Len Function 90

LCase and Functions 91
UCase Function 91
Asc Function 92
Chr Function 92
Supplied Time And Date Functions 93
Now Function 93
Date Function 93
Time Function 93
WEEK 9 Non-Arrays data values 94
Arrays data values 94
Declaring Fixed-Size Arrays 95
Multidimensional Arrays 97
WEEK 10 List Boxes Controls That Work Like Arrays 99
List Boxes: Controls That Work Like Arrays 109
WEEK 11 Combo Boxes 117
The Timer Control 122
Scrolling the Scroll Bars 124
WEEK 12 MsgBox Function 128
InputBox Function 131
WEEK 13 The Basic Elements of a Database 133
WEEK 14 Using Data Control 134
Data Access Objects (DAO) 136
Visual Basic Wizard 137
Data Form Wizard 140
WEEK 15 Menu Basics 147
Menu Control 148
Menu Editor Command (Tools Menu) 148
Menu Editor Dialog Box 148
Creating Menus with the Menu Editor 151
Writing Code for Menu Controls 155
MsgBox Function 156
InputBox Function 158

WEEK 1:

INTRODUCTION

During this week you will learn:

- Concept of programming
- Systems Development Cycle
- How windows work
- The Visual Basic environment
- Starting Visual Basic
- Stopping Visual Basic
- Getting online help
- Opening Application
- Creating Simple application (Wizard)
- Running your application
- Creating Executable File
- Saving your application

Concept of programming

A program is a set of detailed instructions that tells the computer what to do.

VISUAL BASIC is a high level programming language evolved from the earlier DOS version called BASIC. BASIC means Beginners' All purpose Symbolic Instruction Code. It is a fairly easy programming language to learn. The codes look a bit like English Language. Visual Basic falls into a category of programming referred to as event-driven programming. Event-driven programs respond to events from the computer, such as the mouse button being pressed. The designer uses ready-made objects such as CommandButtons and TextBoxes, to build user interfaces that make up the application. This approach to programming drastically reduces the amount of code required to develop a Windows application.

Systems Development Cycle

Most IT projects work in cycles. First, the needs of the computer users must be analyzed. This task is often performed by a professional Systems Analysts who will ask the users exactly what they would like the system to do, and then draw up plans on how this can be implemented on a real computer based system.

The programmer will take the specifications from the Systems Analyst and then convert the broad brushstrokes into actual computer programs. Ideally at this point there should be testing and input from the users so that what is produced by the programmers is actually what they asked for.

Finally, there is the implementation process during which all users are introduced to the new systems, which often involves an element of training.

Once the users start using the new system, they will often suggest new improvements and the whole process is started all over again.

These are methodologies for defining a systems development cycle and often you will see four key stages, as listed below.

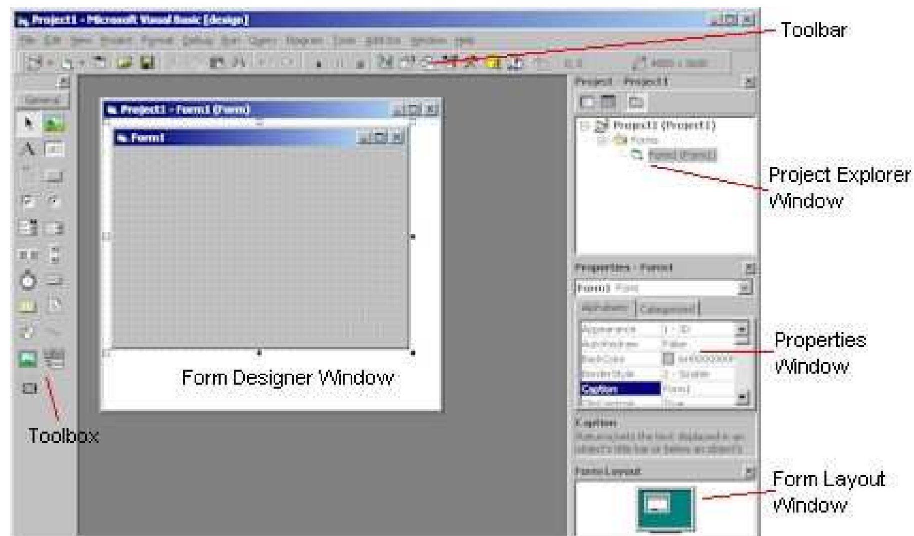
- **Feasibility Study**
- **Design**
- **Programming**
- **Implementation**

How windows work

Windows is an GUI operating system. With GUI it easily recognized graphic icons be selected using the mouse and commands chosen from menus, This is much easier for the user than typing in the specific lines of code that were required by MS-Dos in order to perform basic operations.
In GUI operating system, more than one application can be open at the same time. Processor time is shared between computing tasks and this called multitasking.

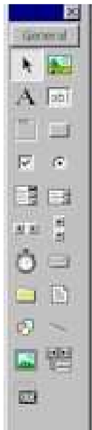
The Visual Basic environment

The Visual Basic environment is made up of several windows. The initial appearance of the windows on your screen will depend on the way your environment has been set up.



The tool bar The Visual Basic tool bar functions like the tool bar in any other Microsoft application. It provides shortcuts for many of the common operating commands. It also shows you the dimensions and location of the form currently being designed.

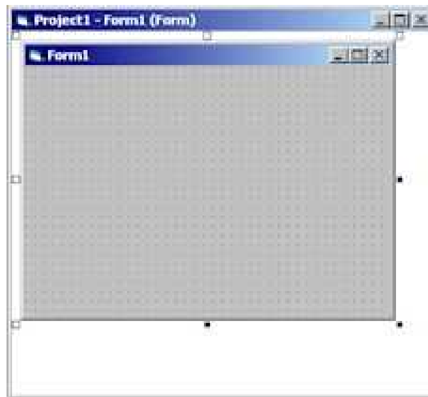
The tool box The tool box gives you access to the controls that you use on a form.



A control is an object such as a button, label or grid.
Controls are used on forms to display output or get input.
Each control appears as a button in the tool box. If the control you are looking for is not in the toolbox, select Components from the Project menu.
If the tool box is not displayed on your screen, or if at any time during the exercises you close it, choose Toolbox from the View menu.

The form designer window

This window is where you design the forms that make up your user interface.



If the form designer window is not displayed on your screen, or if at any time during the exercises you close it, choose Object from the View menu.

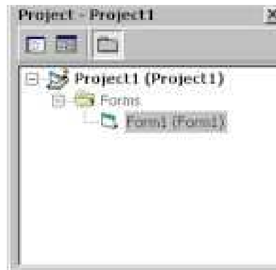
The properties window :

A form, and each control on it, has a set of properties which control its characteristics such as size, position and color.

The properties window lists all the properties a control has and their value. The default value of a property can be changed by setting the property value using the properties window when you design your application or changed by assigning a new value in code while your application is running.
If the properties window is not displayed on your screen, or if at any time during the exercises you close it, choose Properties Window from the View menu.

The project explorer window

A project is a collection of the forms and code that make up an application. Each form in your application is represented by a file in the project explorer window.



A form file contains both the description of the screen layout for the form and the program code associated with it. If the project explorer window is not displayed on your screen, or if at any time during the exercises you close it, choose Project Explorer from the View menu.

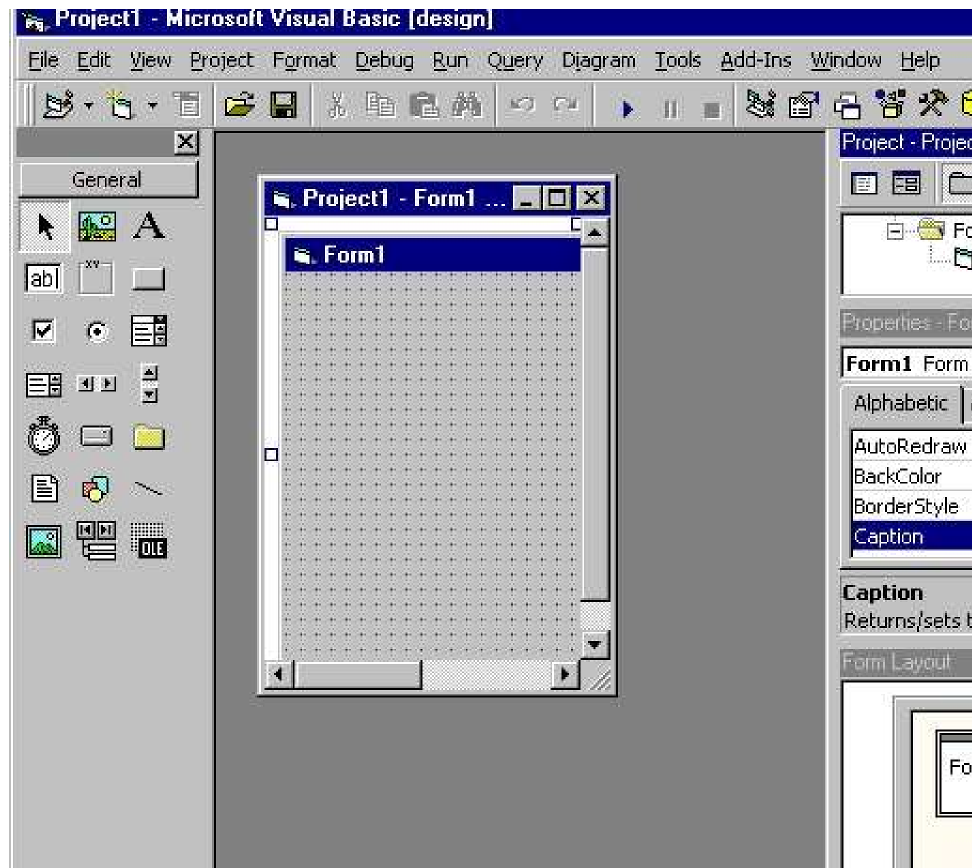
The form layout window

Move the form in the screen in this window to set the position of your form when your application is running.

You may wish to close the form layout window to allow more space for the properties window. To open the window again, select Form Layout Window from the View menu.

Starting Visual Basic

- From the **Windows Start** menu, choose **Programs**, **Microsoft Visual Studio 6.0**, and then Microsoft Visual Basic 6.0.
- Visual Basic 6.0 will display the following dialog box as shown in this figure



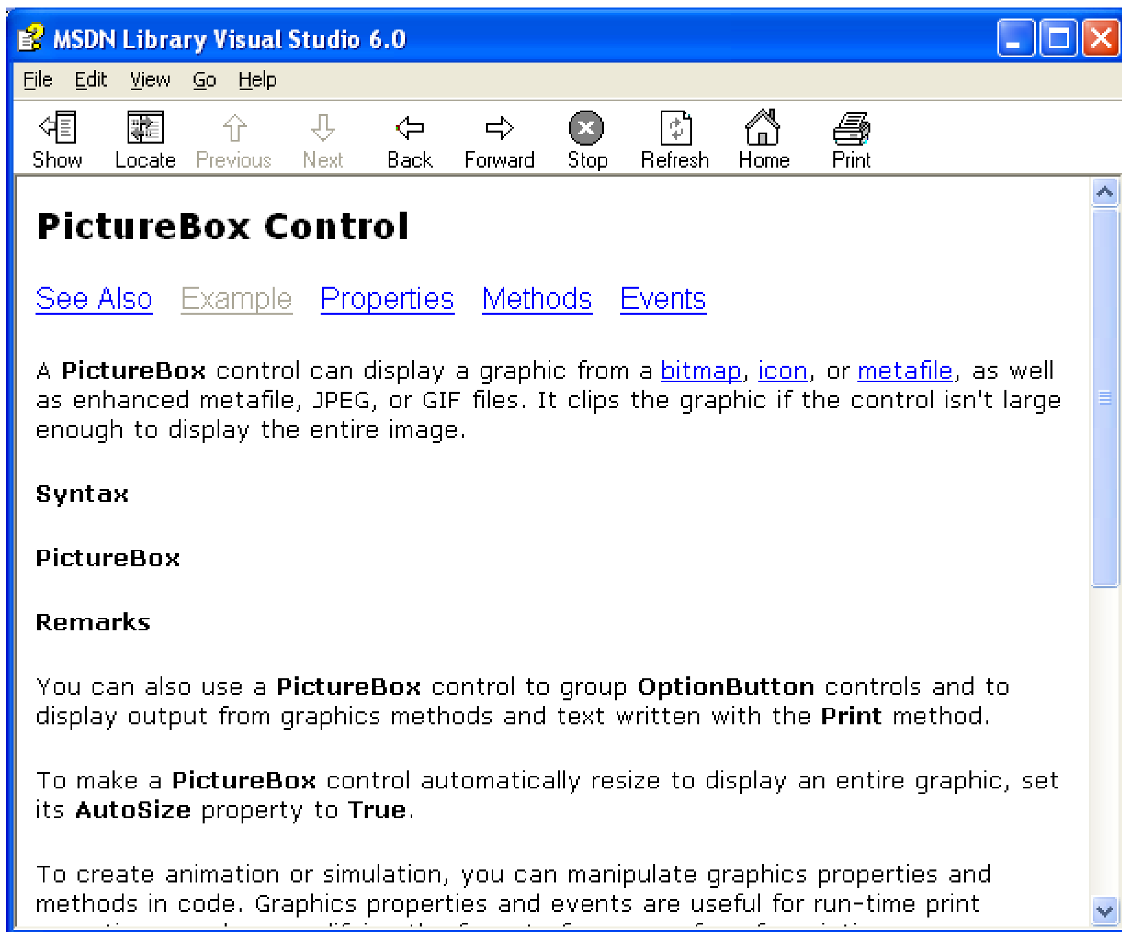
Stopping Visual Basic

- From the **File** menu, choose **Exit** and then Microsoft Visual Basic 6.0. ask you to
- save changes in your project.

Getting online help

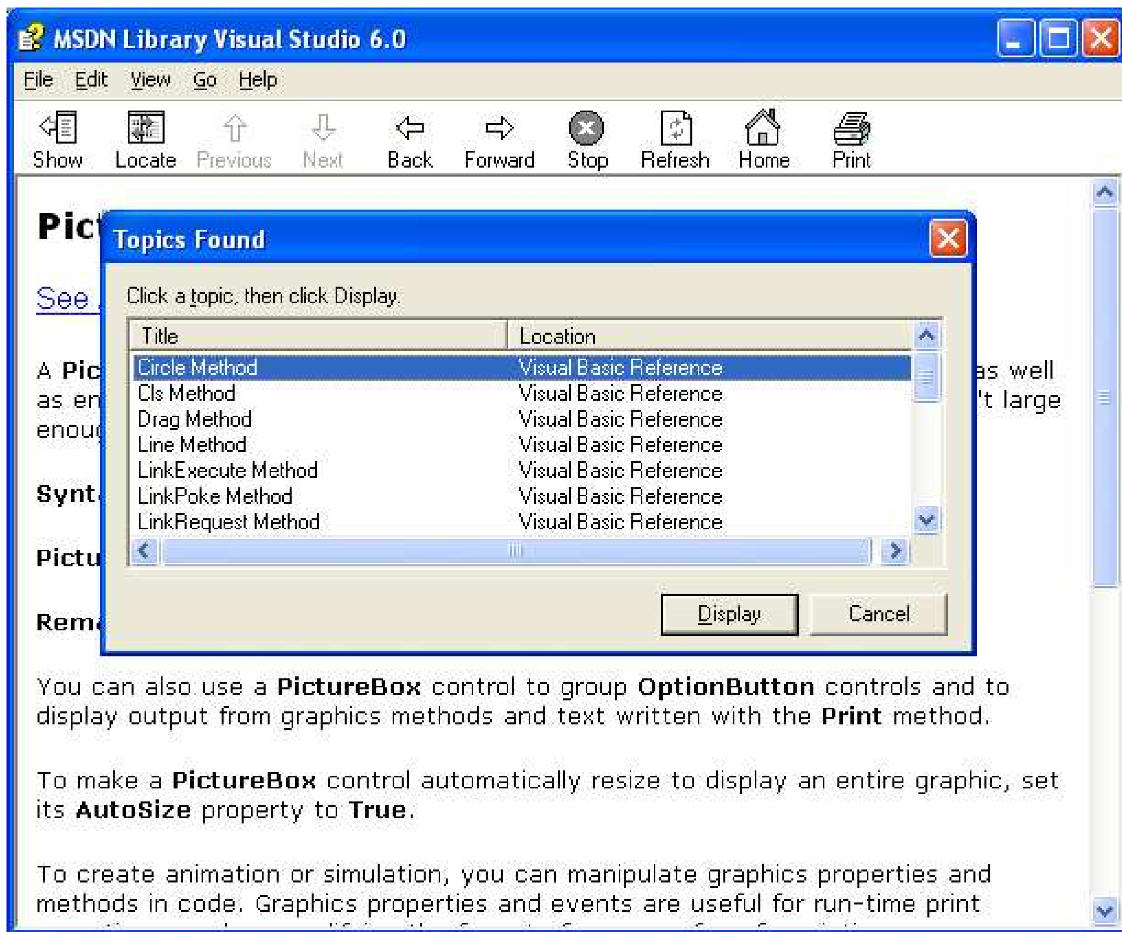
If you've used online help before, you may not think you need to read this section. Although you might be able to figure out Visual Basic's online help yourself, the help is fairly advanced and varies from most other online help you may be used to. This topic section describes some of the help tools available from within Visual Basic.

The content-sensitive nature of Visual Basic's help system extends to almost every menu option, screen element, control, window, and language command. When you want help and aren't sure exactly where to turn first, press F1 and let Visual Basic give it a try. For example, if you think you need to use the Picture Box control but want to read a description first to make sure that you have the right control, click the Toolbox's Picture Box control and then press F1. Visual Basic sees that you've clicked the Picture Box and returns with the help screen shown in this figure



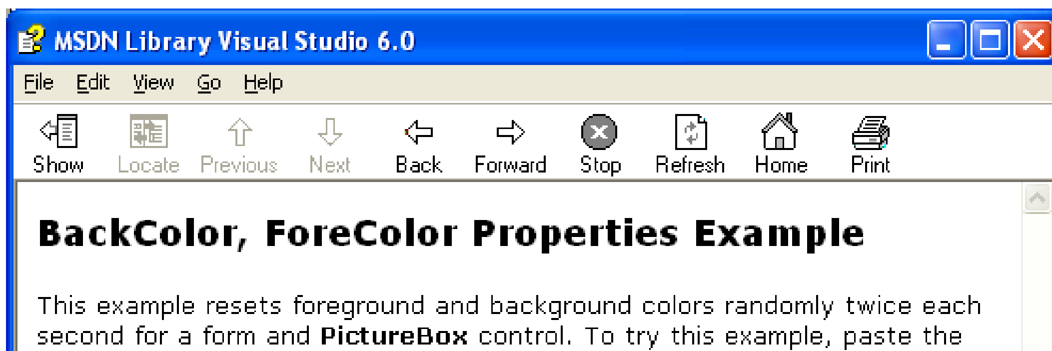
Click any screen element and press F1 for help

Throughout the help screens, Microsoft has scattered numerous links to related topics. When you click any underlined word or phrase inside a help window, Visual Basic responds with a pop-up definition or an additional help screen. Often, so may related topics appear throughout the help system that when you click a link, Visual Basic displays a scrolling Topics Found list, from which you can choose the description that most closely matches the topic you need.



Help links often provide several alternatives.

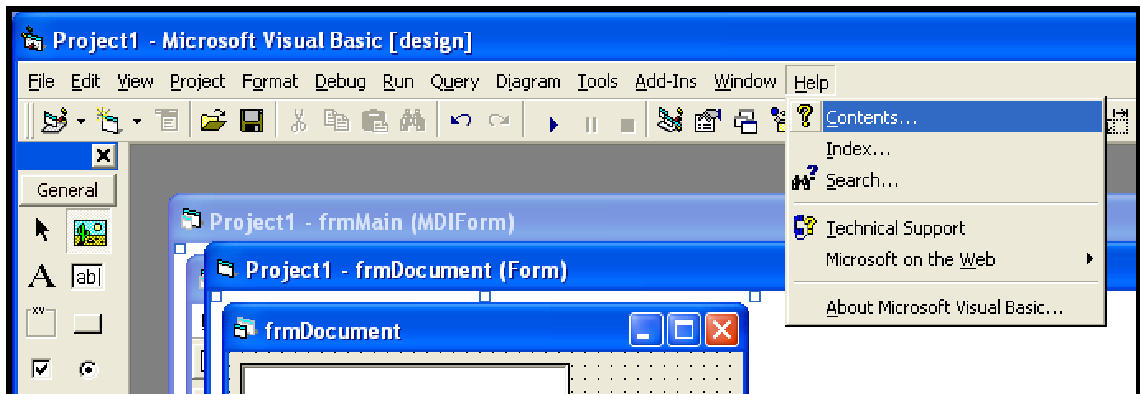
When you click an Example hypertext link, Visual Basic displays a window similar to the one shown in Figure. Although the help might look ambiguous at this point, you'll grow to appreciate the helpful suggestion when you begin learn the Visual Basic language. The Example help link shows you real Visual Basic language code that uses the item you've requested help for. As a programmer, you'll therefore see how to implement the item inside your own Visual Basic code by looking at the sample Visual Basic provides.



Visual Basic shows you sample code that uses the property or control.

The Help Menu

When you choose the first topic on the Help menu, Microsoft Visual Basic Topics, Visual Basic displays a help dialog box . This dialog box contains the usual Windows-like help tools. You can open and close the book icons on the Contents page to read about different Visual Basic topics. You can search for a particular topic in the index by clicking the Index tab. To locate every occurrence of a particular help reference word or phrase, you can click the Find tab to build a comprehensive help database that returns multiple occurrences of topics.

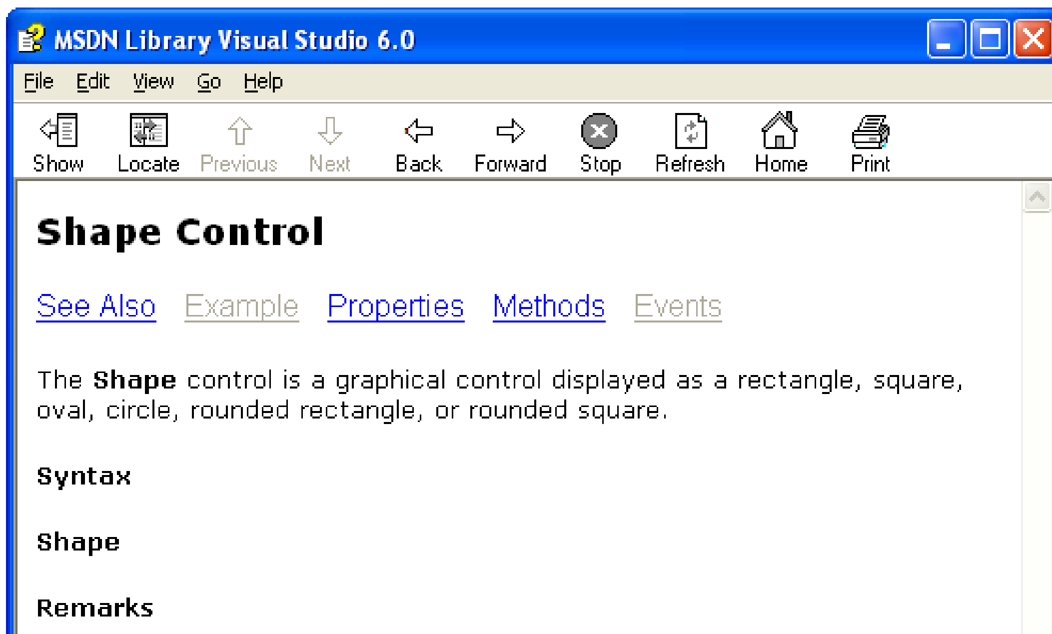


Example :

Get an instant definition for help links with a dotted underline.

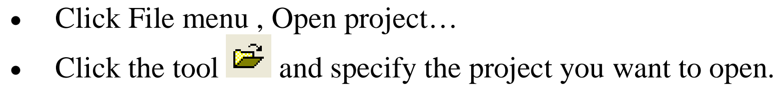
Pop-up definition, Hyperlinks

Close the help window by clicking the window's Close button.

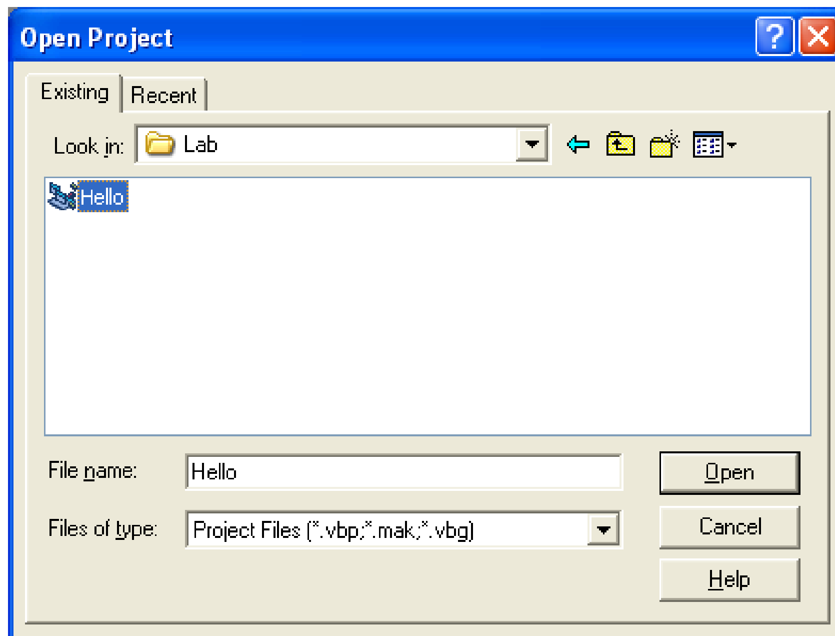


Opening Application

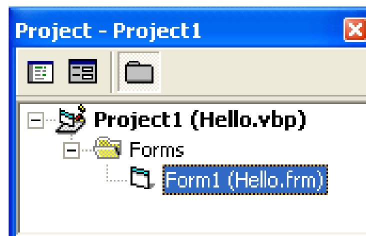
To open a project, you can do one of two things:

- Click File menu , Open project…
- Click the tool  and specify the project you want to open.

Then select Hello project and press Open.

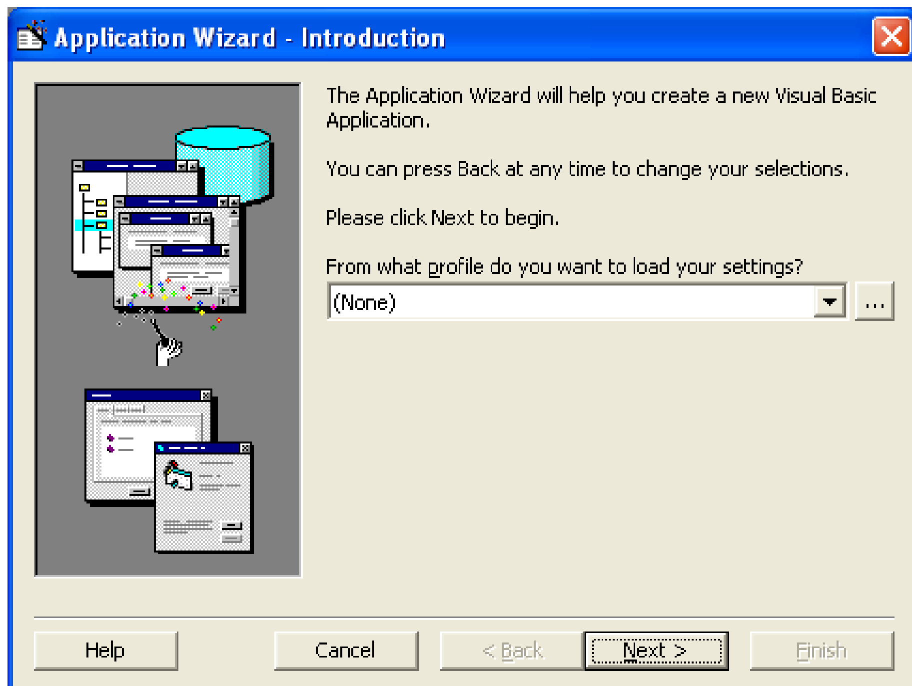


The project window will display the file "Hello.frm" from your project.



Creating Simple application (Wizard)

You start the application wizard from the New Project dialog box or by choosing New Project from the File menu. Click the VB Application Wizard icon to start the wizard. This Figure shows the application wizard's opening screen.



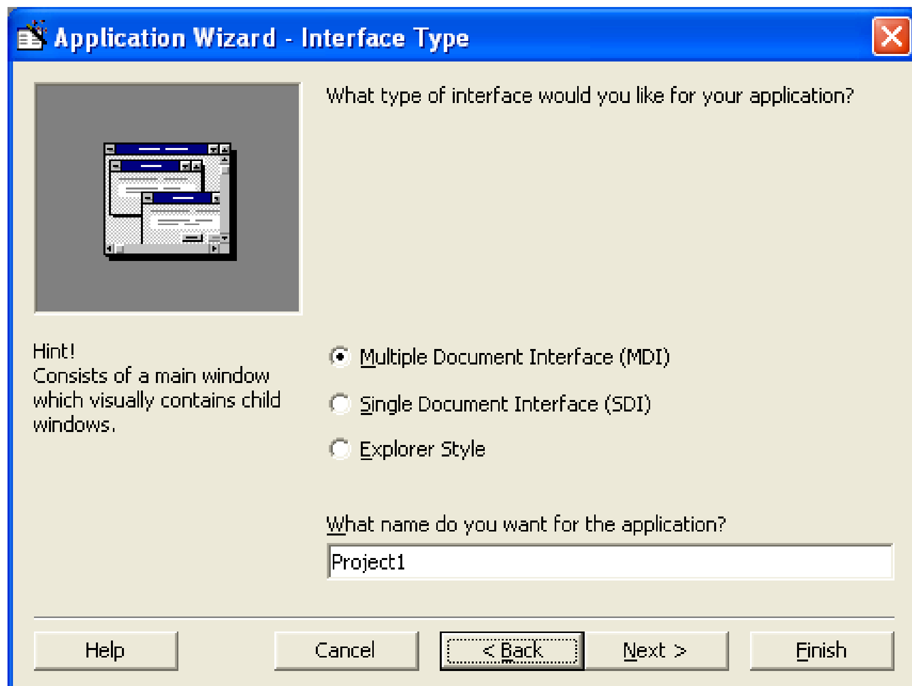
Example

Assuming that you started the application wizard in the previous section, follow these steps to build your first application:

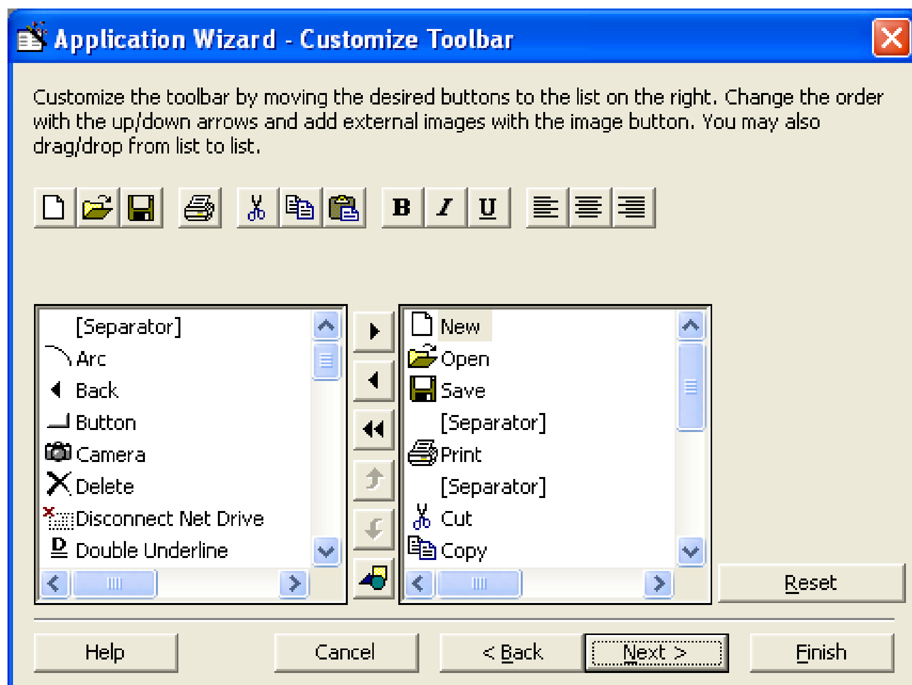
1- Click the Next button to display the Interface Type dialog box. The wizard can generate one of three types of user interfaces for the application you're generating:

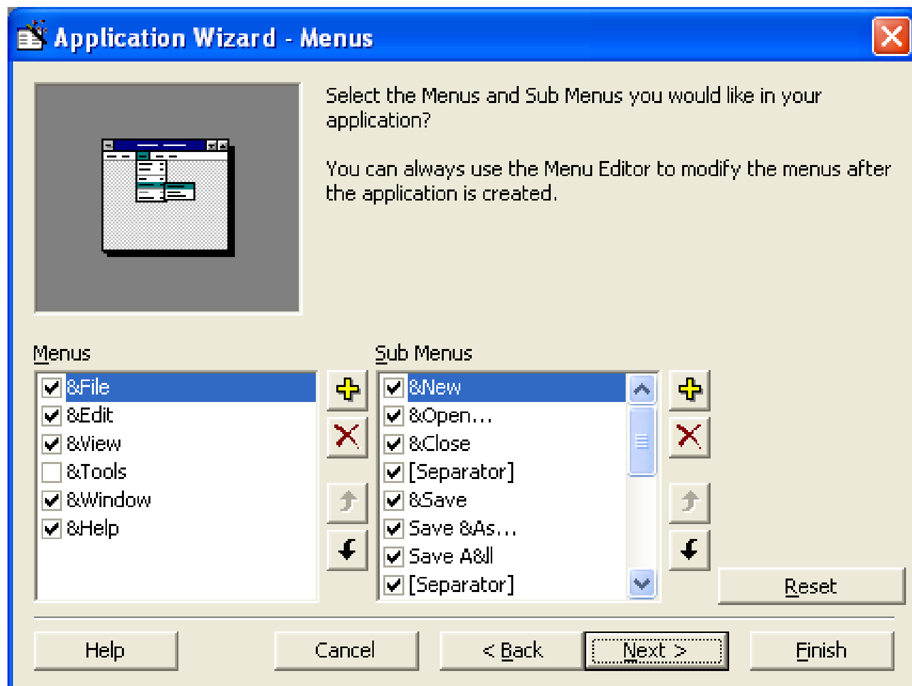
- MDI (Multiple Document Interface) lets you create a program window that contains embedded windows called child windows.
- SDI (Single Document Interface) lets you create a program with one or more windows that exist at the same level (not windows within windows).
- Explorer Style lets you create programs that somewhat take on the Books Online appearance, with a summary of topics or windows in a left pane and the matching program details in the right pane.

2- The MDI option should already be selected. If not, click the MDI option.

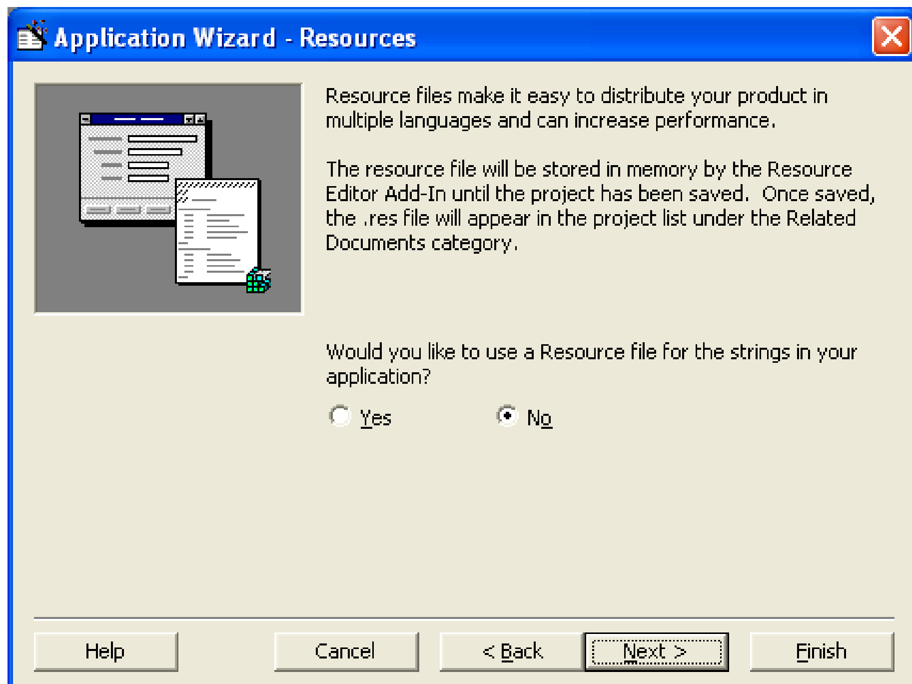


3- Click Next to display the menu selection dialog box. You can select certain menu options that will appear on your application's menu bar. By using the dialog box's options, you can help ensure that your application retains the standard Windows program look and feel. (You can add your own menu options after the wizard generates the program's initial shell.) For now, leave these options selected: File, Edit, Window, and Help.

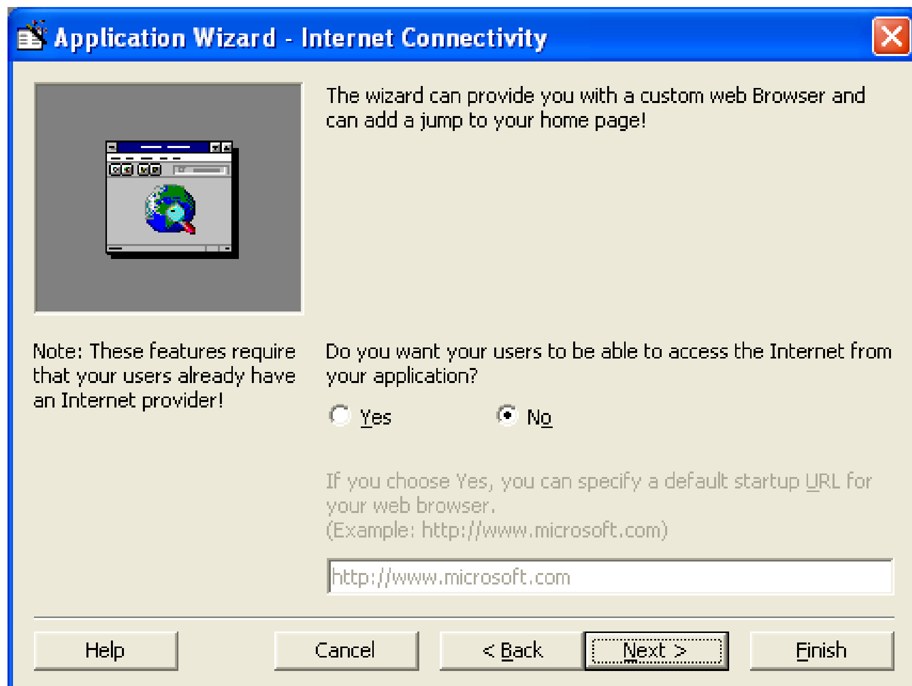




4- Click Next to display the wizard's Resources dialog box. A resource might be a menu, a text string, a control, a mouse cursor, or just about any item that appears in a program.

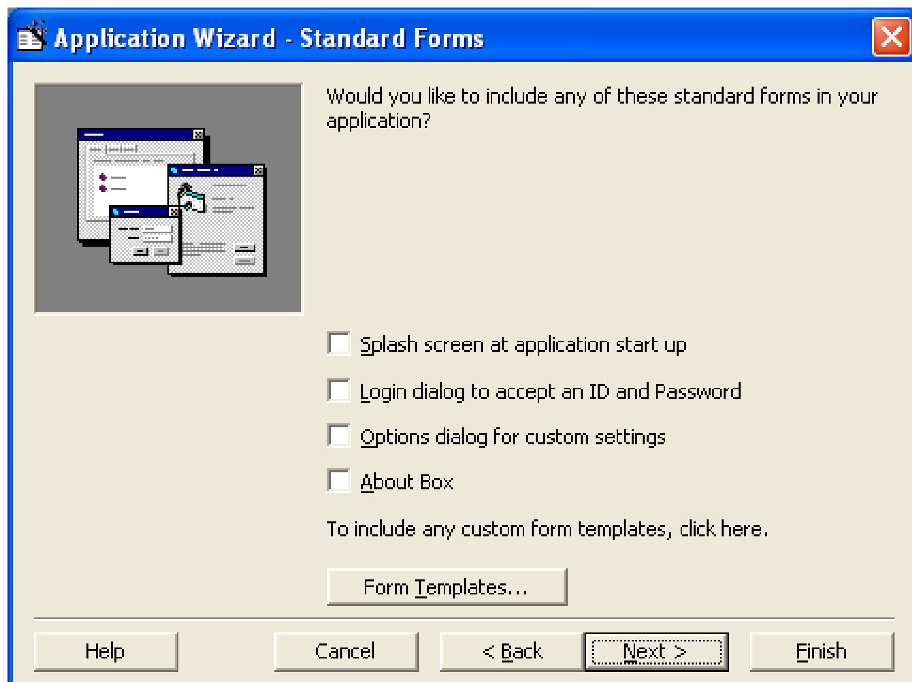


5- Click Next, you'll bypass the Internet connectivity dialog box because you don't need to add such connectivity to your first application shell.

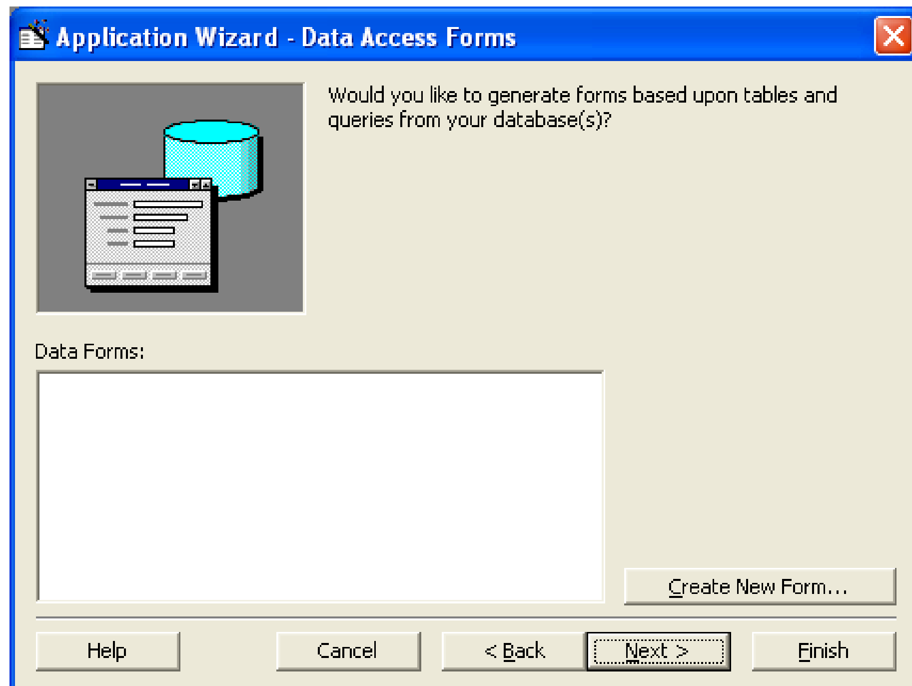


6- Determines which forms appear in your application:

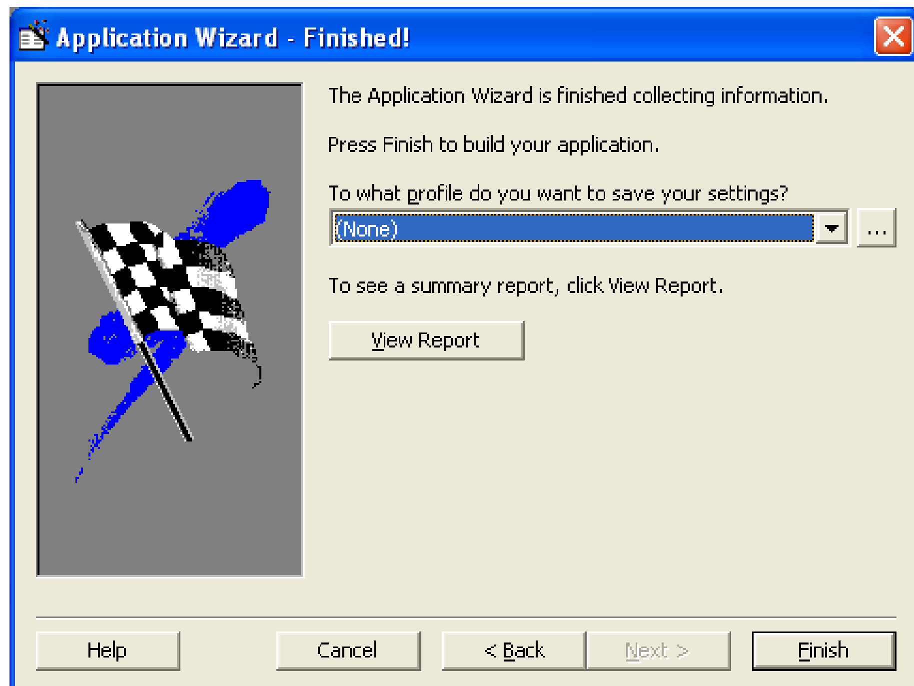
- A splash form is an opening title form that your users see when they first run your application.
- A login form requests the user's ID and password, in case you want to add security features to your application.
- The options dialog box gives users the ability to modify certain application traits.
- The About box is accessed from most Windows Help menus and provides your program description and version.



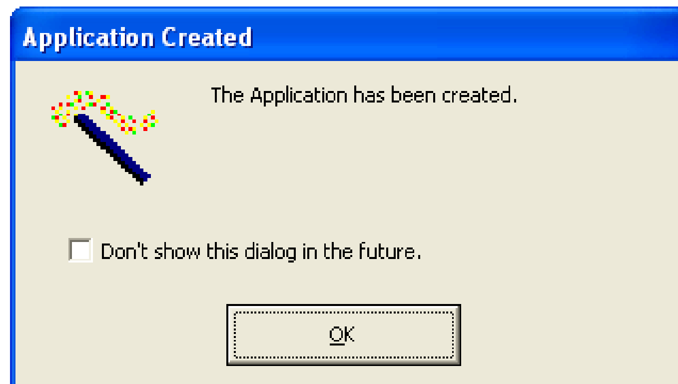
8. Check the About Box but leave the other options unchecked.
9. Click the Next button twice to display the final application wizard dialog box. (You'll bypass the database access dialog box because you won't be retrieving database data in this first application.)



10. Click the Finish button. The wizard generates the application before your eyes. You'll see the wizard generating forms and titles; without the wizard, you would have to perform these steps yourself. When finished, the application wizard displays a dialog box to tell you that the application is completed.



11. Click OK to close the final application wizard dialog box. A summary report appears, to describe the generated program.



Running your application

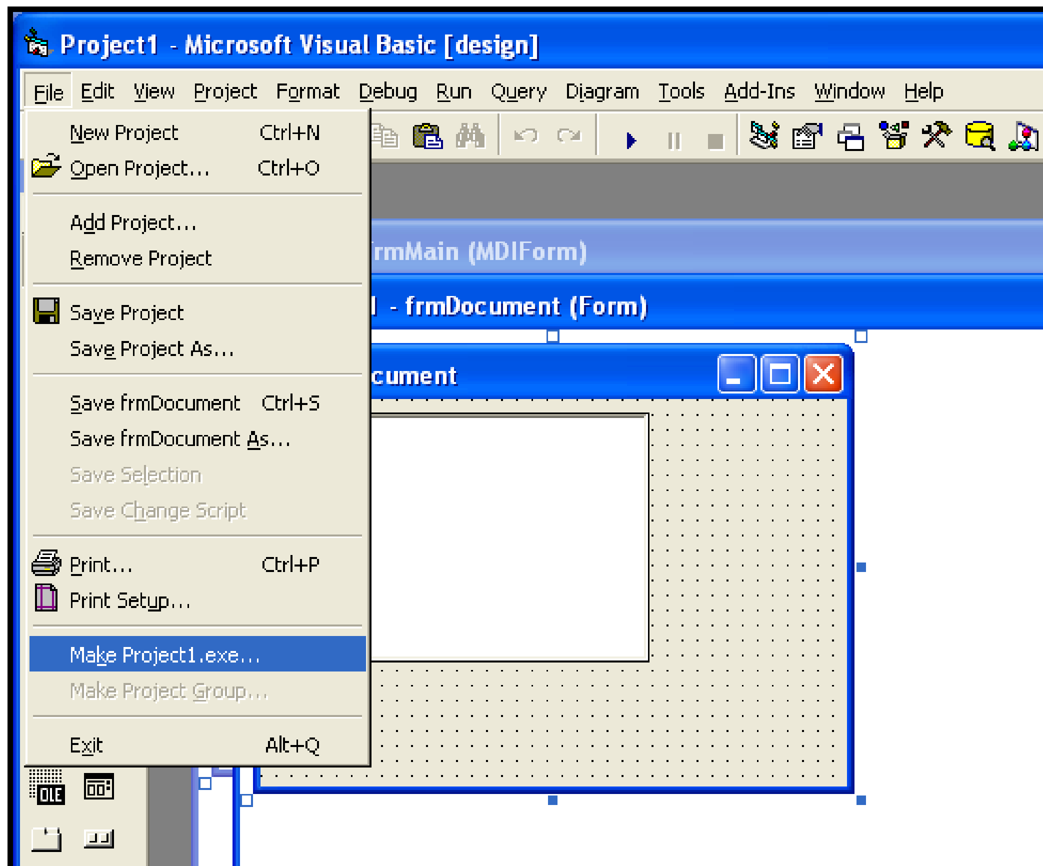
Now that the form is complete you can see it in action by running it by simply pressing F5 key. Or click on run button on the standard tool bar

When you have written code for the buttons, running the application will allow you to activate the code. For now your buttons will not do anything.

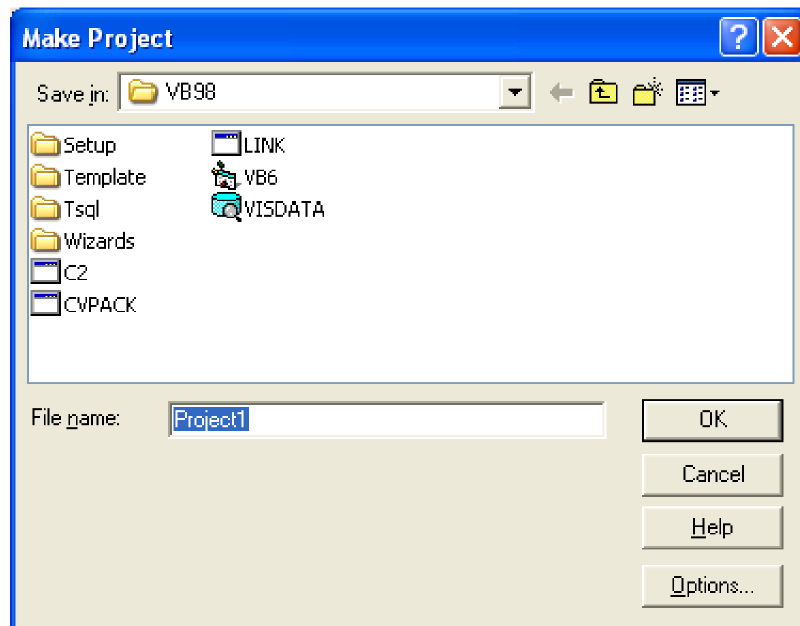
Your form will appear like a window from any other Microsoft application.

Creating Executable File

- Click File, Then Make Project1.exe…



- Specify the location and the name of the project, then click OK.



Saving your application

The last step in this chapter is to save your application so that you can use it for the exercises later in the book To do this Click on file , Click save project as .

Visual Basic first asks you to save the form and then the project file. Remember that each represents a separate file.

Specify the filename for the form as hello.frm. The file extension “frm” indicates that the file is a form file.

Always take care to ensure that you save all the files that make up a project.

WEEK 2 :

VISUAL BASIC CHARACRER SETS AND DATA TYPES

During this week you will learn :

- Basic character sets
- Data Types

VB Character set

Visual Basic Character set refers to those characters acceptable or allowed in VB programming eg GB is unknown in English likewise X is unknown in Yoruba language.

The Microsoft Visual basic character sets consists of the following:

Alphabet A/ a – Z/z (Both Upper and Lower Case)

Numeric Digits 0 – 9

Decimal point (.)

Grouping Characters (eg. Comma, colons, Semicolons, single and double apostrophe,

parenthesis)

Relational Operators (eg. =, <>,>,< etc)

Arithmetic Operators (eg. +, -, /, =. etc)

Blank character

Data Types

The Visual Basic language works with all kinds of data. Before you learn how to manipulate data, you will learn how to distinguish among the various data types, that Visual Basic supports. Some data falls into more than one category. For example, a dollar amount can be considered both a currency data type and a single data type. When you write a program, you need to decide which data type best fits your program's data values.

Visual Basic data types.

Type	Stores	Memory Requirement	Range of Values
Integer	Whole numbers	2 bytes	-32,768 to 32,767
Long	Whole numbers	4 bytes	Approximately +/- 2.1E9
Single	Decimal numbers	4 bytes	-3.402823E38 to -1.401298E-45 for negative values and 1.401298E-45 to 3.402823E38 for positive values
Double	Decimal numbers (double-precision floating-point)	8 bytes	-1.79769313486232E308 to -4.94065645841247E-324 for negative values and 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency	Numbers with up to 15 digits left of the decimal and 4 digits right of the decimal	8 bytes	922,337,203,685,477.5808 to 922,337,203,685,477.5807
String	Text information	1 byte per character	Up to 65,000 characters for fixed-length strings and up to 2 billion characters for dynamic strings
Byte	Whole numbers	1 byte	0 to 255
Boolean	Logical values	2 bytes	True or False
Date	Date and time information	8 bytes	Jan 1st 100 to December 31st 9999

The following data values can take on the integer data type:

21

0

-9455

32766

You can also store these integer data types as long data types, although doing so wastes storage and time unless you plan to change the values later to extremely large or small integer numbers that require the long data type.

The following data values must be stored in a long data type:

32768

-95445

492848559

The following data values can take on the single data type:

0.01

565.32

-192.3424

9543.5645

6.5440E-24

Of course, you can store these data values in double storage locations as well. Use double data types only when you need to store extra large or small values.

The following data values take on the double data type:

-5968.5765934211133

4.532112E+92

928374.344838273567899990

What's that E Doing in the Numbers?: The E stands for exponent. Years ago, mathematicians grew weary of writing long numbers. They developed a shortcut notation called scientific notation. Visual Basic supports scientific notation. You can use scientific notation for single and double numeric data values. When you use scientific notation, you don't have to write—or type, in the case of program writing—extremely long numbers.
When a single or double numeric value contains the letter E, followed by a number, that number is written in scientific notation. To convert a number written in scientific notation to its non-abbreviated form, multiply the number to the left of E by 10 raised to the power indicated by the number to the right of E. Therefore, 2.9876E+17 means to multiply 2.9876 by 10 raised to the 17th power, or 10^{17}. 10^{17} is a very large number—it is 10 followed by 16 zeroes. The bottom line is that 2.9876E+17 means the same as 29876 followed by 13 zeros. If a number uses a negative exponent, you multiply the number to left of E by 10 raised to negative power. Therefore, 2.9876E-17 means the same as 2.9876 multiplied by 10^{-17}—or 2.9876 divided by 10^{17}—which turns out to be a small number indeed.

The following data values can take on the currency data type:

123.45

0.69

63456.75

-1924.57

The currency data type can accept and track numeric values to four decimal places. However, you typically store dollar and cent values in the currency storage locations, and these kinds of values require only two decimal places.

Never use a currency data value along with a dollar sign. In other words, the following cannot be a currency value even though it looks like one:

$5,234.56

Visual Basic does not want you to use a dollar sign or commas in numeric data of any kind—unless, of course, your country uses the comma for the fractional portion of numbers. If your data contains anything other than numbers, a plus sign, a minus sign, or an exponent, Visual Basic cannot treat the data as if it were numeric; therefore, it cannot perform mathematical calculations with the data. Instead, Visual Basic treats the data as string data.

The following data values take on the string data type:

"London Bridge"

"1932 Sycamore Street"

"^%#@#$%3939$%^&^&"

Notice the quotation marks around the three string values. string values require the quotation marks, which tell Visual Basic where the string begins and ends. If you wanted to embed spaces at the beginning or end of a string, you indicate those spaces by including them inside the quotation marks. For example, here are three different strings, each with a different set of embedded spaces: " house", "house ", " house ". You cannot embed quotation marks directly inside a string; Visual Basic thinks the string ends as soon as it comes to the second quotation mark, even if it is not the actual end of the string.

The following data values can take on the variant data type:

"^%#@#$%3939$%^&^&"

123.45

4.532112E+92

21

03-Mar-1996

Do these values look familiar? The variant data type can hold data of any other data type. Think about the kind of data stored in label controls. You often want to display numbers, dollar amounts, times, and dates in labels on a form. You can always store variant data in these controls. The data that comes from these controls is also variant.

The following data values must be stored in a Boolean data type:

True

False

WEEK 3:

VARIABLE DECLARATION IN VISUAL BASIC/STORING AND RETRIVING DATA IN A VARIABLE

During this week you will learn :

- Declaring Variables
- Assigning Values to Variables
- Mathematical Expressions
- Conditional Operators

Declaring Variables

Definition: To define a variable means to create and name a variable.

A program can have as many variables as you need it to have. Before you can use a variable, you must request that Visual Basic create the variable by defining the variable first. When you define a variable, you tell Visual Basic these two things:

- The name of the variable
- The data type of the variable

Once you define a variable, that variable always retains its original data type. Therefore, a single-precision variable can hold only single-precision values. If you stored an integer in a single-precision variable, Visual Basic would convert the integer to a single-precision number before it stored the number in the variable. Such conversions are common, and they typically do not cause many problems.

The Dim statement defines variables. Using Dim, you tell Visual Basic

- that you need a variable
- what to name the variable

- what kind of variable you want

Dim—short for dimension—is a Visual Basic statement that you write in an application's Code window. Whenever you learn a new statement, you need to learn the format for that statement. Here is the format of the Dim statement:

```
Dim VarName AS DataType
```

VarName is a name that you supply. When Visual Basic executes the Dim statement at runtime, it creates a variable in memory and assigns it the name you give in the VarName location of the statement. DataType is one of the seven Visual Basic data types that you learned .

Variable names must follow the naming rules

- Names can be as short as one character or as long as 40 characters.

- Names must begin with a letter of the alphabet and can be in either uppercase or lowercase letters.

- After the initial letter, names can contain letters, numbers, or underscores in names.
- Names cannot be a reserved word.
- Never define two variables with the same name

Always use a Dim statement to define variables before you use variables. If the Options Environment Require Variable Definition option is set to Yes—as it is by default—Visual Basic will issue an error message whenever you use a variable that you have not defined. This option prevents you from inadvertently misspelling variable names and helps you avoid errors in your program.

Suppose that you give a partial program to another Visual Basic programmer to work on. If you want to require that all variables be defined but are unsure of the other programmer's Options Environment Require Variable Definition setting, select (general) from the Code window's Object dropdown list and add the following special statement:

```
Option Explicit
```

No matter how the Options Environment setting is set, the programmer cannot slip variables into your program without defining them properly in Dim statements. Again, requiring variable definitions helps eliminate bugs down the road, so you are wise to get in the habit of putting Option Explicit in the (general) section of every program's Code window.

The following statement defines a variable named ProductTotal:

```
Dim ProductTotal As Currency
```

From the Dim statement, you know that the variable holds currency data and that its name is ProductTotal. All Visual Basic commands and built-in routines require an initial capital letter. Although you don't have to name your variables with an initial capital letter, most Visual Basic programmers do for the sake of consistency. Additional caps help distinguish

individual words inside a name. (Remember that you cannot use spaces in the name of variable.)

The following statements define integer, single-precision, and double-precision variables:

```
Dim Length As Integer

Dim Price As Single

Dim Structure As Double
```

If you want to write a program that stores the user's text box entry for the first name, you would define a string like this:

```
Dim
FirstName As String
```

You can get fancy when you define strings. This FirstName string can hold any string from 0 to 65,500 characters long. You will learn in the next section how to store data in a string. FirstName can hold data of virtually any size. You could store a small string in FirstName—such as "Joe"—and then store a longer string in FirstName—such as "Mercedes". FirstName is a variable-length string.

Sometimes you want to limit the amount of text that a string holds. For example, you might need to define a string variable to hold a name that you read from the disk file. Later, you will display the contents of the string in a label on the form. The form's label has a fixed length, however—assuming that the AutoSize property is set to True. Therefore, you want to keep the string variable to a reasonable length. The following Dim statement demonstrates how you can add the * StringLength option when you want to define fixed-length strings:

```
Dim Title As String * 20
```

Assigning Values to Variables

Definition: A null string is a zero-length empty string that is often represented as "".

When you first define variables, Visual Basic stores zeroes in the numeric variables and null strings in the string variables. Use the assignment statement when you want to put other data values into variables. Variables hold data of specific data types, and many lines inside a Visual Basic program's Code window consist of assignment statements that assign data to variables. Here is the format of the assignment statement:

```
VarName = Expression
```

The Let command name is optional; it is rarely used these days. The VarName is a variable name that you have defined using the Dim statement. Expression can be a constant, another variable, or a mathematical expression.

Suppose that you need to store a minimum age value of 18 in an integer variable named MinAge. The following assignment statements do that:

```
MinAge = 18
```

To store a temperature in a single-precision variable named TodayTemp, you could do this:

```
TodayTemp = 42.1
```

The data type of Expression must match the data type of the variable to which you are assigning. In other words, the following statement is invalid. It would produce an error in Visual Basic programs if you tried to use it.

```
TodayTemp = "Forty-Two point One"
```

If TodayTemp were a single-precision variable, you could not assign a string to it. However, Visual Basic often makes a quick conversion for you when the conversion is trivial. For example, it is possible to perform the following assignment even if you have defined Measure to be a double-precision variable:

```
measurement
= 921.23
```

At first glance, it appears that 921.23 is a single-precision number because of its size. 921.23 is actually a variant data value. Recall that Visual Basic assumes all data constants are variant unless you explicitly add a suffix character to the constant to make the constant a different data type. Visual Basic can easily and safely convert the variant value to double-precision. That's just what Visual Basic does, so the assignment works fine.

In addition to constants, you can assign other variables to variables. Consider the following code:

```
Dim Sales As Single, NewSales As Single

Sales = 3945.42

NewSales = Sales
```

When the third statement finishes, both Sales and NewSales have the value 3945.42.

Feel free to assign variables to controls and controls to variables. Suppose, for example, that the user types the value 18.34 in a text box's Text property. If the text box's Name property is txtFactor, the following statement stores the value of the text box in a variable named FactorVal:

```
FactorVal = txtFactor.Text
```

Suppose that you defined Title to be a string variable with a fixed length of 10, but a user types Mondays Always Feel Blue in a text box's Text property that you want to assign to Title. Visual Basic stores only the first ten characters of the control to Title and truncates the rest of the title. Therefore, Title holds only the string "Mondays Al".

This is the first of several program code reviews that you will find throughout the rest of the book.

```
Sub cmdJoke_Click ()

        cmdJoke.Caption = "Bird Dogs Fly"

End Sub
```

If you have a variable that will contain simple true/false, yes/no, or on/off information, you can declare it to be of type Boolean. The default value of Boolean is False. In the following example, `blnRunning` is a Boolean variable which stores a simple yes/no setting.

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

Mathematical Expressions

Data values and controls are not the only kinds of assignments that you can make. With the Visual Basic math operators, you can calculate and assign expression results to variables when you code assignment statements that contain expressions.

Definition: An operator is a word or symbol that does math and data manipulation.

It is easy to make Visual Basic do your math. This table describes Visual Basic's primary math operators. There are other operators, but the ones in Table will suffice for most of the programs that you write. Look over the operators. You are already familiar with most of them because they look and act just like their real-world counterparts.

The primary math operators.

Operator	Example	Description
+	Net + Disc	Adds two values
-	Price - 4.00	Subtracts one value from another value
*	Total * Fact	Multiplies two values
/	Tax / Adjust	Divides one value by another value
^	Adjust ^ 3	Raises a value to a power
& or +	Name1 & Name2	Concatenates two strings

Suppose that you wanted to store the difference between the annual sales (stored in a variable named AnnualSales) and cost of sales (stored in a variable named CostOfSales) in a variable named NetSales. Assuming that all three variables have been defined and initialized, the following assignment statement computes the correct value for NetSales:

```
NetSales = AnnualSales - CostOfSales
```

This assignment tells Visual Basic to compute the value of the expression and to store the result in the variable named NetSales. Of course, you can store the results of this expression in Caption or Text properties, too.

If you want to raise a value by a power—which means to multiply the value by itself a certain number of times—you can do so. The following code assigns 10000 to Value because ten raised to the fourth power—10x10x10x10—is 10,000:

```
Years = 4

Value = 10 ^ Years
```

No matter how complex the expression is, Visual Basic computes the entire result before it stores that result in the variable at the left of the equals sign. The following assignment statement, for example, is rather lengthy, but Visual Basic computes the result and stores the value in the variable named Ans:

```
Ans = 8 * Factor - Pi + 12 * MonthlyAmts
```

Combining expressions often produces unintended results because Visual Basic computes mathematical results in a predetermined order. Visual Basic always calculates exponentiation first if one or more ^ operators appear in the expression. Visual Basic then computes all multiplication and division before any addition and subtraction.

Visual Basic assigns 13 to Result in the following assignment:

```
Result = 3 + 5 * 2
```

At first, you might think that Visual Basic would assign 16 to Result because 3 + 5 is 8 and 8 * 2 is 16. However, the rules state that Visual Basic always computes multiplication—and division if division exists in the expression—before addition. Therefore, Visual Basic first computes the value of 5 * 2, or 10, and next adds 3 to 10 to get 13. Only then does it assign the 13 to Result.

If both multiplication and division appear in the same expression, Visual Basic calculates the intermediate results from left to right. For example, Visual Basic assigns 20 to the following expression:

```
Result = 8
/ 2 + 4 + 3 * 4
```

Visual Basic computes the division first because the division appears to the left of the multiplication. If the multiplication appeared to the left of the division, Visual Basic would

have multiplied first. After Visual Basic calculates the intermediate answers for the division and the multiplication, it performs the addition and stores the final answer of 20 in Result.

Note: The order of computation has many names. Programmers usually use one of these terms: order of operators, operator precedence, or math hierarchy.

It is possible to override the operator precedence by using parentheses. Visual Basic always computes the values inside any pair of parentheses before anything else in the expression, even if it means ignoring operator precedence. The following assignment statement stores 16 in Result because the parentheses force Visual Basic to compute the addition before the multiplication:

```
Result = (3 + 5) * 2
```

The following expression stores the fifth root of 125 in the variable named root5:

```
root5 = 125 ^ (1/5)
```

As you can see from this expression, Visual Basic supports fractional exponents.

Definition: Concatenation means the joining together of two or more strings.

One of Visual Basic's primary operators has nothing to do with math. The concatenation operator joins one string to the end of another. Suppose that the user entered his first name in a label control named lblFirst and his last name in a label control named lblLast. The following concatenation expression stores the full name in the string variable named FullName:

```
FullName = lblFirst & lblLast
```

There is a problem here, though, that might not be readily apparent—there is no space between the two names. The & operator does not automatically insert a space because you don't always want spaces inserted when you concatenate two strings. Therefore, you might have to concatenate a third string between the other two, as in

```
FullName = lblFirst & " " & lblLast
```

Visual Basic actually supports a synonym operator, the plus sign, for concatenation. In other words, the following assignment statement is identical to the previous one:

```
FullName = lblFirst + " " + lblLast
```

Even Microsoft, the maker of Visual Basic, recommends that you use & for string concatenation and reserve + for mathematical numeric additions. Using the same operator for two different kinds of operations can lead to confusion.

Review: The math operators enable you to perform all kinds of calculations and assignments. Visual Basic means never having to use a calculator again! When you use expressions, however, remember operator precedence. When in doubt, use parentheses to indicate exactly which operations in an expression you want Visual Basic to calculate first.

Conditional Operators

You can combine or modify search conditions using the standard logical operators listed in the following table. The operators are listed in the order that they are evaluated.

Operator	Meaning	Example
NOT	Logical opposite of condition	`NOT True = False` `NOT False = True`

AND	Both conditions must apply (True)	`True AND True = True`
		`True AND False = False`
		`False AND True = False`

OR	Either condition can apply (True)	`True OR True = True`
		`True OR False = True`
		`False OR True = True`
		`False OR False = False`

WEEK 4 :

BASIC CONTROLS

During this week you will learn :

- Properties of the Form
- Intrinsic Controls
- Label control
- Text Box
- Focus In on Controls
- The Command Button
- The PictureBox Control
- The Image Control

Properties of the Form

The form is yet another Visual Basic object. As such, it has property settings that you can set and change while you design the application and during the program's execution. This section explains all the form's property settings in detail.

This table describes the property settings of the form that appear in the Properties window when you click the Form window and press F4. The form has more properties than the command button, label, and text box controls, whose properties you saw in the previous unit. As with all control property values, you never need to worry about all these properties at once. Most of the time, the default values are satisfactory for your applications.

Properties of the form.

Property	Description
AutoRedraw	If True, Visual Basic automatically redraws graphic images that reside on the form when another window hides the image or when the user resizes the object. If False (the default), Visual Basic does not automatically redraw.
BackColor	The background color of the form. You can enter a hexadecimal Windows color value or select from the color palette.
BorderStyle	Set to 0 for no border or border elements such as a control menu or minimize and maximize buttons, 1 for a fixed-size border, 2 (the default) for a sizable border, or 3 for a fixed-size border that includes a double-size edge.
Caption	The text that appears in the form's title bar. The default Caption is the Name of the form.
Enabled	If True (the default), the form can respond to events. Otherwise, Visual Basic halts event processing for the form.
FillColor	The color value used to fill shapes drawn on the form.
FillStyle	Contains eight styles that determine the appearance of the interior patterns

	of shapes drawn on the form.
FontBold	Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command.
FontItalic	Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command.
FontName	Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command.
FontSize	Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command.
FontStrikethru	Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command.
FontTransparent	Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command.
FontUnderline	Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command.
ForeColor	The color of foreground text that you display on the form if you use the Print command.
Height	The form’s height in twips.
HelpContextID	Provides the identifying number for the help text if you add advanced context-sensitive help to your application.
Icon	The picture icon that the user sees after minimizing the form.
KeyPreview	If False (the default), the control with the focus receives these events: KeyDown, KeyUp, and KeyPress before the form does . If True, the form receives the events before the focused control.
Left	The number of twips from the left edge of the screen to the left edge of the form.
MaxButton	If True (the default), the maximize button appears on the form at runtime. If False, the user cannot maximize the form window.
MinButton	If True (the default), the minimize button appears on the form at runtime. If False, the user cannot minimize the form window.
MousePointer	The shape to which the mouse cursor changes if the user moves the mouse cursor over the form. The possible values range from 0 to 12 and represent the different shapes the mouse cursor can take on.
Name	The name of the form. By default, Visual Basic generates the name Form1.
Picture	A picture file that displays on the form's background.
ScaleMode	Enables you to determine how to measure coordinates on the form. You can choose from eight values. The default unit of measurement is twips, indicated by 1. The other Scale... properties measure use twips.
Top	The number of twips from the top edge of the screen to the top of the form.
Visible	True or False, indicating whether the user can see and, therefore, use the form.

Width	The width of the form in twips.
WindowState	Describes the startup state of the form when the user runs the program. If set to 0 (the default), the form first appears the same size as you designed it. If set to 1, the form first appears minimized. If set to 2, the form first appears maximized.

ScaleMode enables you to determine how to measure coordinates on the form. You can choose from eight values. The default unit of measurement is twips. This table describes the possible units of measurement.

The ScaleMode property values.

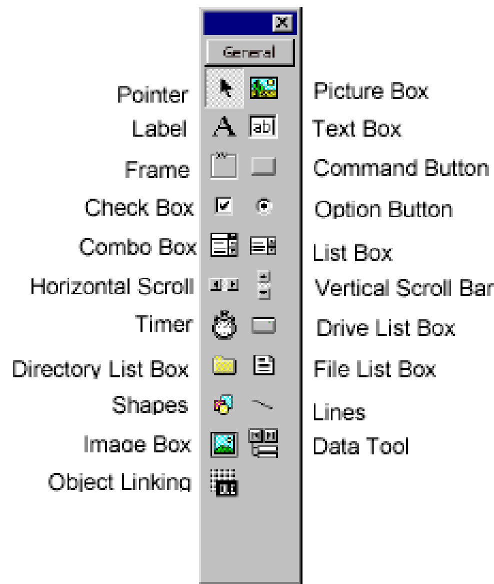
Value	Description
0	Customized values
1	Twips (the default)
2	Points
3	Pixels
4	A standard character that is 120 twips wide and 240 twips high
5	Inches
6	Millimeters
7	Centimeters

Intrinsic Controls

In Microsoft Visual Basic jargon, intrinsic controls (or built-in controls) are those controls visible in the Toolbox window when you launch the environment. This important group includes controls, such as Label, TFextbox, and CommandButton controls, that are used in nearly every application. As you know, Visual Basic can be extended using additional Microsoft ActiveX Controls (formerly known as OCX controls, or OLE custom controls) either provided in the Visual Basic package or available as commercial, shareware, or even freeware third-party products. Even if such external controls are often more powerful than built-in controls, intrinsic controls have a few advantages that you should always take into account:

- Support for intrinsic controls is included in the MSVBVM60.DLL, the runtime file that's distributed with every Visual Basic application. This means that if a program exclusively uses intrinsic controls, you don't need to distribute any additional OCX files, which greatly simplifies the installation process and reduces disk requirements.
- In general, Visual Basic can create and display intrinsic controls faster than it can external ActiveX controls because the code for their management is already in the Visual Basic runtime module and doesn't have to be loaded when a form references an intrinsic control for the first time. Also, applications based on intrinsic controls usually perform faster on machines with less memory; no extra memory is needed by additional OCX modules.
- Because programs based exclusively on intrinsic controls require fewer ancillary files, they can be downloaded faster through the Internet. Moreover, if end users previously installed any other Visual Basic application, Visual Basic runtime files are already installed on the target machine, which reduces download times to the minimum.

For all these reasons, it's important to learn how to make the best of intrinsic controls. In this chapter, I focus on their most important properties, methods, and events, and I also show how to address some common programming issues using intrinsic controls exclusively.



Label control

The label holds text on the form. Although there are several ways to display text, the label control enables you to post messages on the form that you can change by means of Visual Basic code. The user, however, cannot change the value of a text control.

Label controls are vital to Visual Basic applications because you are always putting text on the form for the user to read. Here are some of the uses for the label control:

Titles (boxed and unboxed)

Data descriptions

Color warning messages

Graphic descriptions

Instructions

Labels are extremely easy to place and initialize. If you don't want a message to appear in a label when the user first starts the application, be sure to delete all the text from the label's Caption property. This table lists the properties available for labels within the Properties window.

label properties.

Property	Description
Alignment	Set to 0 for left-justification (the default), 1 for right-justification, or 2 for centering the Caption within the label. Putting a border around the label or shading the label a color often makes the justification stand out.
AutoSize	If True, the control automatically adjusts its own size to shrinkwrap around the contents of its caption. If False (the default), the control clips off the right

	of the text if the label is not large enough to hold the entire caption.
BackColor	The background color of the label. It is a hexadecimal number that represents one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you are ready to set the BackColor property. The default background color is the same as the form's default background color.
BackStyle	If set to 0, meaning transparent, the form’s background color comes through the label’s background. If set to 1 (the default), the label's background color hides the form behind the label.
BorderStyle	Either 0 (the default) for no border or 1 for a fixed single-line border.
Caption	The text that appears on a label.
Enabled	If True (the default), the label control can respond to events. Otherwise, Visual Basic halts event processing for that particular control.
FontBold	If True (the default), the Caption displays in boldfaced characters.
FontItalic	If True (the default), the caption displays in italicized character.
FontName	The name of the label control’s style. You typically use the name of a Windows True Type font.
FontSize	The size, in points, of the font used for the label control's caption.
FontStrikethru	If True (the default), the caption displays in strikethrough letters. In other words, the characters have a line drawn through them.
FontUnderline	If True (the default), the caption displays in underlined letters.
ForeColor	The color of the text inside the caption.
Height	The height of the label control in twips.
Index	If the label control is part of a control array, the Index property provides the numeric subscript for each particular label control.
Left	The number of twips from the left edge of the Form window to the left edge of the label control.
MousePointer	The shape to which the mouse cursor changes if the user moves the mouse cursor over the label control. The possible values range from 0 to 12 and represent the different shapes that the mouse cursor can take on.
Name	The name of the control. By default, Visual Basic generates the names Label1, Label2, and so on, as you add subsequent label controls to the form.
TabIndex	The focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing value of the TabIndex of the control. No two controls on the same form can have the same TabIndex value.
Top	The number of twips from the top edge of a label control to the top of the form.
Visible	True or False, indicating whether the user can see and, therefore, use the label control.
Width	The width of the label control in twips.
WordWrap	If True, the text wraps to hold the entire caption. If False (the default), the

	text does not wrap but is truncated to fit the caption.

Text Box

A TextBox control, sometimes called an edit field or edit control, displays information

Text box controls display default values and accept user input. Text box controls enable you to determine how the user enters data and responds to questions and controls that you display.

When you display a text box on a form, you give the user a chance to accept a default value—the text box's initial Text property—or to change it to something else. The user can enter text of any data type—numbers, letters, and special characters. He can scroll left and right by using the arrow keys, and he can use the Ins and Del keys to insert and delete text within the text box control.

Most of the text box's properties work like the label control's properties. Unlike the label, however, the text box properties describe data-entry properties so that the control can deal with user input instead of simple text display. This table describes the property values for the text box control.

Text box properties.

Property	Description
Alignment	Set to 0 for left-justification (the default), 1 for right-justification, or 2 for centering the Caption within the text box. If MultiLine contains False, Visual Basic ignores the Alignment setting.
BackColor	The background color of the text box. It is a hexadecimal number that represents one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you are ready to set the BackColor property. The default background color is the same as the form's default background color.
BorderStyle	Either 0 (the default) for no border or 1 for a fixed single-line border.
Enabled	If True (the default), the text box can respond to events. Otherwise, Visual Basic halts event processing for that particular control.
FontBold	If True (the default), the Text displays in boldfaced characters.
FontItalic	If True (the default), the Text displays in italicized characters.
FontName	The name of the text box's style. You typically use the name of a Windows True Type font.
FontSize	The size, in points, of the font used for the text box control's Text value.
FontStrikethru	If True (the default), the text value displays in strikethrough letters. In other words, the characters have a line drawn through them.
FontUnderline	If True (the default), the text value displays in underlined letters.
ForeColor	The color of the text inside the Text property.

Height	The height of the text box in twips.
HelpContextID	If you add advanced context-sensitive help to your application, the HelpContextID provides the identifying number for the help text.
HideSelection	Keeps text highlighted even when the text box loses its focus.
Index	If the text box is part of a control array, the Index property provides the numeric subscript for each particular text box.
Left	The number of twips from the left edge of the Form window to the left edge of the text box.
MaxLength	If set to 0 (the default), the limit of the Text value can be as great as approximately 32,000 characters. Otherwise, the MaxLength specifies how many characters the user can enter in the text box.
MousePointer	The shape to which the mouse cursor changes if the user moves the mouse cursor over the text box. The possible values range from 0 to 12 and represent the different shapes that the mouse cursor can take on.
MultiLine	If True, the text box can display more than one line of text. If False (the default), the text box contains a single, and often long, line of text. The text can contain a carriage return.
Name	The name of the control. By default, Visual Basic generates the names Text1, Text2, and so on, as you add subsequent text boxes to the form.
PasswordChar	If you enter a character, such as an asterisk (*) for the PasswordChar, Visual Basic does not display the user's text but instead displays the PasswordChar as the user types the text. Use text boxes with a PasswordChar set when the user needs to enter a password and you don't want others looking over his shoulder to peek at the password.
ScrollBars	Set to 0 (the default) for no scroll bars, 1 for a horizontal scroll bar, 2 for a vertical scroll bar, or 3 for both kinds of scroll bars.
TabIndex	The focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the value of the TabIndex of the control. No two controls on the same form can have the same TabIndex value.
TabStop	If True, the user can press Tab to move the focus to this label control. If False, the label control cannot receive the focus.
Text	The initial value that the user sees in the text box. The default value is the name of the control. The value continues to update as the user enters new text at runtime.
Top	The number of twips from the top edge of a text box to the top of the form.
Visible	True or False, indicating whether the user can see and, therefore, use the text box.
Width	The width of the text box in twips.

Focus In on Controls

As you learn about controls, you often hear the term focus. Learning about focus now saves you a lot of trouble later.

Definition: Focus refers to the control that is currently highlighted.

The control with the focus is the next control that will accept the user's response. Most Windows users instinctively understand focus even though very few have thought much about focus. The control with the focus is always the control that is highlighted. The focus often moves from control to control as the user works. The focus determines where the next action will take place.

Only one control can have the focus at any one time, and not every control can get the focus.

Definition: The focus order determines the control next in line for the focus.

Every form has a focus order that determines the next control that will receive the focus.

When you see a dialog box such as a File Open dialog box, the OK button is almost always the command button that has the focus. You can press Enter to select the OK command button, click a different command button (such as a Cancel command button), or press Tab until the command button that you want has the focus.

Through property settings, you can determine the focus order and whether a control can receive the focus. Your form might have some controls that you don't want the user to be able to highlight; therefore, you prevent them from getting the focus.

The Command Button

The command button is the cornerstone of most Windows applications. With the command button, your user can respond to events, signal when something is ready for printing, and tell the program when to terminate. Although all command buttons work in the same fundamental way—they visually depress when the user clicks them, and they trigger events such as Click events—numerous properties for command buttons uniquely define each one and its behavior.

You are already familiar with several command button properties. You have seen that command buttons vary in size and location. You might want to open a new project, add a command button to the Form window by double-clicking the command button on the toolbox, and press F4 to scroll through the Properties window.

Several of the property settings can accept a limited range of numeric values. Most of these values are named in CONSTANT.TXT, and they are mentioned in this book as well. Some property settings accept either True or False values, indicating a yes or no condition. For example, if the Cancel property is set to True, that command button is the cancel command button and all other command buttons must contain a False Cancel property because only one command button on a form can have a True Cancel property.

Command button properties.

Property	Description
BackColor	The command button is one of the few controls for which the background color property means very little. When you change the background color, only the dotted line around the command button's caption changes color.
Cancel	If True, Visual Basic automatically clicks this command button when the user presses Esc. Only one command button can have a True Cancel property value at a time. All command buttons initially have their Cancel property set to False.
Caption	The text that appears on the command button. If you precede any character in the text with an ampersand (&),it acts as the access key. Therefore, the access key for a command button with a Caption property set to E&xit is Alt+X. The default Caption value is the command button's Name value.
Default	The command button with the initial focus when the form first activates has a Default property setting of True. All command buttons initially have False Default property values until you change one of them.

Definition: An icon is a picture on the screen.

DragIcon	The icon that appears when the user drags the command button around on the form.

Because you only rarely enable the to user move a command button, you won't use the Drag... property settings very much.

Enabled	If True (the default), the command button can respond to events. Otherwise, Visual Basic halts event processing for that particular control.
FontBold	If True (the default), the Caption displays in boldfaced characters.
FontItalic	If True (the default), the caption displays in italicized characters.
FontName	The name of the style of the command button caption. You typically use the name of a Windows True Type font.

(margin)Definition: A point is 1/72 of one inch.

FontSize	The size, in points, of the font used for the command button's caption.
FontStrikethru	If True (the default), the caption displays in strikethrough letters. In other words, characters have a line drawn through them.
FontUnderline	If True (the default), the caption displays in underlined letters.
Height	The height of the command button in twips.
HelpContextID	If you add advanced context-sensitive help to your application, the

	HelpContextID provides the identifying number for the help text.
Index	If the command button is part of a control array, the Index property provides the numeric subscript for each particular command button.
Left	The number of twips from the left edge of the Form window to the left edge of the command button.
MousePointer	The shape to which the mouse cursor changes if the user moves the mouse cursor over the command button. The possible values range from 0 to 12 and represent the different shapes that the mouse cursor can take on.
Name	The name of the control. By default, Visual Basic generates the names Command1, Command2, and so on, as you add subsequent command buttons to the form.
TabIndex	The focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the TabIndex values of the controls. No two controls on the same form can have the same TabIndex value.
TabStop	If True, the user can press Tab to move the focus to this command button. If False, the command button cannot receive the focus.
Tag	Not used by Visual Basic. The programmer can use it for an identifying comments applied to the command button.
Top	The number of twips from the top edge of a command button to the top of the form.
Visible	True or False, indicating whether the user can see and, therefore, use the command button.
Width	The width of the command button in twips.

The PictureBox Control

PictureBox controls are among the most powerful and complex items in the Visual Basic Toolbox window. In a sense, these controls are more similar to forms than to other controls. For example, PictureBox controls support all the properties related to graphic output, including AutoRedraw, ClipControls, HasDC, FontTransparent, CurrentX, CurrentY, and all the Drawxxxx, Fillxxxx, and Scalexxxx properties. PictureBox controls also support all graphic methods, such as Cls, PSet, Point, Line, and Circle and conversion methods, such as ScaleX, ScaleY, TextWidth, and TextHeight. In other words, all the techniques that I described for forms can also be used for PictureBox controls (and therefore won't be covered again in this section).

Loading images

Once you place a PictureBox on a form, you might want to load an image in it, which you do by setting the Picture property in the Properties window. You can load images in many different graphic formats, including bitmaps (BMP), device independent bitmaps (DIB), metafiles (WMF), enhanced metafiles (EMF), GIF and JPEG compressed files, and icons (ICO and CUR). You can decide whether a control should display a border, resetting the BorderStyle to 0-None if necessary. Another property that comes handy in this phase is

AutoSize: Set it to True and let the control automatically resize itself to fit the assigned image.

You might want to set the Align property of a PictureBox control to something other than the 0-None value. By doing that, you attach the control to one of the four form borders and have Visual Basic automatically move and resize the PictureBox control when the form is resized. PictureBox controls expose a Resize event, so you can trap it if you need to move and resize its child controls too.

You can do more interesting things at run time. To begin with, you can programmatically load any image in the control using the LoadPicture function:

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

and you can clear the current image using either one of the following statements:

```
' These are equivalent.
Picture1.Picture = LoadPicture("")
Set Picture1.Picture = Nothing
```

The LoadPicture function has been extended in Visual Basic 6 to support icon files containing multiple icons. The new syntax is the following:

```
LoadPicture(filename, [size], [colordepth], [x], [y])
```

where values in square brackets are optional. If filename is an icon file, you can select a particular icon using the size or colordepth arguments. Valid sizes are 0-vbLPSmall, 1-vbLPLarge (system icons whose sizes depend on the video driver), 2-vbLPSmallShell, 3-vbLPLargeShell (shell icons whose dimensions are affected by the Caption Button property as set in the Appearance tab in the screen's Properties dialog box), and 4-vbLPCustom (size is determined by x and y). Valid color depths are 0-vbLPDefault (the icon in the file that best matches current screen settings), 1-vbLPMonochrome, 2-vbLPVGAColor (16 colors), and 3-vbLPColor (256 colors).

You can copy an image from one PictureBox control to another by assigning the target control's Picture property:

```
Picture2.Picture = Picture1.Picture
```

The PaintPicture demonstration program shows several graphic effects.

The Image Control

Image controls are far less complex than PictureBox controls. They don't support graphical methods or the AutoRedraw and the ClipControls properties, and they can't work as containers, just to hint at their biggest limitations. Nevertheless, you should always strive to use Image controls instead of PictureBox controls because they load faster and consume less memory and system resources. Remember that Image controls are windowless objects that are actually managed by Visual Basic without creating a Windows object. Image controls can load bitmaps and JPEG and GIF images.

When you're working with an Image control, you typically load a bitmap into its Picture property either at design time or at run time using the LoadPicture function. Image controls don't expose the AutoSize property because by default they resize to display the contained image (as it happens with PictureBox controls set at AutoSize = True). On the other hand, 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. In fact, 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
```

Image controls support all the usual mouse events. For this reason, many Visual Basic developers have used Image controls to simulate graphical buttons and toolbars. Now that Visual Basic natively supports these controls, you'd probably better use Image controls only for what they were originally intended.

WEEK 5:

ADDING AND ACTIVATING VB CONTROLS IN A PROGRAM

During this week you will learn :

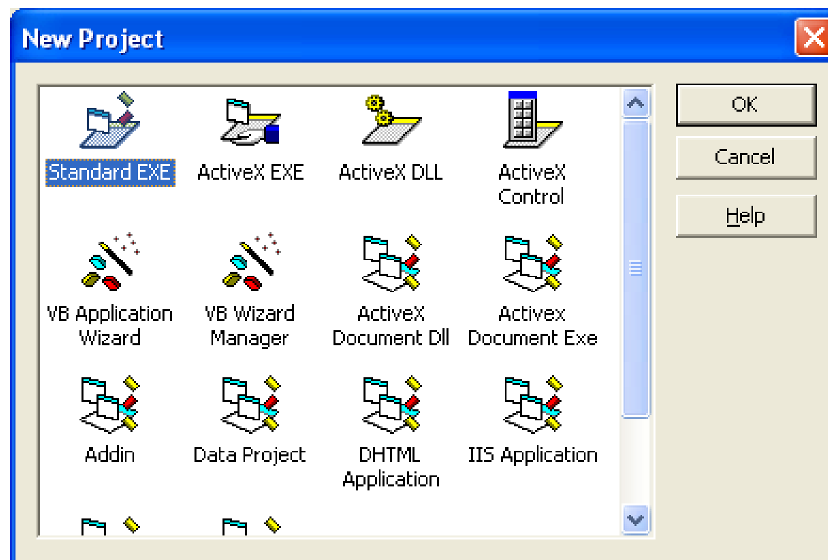
- Creating non-wizard applications
- Adding Controls to applications
- Managing Controls
- Change the control properties
- Common Properties
- Handling Control Event
- Common Events
- Writing Code

Creating non-wizard applications

You can create a single project .

1. On the File menu, click Create Project to display the Create Project dialog box.

2. In the New Project box, Click Standard.EXE and then Click OK.



Adding Controls to applications

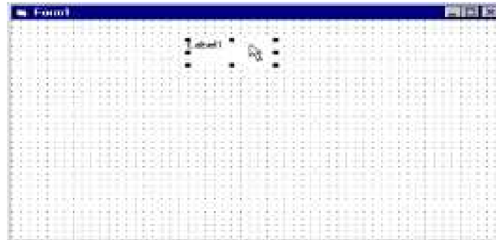
To place a control on the form, you can do one of two things:

- Double-click the Toolbox control. Visual Basic places and centers the control on your form. You then have to move and size the control to your preferences.
- Click the control and move the mouse pointer to the Form window. Drag to draw the control's outline on your form in the placement and size you need.

Managing Controls

Moving controls

Controls can be moved around a form by dragging them with the mouse.



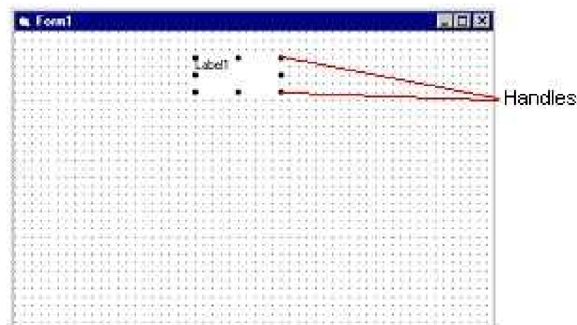
Deleting Controls

First selected the control you want to delete, then you can use many ways

- Press Delete Key
- Right Click then select Delete.
- From Edit menu Select Delete

Resizing controls

Before a control can be resized it must be selected. When a control is selected it will be surrounded by black dots or handles. Clicking and dragging one of these handles will adjust the size of the control.



Change the control properties

The properties window lists all the properties a control has and their value. The default value of a property can be changed. Just select the property first then change its value.

Common Properties

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

All visible objects—forms and controls—expose these properties, which affect the object's position and size. These values are always relative to the object's container—that is, the screen for a form and the parent form for a control. A control can also be contained in another control, which is said to be its container: In this case, Top and Left properties are relative to such a container control. 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.

While you can enter numeric values for these properties right in the Properties window at design time, you often set them in a visual manner by moving and resizing the control on its parent form. Keep in mind that Visual Basic also offers many interactive commands in the Format menu that let you resize, align, and space multiple controls in one operation. You can also access and modify these properties through code to move or resize objects at run time:

```
' 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
```

Note that while all controls—even invisible ones—expose these four properties at design time in the Properties window, controls that are inherently invisible—Timer controls, for example—don't support these properties at run time, and you can't therefore read or modify them through code.

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. As far as I know, this is the only example of a property that appears in the Properties window but can't be modified at design time. This happens because the height of a ComboBox control depends on the control's Font attributes. Remember this exception when writing code that modifies the Height property for all the controls in a form.

The ForeColor and BackColor Properties

Most visible objects expose ForeColor and BackColor properties, which affect the color of the text and the color of the background, respectively. The colors of a few controls—scroll bars, for example—are dictated by Microsoft Windows, however, and you won't find ForeColor and BackColor entries in the Properties window. In other cases, 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. (Because the default value for the Style property is 0-Standard, it might take you a while until you understand why the BackColor property doesn't affect the background color in the usual manner.)

When you're setting one of these two properties in the Properties window, you can select either a standard Windows color or a custom color using the System tab in the first case and the Palette tab in the second, as you can see in Figure. My first suggestion is always use a standard color value unless you have a very good reason to use a custom color. System colors display well on any Windows machine, are likely to conform to your customers' tastes, and contribute to making your application look well integrated in the system. My second suggestion is if you want to use custom colors, develop a consistent color scheme and use it throughout your application. I also have a third suggestion: Never mix standard and custom colors on the same form, and don't use a standard color for the ForeColor property and a custom color for the BackColor property of the same control (or vice versa), because the user might change the system palette in a way that makes the control completely unreadable.

You can choose from several ways to assign a color in code. Visual Basic provides a set of symbolic constants that correspond to all the colors that appear in the System tab in the Properties window at design time:

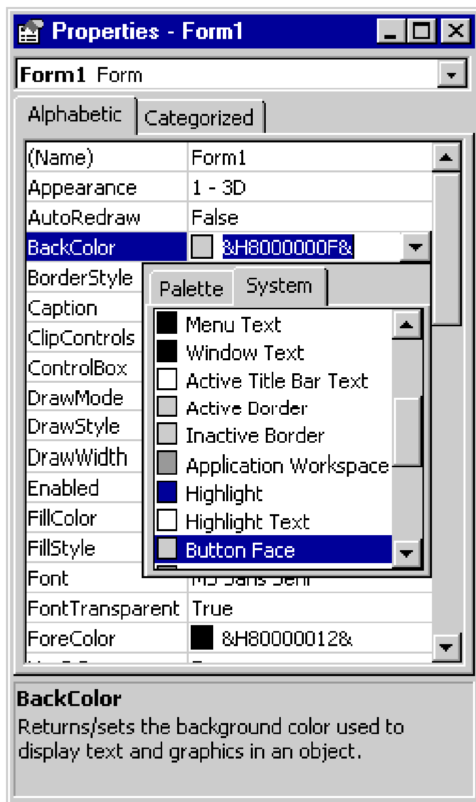
```
' Make Label1 appear in a selected state.
Label1.ForeColor = vbHighlightText
Label1.BackColor = vbHighlight
```

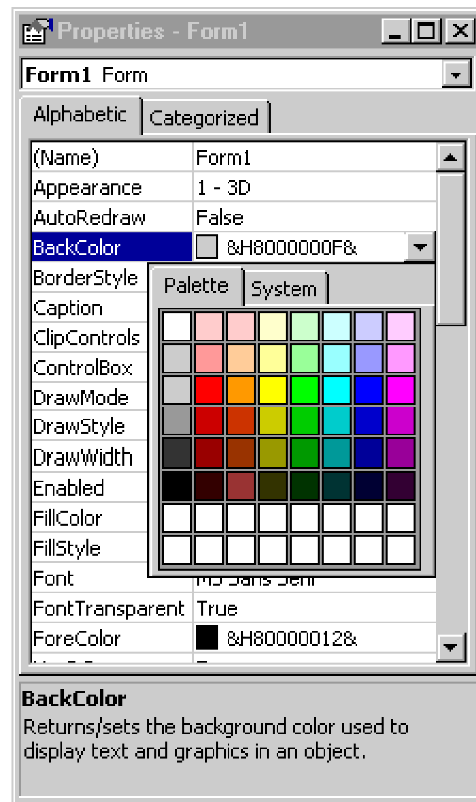
All the symbolic constants are shown in the following table.

Constant	Hex Value	Description
vb3DDKShadow	&H80000015	Darkest shadow
vb3Dface	&H8000000F	Dark shadow color for 3-D display elements
vb3Dhighlight	&H80000014	Highlight color for 3-D display elements
vb3Dlight	&H80000016	Second lightest of the 3-D colors after vb3Dhighlight
vb3Dshadow	&H80000010	Color of automatic window shadows
vbActiveBorder	&H8000000A	Active window border color
vbActiveTitleBar	&H80000002	Active window caption color
vbActiveTitleBarText	&H80000009	Text color in active caption, size box, scroll bar arrow box
vbApplicationWorkspace	&H8000000C	Background color of multiple-document interface (MDI) applications
vbButtonFace	&H8000000F	Face shading on command buttons
vbButtonShadow	&H80000010	Edge shading on command buttons

vbButtonText	&H80000012	Text color on push buttons
vbDesktop	&H80000001	Desktop color
vbGrayText	&H80000011	Grayed (disabled) text
vbHighlight	&H8000000D	Background color of items selected in a control
vbHighlightText	&H8000000E	Text color of items selected in a control
vbInactiveBorder	&H8000000B	Inactive window border color
vbInactiveCaptionText	&H80000013	Color of text in an inactive caption
vbInactiveTitleBar	&H80000003	Inactive window caption color
vbInactiveTitleBarText	&H80000013	Text color in inactive window caption, size box, scroll bar arrow box
vbInfoBackground	&H80000018	Background color of ToolTips
vbInfoText	&H80000017	Color of text in ToolTips
vbMenuBar	&H80000004	Menu background color
vbMenuText	&H80000007	Text color in menus
vbScrollBars	&H80000000	Scroll bar gray area color
vbTitleBarText	&H80000009	Text color in active caption, size box, scroll bar arrow box
vbWindowBackground	&H80000005	Window background color
vbWindowFrame	&H80000006	Window frame color
vbWindowText	&H80000008	Text color in windows

You can also browse them in the Object Browser window, after clicking the SystemColorConstants item in the leftmost list box. (If you don't see it, first select <All libraries> or VBRUN in the top ComboBox control). Note that all the values of these constants are negative.





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)
```

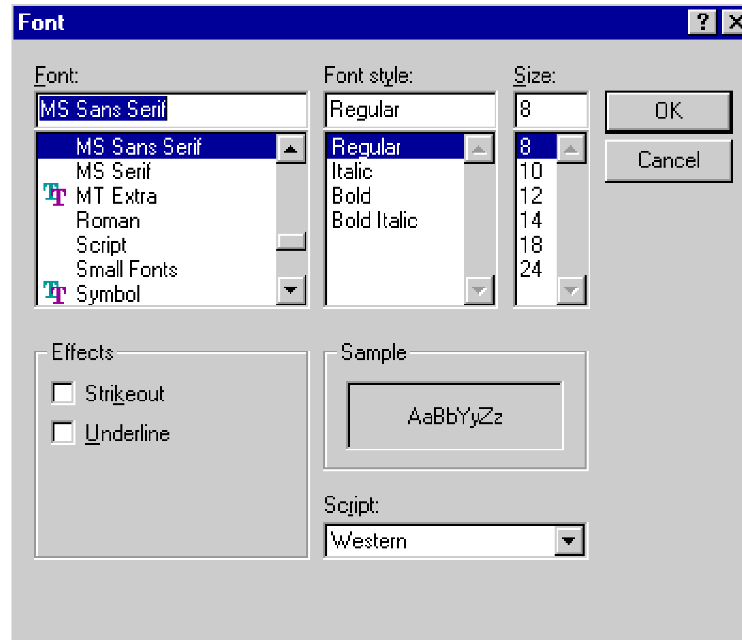
The Font Property

Forms and those controls that can display strings of characters expose the Font property. At design time, you set font attributes using a common dialog box, which you can see in Figure. Dealing with fonts at run time, however, is less simple because you must account for the fact that 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
```

At design time the Font dialog box lets you modify all font attributes at once and preview the result.



It should be made clear, however, that the preceding code actually assigns the same Font objects to both controls. This means that if you later change Text1's font attributes, the appearance of Text2 will also be affected. This behavior is perfectly consistent with the Font object's nature, even though the reasons for. You can take advantage of this approach—for example, if all the controls in your form always use the same font—but you should absolutely avoid it when the controls in question are supposed to have independent font attributes.

Visual Basic 6 still supports old-style Font properties such as FontName, FontSize, FontBold, FontItalic, FontUnderline, and FontStrikethru, but you can modify them only through code because they don't appear in the Properties window at design time. You can use the syntax that you like most because the two forms are perfectly interchangeable. In this book, however, I mostly follow the newer object-oriented syntax.

The Font.Size property (or the equivalent FontSize property) is peculiar because in general you can't be sure that Visual Basic is able to create a font of that particular size, especially if you aren't working with a TrueType font. The short code snippet below proves this.

```
Text1.Font.Name = "Courier"
Text1.Font.Size = 22
Print Text1.Font.Size  ' Prints 19.5
```

The Caption and Text Properties

The Caption property is a string of characters that appears inside a control (or in the title bar of a form) and that the user can't directly modify. Conversely, the Text property corresponds to the "contents" of a control and is usually editable by the end user. No intrinsic control exposes both a Caption and a Text property, so in practice a look at the Properties window can resolve your doubts as to what you're working with. Label, CommandButton, CheckBox, OptionButton, Data, and Frame controls expose the Caption property, whereas TextBox, ListBox, and ComboBox controls expose the Text property.

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
```

The Parent and Container Properties

The Parent property is a run time_only property (that is, you don't see it in the Properties window), which returns a reference to the form that hosts the control. The Container property is also a run time_only property, which returns a reference to the container of the control. These two properties are correlated, in that they return the same object—the parent form—when a control is placed directly on the form surface.

While you can't move a control from one form to another using the Parent property (which is read-only), you can move a control to another container by assigning a different value to its Container property (which is a read-write property). Because you're assigning objects and not plain values, you must use the Set keyword:

```
' Move Text1 into the Picture1 container.
Set Text1.Container = Picture1
' Move it back on the form's surface.
Set Text1.Container = Form1
```

The Enabled and Visible Properties

By default, all controls and forms are both visible and enabled at run time. For a number of reasons, however, you might want to hide them or show them in a disabled state. For example, you might use a hidden DriveListBox control simply to enumerate all the drives in the system. In this case, you set the Visible property of the DriveListBox control to False in the Properties window at design time. More frequently, however, you change these properties at run time:

```
' 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
```

Disabled controls don't react to user's actions, but otherwise they're fully functional and can be manipulated through code. Invisible controls are automatically disabled, so you never need to set both these properties to False. All mouse events for disabled or invisible controls are passed to the underlying container or to the form itself.

If an object works as a container for other objects—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.

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. In general, intrinsic lightweight controls don't support this property because they can never receive the input focus. 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. The TabIndex property is also supported by Label and Frame controls, but since these two controls don't support the TabStop property, the resulting effect is that when the user clicks on a Label or a Frame control (or presses the hot key specified in the Label or Frame Caption property), the input focus goes to the control that follows in the Tab order sequence. You can exploit this feature to use Label and Frame controls to provide hot keys to other controls:

```
' 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

These properties affect the shape of the mouse cursor when it hovers over a control. Windows permits a very flexible mouse cursor management in that each form and each control can display a different cursor, and you can also set an application-wide mouse cursor using the Screen global object. Nevertheless, the rules that affect the actual cursor used aren't straightforward:

- If the Screen.MousePointer property is set to a value different from 0-vbDefault, the mouse cursor reflects this value and no other properties are considered. But when the mouse floats over a different application (or the desktop), the cursor appearance depends on that application's current state, not yours.
- If Screen.MousePointer is 0 and the mouse cursor is over a control, Visual Basic checks that control's MousePointer property; if this value is different from 0-vbDefault, the mouse cursor is set to this value.
- If Screen.MousePointer is 0 and the mouse is over a form's surface or it's over a control whose MousePointer property is 0, Visual Basic uses the value stored in the form's MousePointer property.

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
```

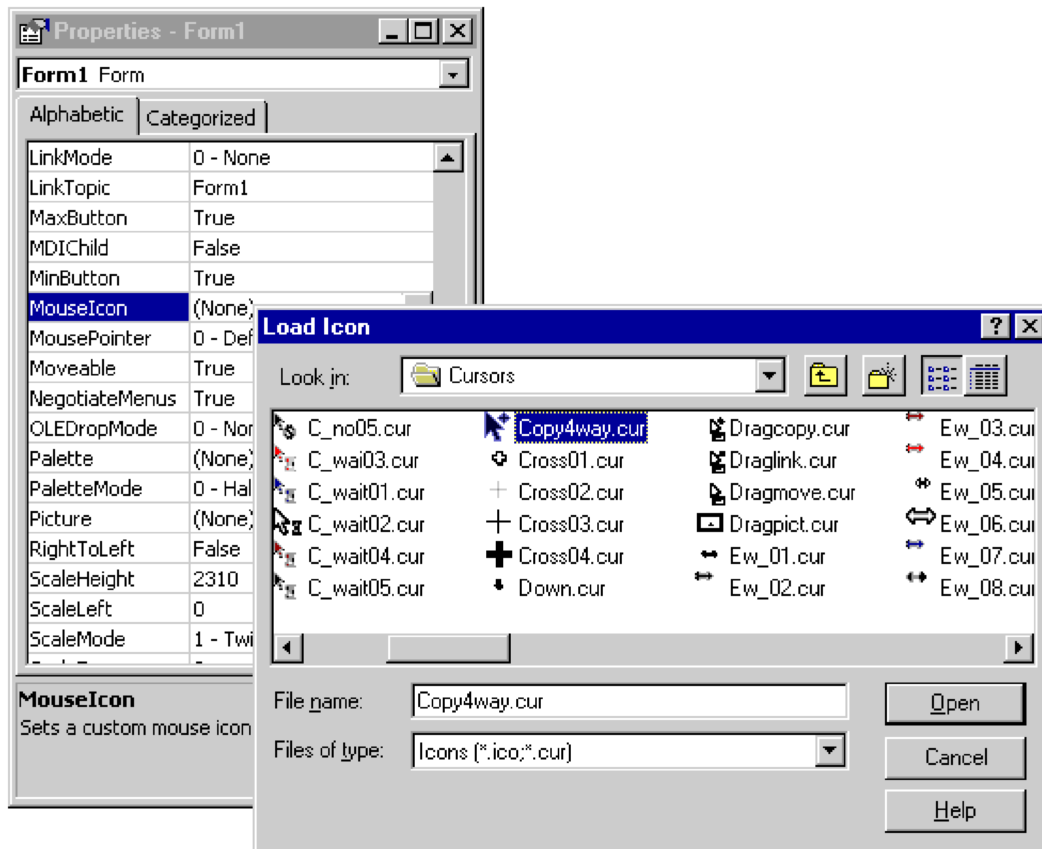
Here's another example:

```
' 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:

```
' Display a red Stop sign mouse cursor. The actual path may differ,
' depending on the main directory where you installed Visual Basic.
MousePointer = vbCustom
MouseIcon = LoadPicture("d:\vb6\graphics\icons\computer\msgbox01.ico")
```

You don't need to load a custom mouse cursor at run time using the LoadPicture command. For example, you can assign it to the MouseIcon property at design time in the Properties window, as you can see in Figure, and activate it only when needed by setting the MousePointer property to 99-vbCustom. If you need to alternate among multiple cursors for the same control but don't want to distribute additional files, you can load additional ICO files in hidden Image controls and switch among them at run time.



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.

Handling Control Event

Visual Basic applications work by executing code written in the Visual Basic programming language. Code is associated with events.

Each Visual Basic object has a set of events. Events are actions which Visual Basic can detect and respond to. For example, a user clicking on a command button on a form will generate a click event for that button.

When an event is generated Visual Basic will run any code you have entered for that event.

Common Events

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

A Click event occurs when the user left-clicks on a control, whereas the DblClick event occurs—you guessed it—when he or she double-clicks on the control using the left mouse button. But don't be fooled by this apparent simplicity because the Click event can occur under different circumstances as well. For example, 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. This behavior is useful because it lets you deal with the two different cases in a uniform way. ListBox and ComboBox controls also fire Click events whenever their ListIndex properties change.

Click and DblClick events don't pass arguments to the program, and therefore you can't count on these events to tell you where the mouse cursor is. To get this information, you must trap the MouseDown event instead, about which I'll say more later in this chapter. 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 because when Visual Basic calls your Click event procedure you don't know whether it will later call the DblClick procedure. At any rate, you should avoid assigning different functions to click and double-click actions on the same control because it tends to confuse the user.

The Change Event

The Change event is the simplest event offered by Visual Basic: Whenever the contents of a control change, Visual Basic fires a Change event. Unfortunately, this simple scheme hasn't been consistently followed in the Visual Basic architecture. As I explained in the previous section, when you click on CheckBox and OptionButton controls, they fire a Click event (rather than a Change event). Fortunately, this inconsistency isn't a serious one.

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 Change event also fires when the contents of the control are changed through code. This behavior often leads to some inefficiencies in the program. For instance, many programmers initialize the Text properties of all TextBox controls in the form's Load event, thus firing many Change events that tend to slow down the loading process.

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. At first glance, these events seem ideal for implementing a sort of validation mechanism—that is, a piece of code that checks the contents of a field and notifies the user if the input value isn't correct as soon as he or she moves the focus to another control. In practice, the sequence of these events is subject to several factors, including the presence of MsgBox and DoEvents statements. Fortunately, Visual Basic 6 has introduced the new Validate event, which elegantly solves the problem of field validation

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 Text1_KeyPress(KeyAscii As Integer)
    MsgBox "User pressed " & Chr$(KeyAscii)
End Sub
```

If you modify the KeyAscii parameter, your changes affect how the program interprets the key. You can also "eat" a key by setting this parameter to 0, as shown in the code below.

```
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.

You should note that KeyPress, KeyDown, and KeyUp events might pose special problems during the debugging phase. In fact, if you place a breakpoint inside a KeyDown event procedure, the target control will never receive a notification that a key has been pressed and the KeyPress and KeyUp events will never fire. Similarly, if you enter break mode when Visual Basic is executing the KeyPress event procedure, the target control will receive the key but the KeyUp event will never fire.

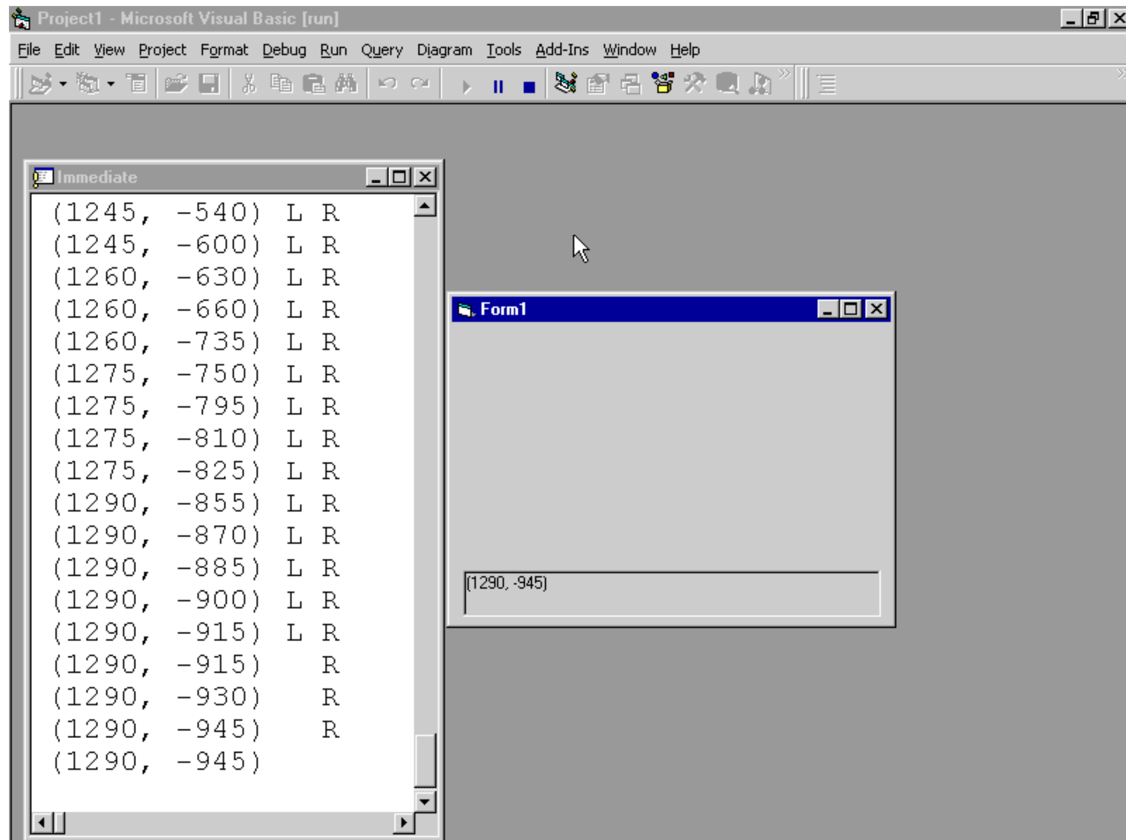
The KeyDown, KeyPress, and KeyUp events are received only by the control that has the input focus when the key is pressed. This circumstance, however, makes it difficult to create form-level key handlers, that is, code routines that monitor keys pressed in any control on the form. For example, suppose 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

These events fire when the mouse is clicked, released, or moved on a control, respectively. All of them receive the same set of parameters: the state of mouse buttons, the state of Shift/Ctrl/Alt keys, and the x- and y-coordinates of the mouse cursor. The coordinates are always relative to the upper left corner of the control or the form. Following Figure is a code sample that displays the status and position of the mouse on a Label control and creates a log in the Immediate window. You can see the results of running this code in Figure



Monitor mouse state using the MouseDown, MouseMove, and MouseUp events. Note the negative y value when the cursor is outside the form's client area.

```
Private Sub Form_MouseDown(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    ShowMouseState Button, Shift, X, Y
End Sub

Private Sub Form_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    ShowMouseState Button, Shift, X, Y
End Sub
Private Sub Form_MouseUp(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    ShowMouseState Button, Shift, X, Y
End Sub

Private Sub ShowMouseState (Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    Dim descr As String
```

```
    descr = Space$(20)
    If Button And vbLeftButton Then Mid$(descr, 1, 1) = "L"
    If Button And vbRightButton Then Mid$(descr, 3, 1) = "R"
    If Button And vbMiddleButton Then Mid$(descr, 2, 1) = "M"
    If Shift And vbShiftMask Then Mid$(descr, 5, 5) = "Shift"
    If Shift And vbCtrlMask Then Mid$(descr, 11, 4) = "Ctrl"
    If Shift And vbAltMask Then Mid$(descr, 16, 3) = "Alt"
    descr = "(" & X & ", " & Y & ") " & descr
    Label1.Caption = descr
    Debug.Print descr
End Sub
```

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:

- The x and y values are relative to the client area of the form or the control, not to its external border; for a form object, the coordinates (0,0) correspond to the pixel in the upper left corner below the title bar or the menu bar (if there is one). When you move the mouse cursor outside the form area, the values of coordinates might become negative or exceed the height and width of the client area.
- 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:

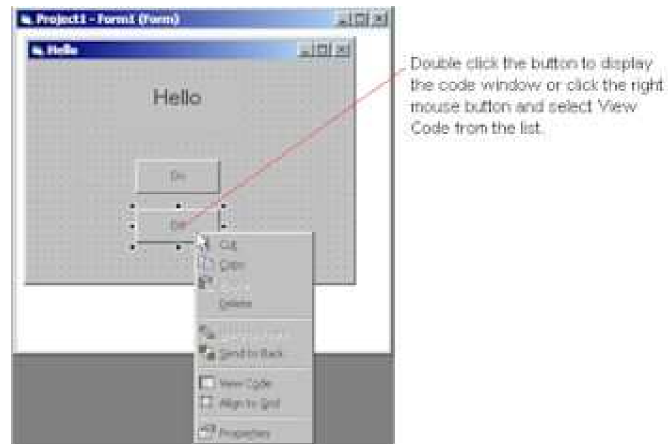
- A Click event occurs after a MouseDown … MouseUp sequence and before the extra MouseMove event.
- When the user double-clicks on a control, the complete event sequence is as follows: MouseDown, MouseUp, Click, MouseMove, DblClick, MouseUp, MouseMove. Note that the second MouseDown event isn't generated.
- 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.

Writing Code

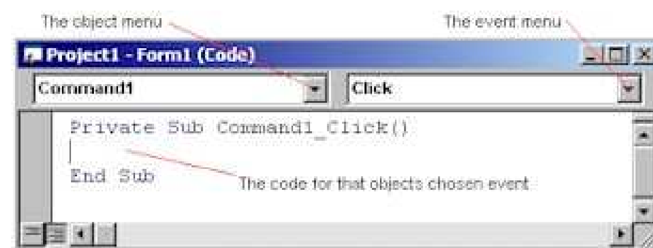
In this exercise you will write code to make your 'Hello' label visible and invisible to the user when the On and Off buttons are clicked.

Each object has an associated set of events which it can respond to. To enter the code, you must access the code window.

Double click on the 'On' button or click the right mouse button and select View Code from the object menu.



The code window will appear as shown below.



Click on the Object menu and select Command1 from the list.

The code window has three main components.

1. The object menu allows you to change between controls. For example, you could use this menu to look at the code for the Off button.
2. The event menu lists all of the available events for the selected control. This list will change depending on the type of control selected.
3. The code for the event is shown in the main part of the window. Your code will appear between the Private Sub and End Sub statements.

Position the cursor in the blank line between the Private Sub and End Sub statements.

Visual Basic 6.0 uses the programming language Visual Basic for Applications, the same language found in many Microsoft Office applications including Microsoft Excel, Microsoft Access and Microsoft Project.

The statement you are going to type assigns a value to the Visible property for the label on your form. This property is set to the value True by default at design time. Your Visual Basic statement will change the value of the property when the command button is clicked.

The syntax to refer to a property in code is:

ObjectName.PropertyName

The syntax to assign a value is

ObjectName.PropertyName = Value

Press the tab key to indent the line of code and type **Label1.Visible = True**, and press the Enter key.

Notice that the True has turned blue. This indicates that True is a special word or symbol that Visual Basic understands. Such words are known as **reserved words.**

Close the code window.

You will now follow the same process to enter the code for the Off button to make the ‘Hello’ label invisible when this button is clicked.

Double click on the Off button to access the code window.

Position the cursor between the Private Sub and End Sub statements.

Press Tab and type **Label1.Visible = False**. Press the Enter key.

Close the code window.

WEEK 6 :

USING LOGICAL EXPRESSION

During this week you will learn :

- If...Then...Else Statement
- Select Case Statement
- OptionButton Control
- CheckBox Control
- Frame Control

If...Then...Else Statement

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

If condition **Then** [statements] [**Else** elsestatements]

Or, you can use the block form syntax:

If condition **Then**
[statements]

[**ElseIf** condition-n **Then**
[elseifstatements] . . .

[**Else**
[elsestatements]]

End If

The **If...Then...Else** statement syntax has these parts:

Part	**Description**
Condition	Required. One or more of the following two types of expressions:
	A numeric expression or string expression that evaluates to **True** or **False**. If condition is Null, condition is treated as **False**.
Statements	Optional in block form; required in single-line form that has no **Else** clause. One or more statements separated by colons; executed if condition is **True**.
condition-n	Optional. Same as condition.
elseifstatements	Optional. One or more statements executed if associated condition-n is **True**.

Elsestatements	Optional. One or more statements executed if no previous condition or condition-n expression is **True**.

Remarks

You can use the single-line form (first syntax) for short, simple tests. However, the block form (second syntax) provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug.

Note With the single-line form, it is possible to have multiple statements executed as the result of an **If...Then** decision. All statements must be on the same line and separated by colons, as in the following statement:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

A block form **If** statement must be the first statement on a line. The **Else**, **ElseIf**, and **End If** parts of the statement can have only a line number or line label preceding them. The block **If** must end with an **End If** statement.

To determine whether or not a statement is a block **If**, examine what follows the **Then** keyword. If anything other than a comment appears after **Then** on the same line, the statement is treated as a single-line **If** statement.

The **Else** and **ElseIf** clauses are both optional. You can have as many **ElseIf** clauses as you want in a block **If**, but none can appear after an **Else** clause. Block **If** statements can be nested; that is, contained within one another.

When executing a block **If** (second syntax), condition is tested. If condition is **True**, the statements following **Then** are executed. If condition is **False**, each **ElseIf** condition (if any) is evaluated in turn. When a **True** condition is found, the statements immediately following the associated **Then** are executed. If none of the **ElseIf** conditions are **True** (or if there are no **ElseIf** clauses), the statements following **Else** are executed. After executing the statements following **Then** or **Else**, execution continues with the statement following **End If**.

Example :

This example shows both the block and single-line forms of the **If...Then...Else** statement. It also illustrates the use of **If TypeOf...Then...Else**.

```
Dim Number, Digits, MyString
Number = 53   ' Initialize variable.
If Number < 10 Then
   Digits = 1
ElseIf Number < 100 Then
' Condition evaluates to True so the next statement is executed.
   Digits = 2
Else
   Digits = 3
End If

' Assign a value using the single-line form of syntax.
If Digits = 1 Then MyString = "One" Else MyString = "More than one"
```

Select Case Statement

Executes one of several groups of statements, depending on the value of an expression.

Syntax

Select Case testexpression
[**Case** expressionlist-n
[statements-n]] . . .
[**Case Else**
[elsestatements]]

End Select

The **Select Case** statement syntax has these parts:

Part	**Description**
Testexpression	Required. Any numeric expression or string expression.
expressionlist-n	Required if a **Case** appears. Delimited list of one or more of the following forms: expression, expression **To** expression, **Is** comparisonoperator expression. The **To** keyword specifies a range of values. If you use the **To** keyword, the smaller value must appear before **To**. Use the **Is** keyword with comparison operators (except **Is** and **Like**) to specify a range of values. If not supplied, the **Is** keyword is automatically inserted.
statements-n	Optional. One or more statements executed if testexpression matches any part of expressionlist-n.
Elsestatements	Optional. One or more statements executed if testexpression doesn't match any of the **Case** clause.

Remarks

If testexpression matches any **Case** expressionlist expression, the statements following that **Case** clause are executed up to the next **Case** clause, or, for the last clause, up to **End Select**. Control then passes to the statement following **End Select**. If testexpression matches an expressionlist expression in more than one **Case** clause, only the statements following the first match are executed.

The **Case Else** clause is used to indicate the elsestatements to be executed if no match is found between the testexpression and an expressionlist in any of the other **Case** selections. Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen testexpression values. If no **Case** expressionlist matches testexpression and there is no **Case Else** statement, execution continues at the statement following **End Select**.

You can use multiple expressions or ranges in each **Case** clause. For example, the following line is valid:

```
Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber
```

Note The **Is** comparison operator is not the same as the **Is** keyword used in the **Select Case** statement.

You also can specify ranges and multiple expressions for character strings. In the following example, **Case** matches strings that are exactly equal to `everything`, strings that fall between `nuts` and `soup` in alphabetic order, and the current value of `TestItem`:

```
Case "everything", "nuts" To "soup", TestItem
```

Select Case statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

Example

This example uses the **Select Case** statement to evaluate the value of a variable. The second **Case** clause contains the value of the variable being evaluated, and therefore only the statement associated with it is executed.

```
Dim Number
Dim strmsg as string
Number = 8   ' Initialize variable.
Select Case Number   ' Evaluate Number.
Case 1 To 5   ' Number between 1 and 5, inclusive.
          strmsg "Between 1 and 5"
' The following is the only Case clause that evaluates to True.
Case 6, 7, 8   ' Number between 6 and 8.
          strmsg "Between 6 and 8"
Case 9 To 10   ' Number is 9 or 10.
       strmsg "Greater than 8"
Case Else   ' Other values.
          strmsg "Not between 1 and 10"
End Select
```

Example

```
Select Case CorrectAnswers%
`Make any answer greater than 11 an A++
       Case 11
              strGrade = "A+"
       Case 10
              strGrade = "A"
       Case 9
              strGrade = "A-"
       Case 8
              strGrade = "B"
       Case 7
              strGrade = "B-"
       Case 6
              strGrade = "C"
       Case 5
              strGrade = "C-"
       Case 4
              strGrade = "D"
       Case 3
              strGrade = "D-"
       Case Else
              strGrade = "F"
```

```
End Select
```

OptionButton Control

An **OptionButton** control displays an option that can be turned on or off.

Syntax

OptionButton

Remarks

Usually, **OptionButton** controls are used in an option group to display options from which the user selects only one. You group **OptionButton** controls by drawing them inside a container such as a **Frame** control, a **PictureBox** control, or a form. To group **OptionButton** controls in a **Frame** or **PictureBox**, draw the **Frame** or **PictureBox** first, and then draw the **OptionButton** controls inside. All **OptionButton** controls within the same container act as a single group.

While **OptionButton** controls and **CheckBox** controls may appear to function similarly, there is an important difference: When a user selects an **OptionButton**, the other **OptionButton** controls in the same group are automatically unavailable. In contrast, any number of **CheckBox** controls can be selected.

Example :

You will now use a set of option buttons on your form to represent three font type choices for the text box.

Double click on the option button control in the tool box.

Move the option button to an appropriate location.

Repeat the steps above to add two more option buttons to your form.

Change the Name and Caption properties of the option buttons as given below.

Name	Caption
OptMSSansSerif	MS Sans Serif
optTimesNewRoman	Times New Roman
OptArial	Arial

You may need to resize the option controls so that the caption fits on one line.

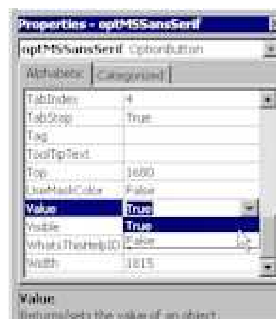
You will set the initial value of the MS Sans Serif option button to True because this is the default font type for the text box.

To do this you set the Value property. The possible values are True or False. True indicates that the option button is selected.

Click on the **MS Sans Serif** option button on your form to select it.

Click on the Value property in the properties window.

Use the pull down menu in the editing panel to change the value to **True** as shown below.



The code for the option buttons sets the FontName property for the text box accordingly.

Double click on the MS Sans Serif option button in your form.

The code window will appear.

Enter the code shown below. Remember that the Private Sub and End Sub statements are automatically included by Visual Basic.

```
Private Sub optMSSansSerif_Click ()
        txtInput.FontName = "MS Sans Serif"
End Sub
```

Repeat this process to enter the code shown below for the Times New Roman button.

```
Private Sub optTimesNewRoman_Click ()
        txtInput.FontName = "Times New Roman"
End Sub
```

Repeat the process again to enter the code shown below for the Arial button.

```
Private Sub optArial_Click ()
        txtInput.FontName = "Arial"
End Sub
```

Run the application to see the result.

Type some text in the text box.

Click on the Times New Roman option button to select it.

The text in the text box will be displayed in Times New Roman.

Try the Arial and MS Sans Serif buttons.

Click on the **Stop** button on the tool bar to return to Visual Basic.

The final task is to change the background colour of the text box when the font changes. Visual Basic provides names for colour values, for example, vbRed for red, vbGreen for green and vbBlue for blue.

You need to add a line of code, which assigns a value to the BackColor property of the text box, to the option button code.

Double click on the MS Sans Serif option button and insert the line shown:

```
Private Sub optMSSansSerif_Click ()
        txtInput.FontName = "MS Sans Serif"
        txtInput.BackColor = vbGreen
End Sub
```

Add code to the Times New Roman button to change the text box colour to Red and code to the Arial button to change the text box colour to Blue.
Try out your application.

To tidy up the form you will add a title to it and change the size.

Click anywhere on the form to select it. Be careful not to click on a control.

Change the Caption property of the form to read Text Formats.

Resize the form to a more appropriate size.

The completed form will appear as shown below.

Run the application to test the changes.

Save the project. Remember to save both the project file and the form.

CheckBox Control

A **CheckBox** control displays an X when selected; the X disappears when the **CheckBox** is cleared. Use this control to give the user a True/False or Yes/No option. You can use **CheckBox** controls in groups to display multiple choices from which the user can select one or more. You can also set the value of a **CheckBox** programmatically with the Value property.

Syntax

CheckBox

Remarks

CheckBox and **OptionButton** controls function similarly but with an important difference: Any number of **CheckBox** controls on a form can be selected at the same time. In contrast, only one **OptionButton** in a group can be selected at any given time.

To display text next to the **CheckBox**, set the **Caption** property. Use the **Value** property to determine the state of the control—selected, cleared, or unavailable.

Using the Check Box Control

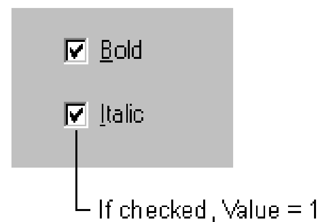
The check box control displays a check mark when it is selected. It is commonly used to present a Yes/No or True/False selection to the user. You can use check box controls in groups to display multiple choices from which the user can select one or more.

The check box control is similar to the option button control in that each is used to indicate a selection that is made by the user. They differ in that only one option button in a group can be selected at a time. With the check box control, however, any number of check boxes may be selected.

For More Information See "Selecting Individual Options with Check Boxes" in "Forms, Controls, and Menus" for a simple demonstration of the check box control.

The Value Property

The Value property of the check box control indicates whether the check box is checked, unchecked, or unavailable (dimmed). When selected, the value is set to 1. For example:



The following table lists the values and equivalent Visual Basic constants that are used to set the Value property.

Setting	Value	Constant
Unchecked	0	vbUnchecked
Checked	1	vbChecked
Unavailable	2	vbGrayed

The user clicks the check box control to indicate a checked or unchecked state. You can then test for the state of the control and program your application to perform some action based on this information.

By default, the check box control is set to vbUnchecked. If you want to preselect several check boxes in a series of check boxes, you can do so by setting the Value property to vbChecked in the Form_Load or Form_Initialize procedures.

You can also set the Value property to vbGrayed to disable the check box. For example, you may want to disable a check box until a certain condition is met.

The Click Event

Whenever the user clicks on the check box control, the Click event is triggered. You can then program your application to perform some action depending upon the state of the check box. In the following example, the check box control's Caption property changes each time the control is clicked, indicating a checked or unchecked state.

```
Private Sub Check1_Click()
   If Check1.Value = vbChecked Then
```

```
      Check1.Caption = "Checked"
   ElseIf Check1.Value = vbUnchecked Then
      Check1.Caption = "Unchecked"
   End If
End Sub
```

Note If the user attempts to double-click the check box control, each click will be processed separately; that is, the check box control does not support the double-click event.

Responding to the Mouse and Keyboard

The Click event of the check box control is also triggered when the focus is shifted to the control with the keyboard by using the TAB key and then by pressing the SPACEBAR.

You can toggle selection of the check box control by adding an ampersand character before a letter in the Caption property to create a keyboard shortcut. For example:

In this example, pressing the ALT+C key combination toggles between the checked and unchecked states.

Example :

The next step is to add check boxes to the form. These will allow the user to change the format of the text in the text box to bold or italics.

Check boxes are used to represent on/off values. They can either be checked or unchecked. When they are checked a tick will appear in the box.

Double click on the check box control in the tool box.

Move the check box to an appropriate location.

Repeat the last two steps to add a second check box below the first.

Your form will appear like this.

As with the text box you should now change the names of the check boxes to something more meaningful.

Click on the **Check1** check box in your form.

Change the Name property to **chkBold**.

You will also need to change the caption of the check box. Remember that this is what appears on the form.

Change the Caption property to **Bold**.

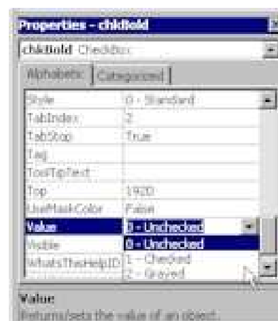
Repeat the steps above to change the name of the second check box to **chkItalic** and the caption to **Italic**.

The Value property of a check box determines whether the check box is checked, unchecked or greyed.

Click on the **Bold** check box on your form to select it.

Click on the Value property in the properties window.

Use the pull down menu to see the possible values for this property.



For now, leave the value property as **0 – Unchecked**. You are going to write code so that if this property has the value one (i.e. the user has checked the box), the font will be bold, otherwise it will not be bold.

At this stage the check boxes on the form will not perform any action. To make the appearance of the text in the text box change when the check boxes are checked and unchecked you need to write code.

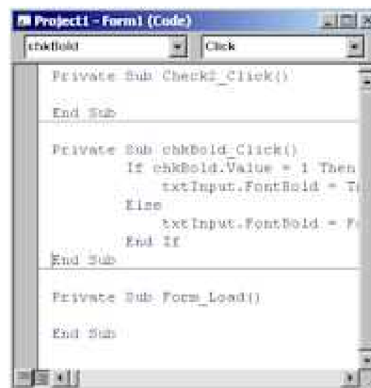
Double click on the **Bold** check box on your form.

The code window will appear.

Enter the code shown below. Remember that the Private Sub and End Sub statements are automatically included by Visual Basic

```
Private Sub chkBold_Click ()
        If chkBold.Value = 1 Then
                txtInput.FontBold = True
        Else
                txtInput.FontBold = False
        End If
End Sub
```

This is shown below.



This is an example of the Visual Basic If statement. The If statement in code allows a choice of actions. The general structure is:

```
If expression Then
        statements1
Else
        statements2
End If
```

The first line of the If statement in your example tests the current value of the check box. If the value of the property Value is equal to 1, then the check box is checked.

If the bold checkbox is checked, set the FontBold property of the txtInput control to True.

Otherwise, if the check box value is not 1 (unchecked or inactive), set the FontBold property of the txtInput control to False.

Run the application to see the result.

Click on the **play** button on the tool bar.

Type some text in the text box.

Click on the **Bold** check box to check it.

The text in the text box should be displayed as bold.

Click on the **Bold** check box to uncheck it.

The text in the text box should be displayed as normal.

Click on the stop button on the tool bar to return to Visual Basic.

The code for the Italic check box is similar to the Bold check box but changes the FontItalic property.

Double click on the **Italic** check box.

The code window will appear.

Enter the code shown below. Remember that the Private Sub and End Sub statements are automatically included by Visual Basic.

```
Private Sub chkItalic_Click ()
        If chkItalic.Value = 1 Then
                txtInput.FontItalic = True
        Else
                txtInput.FontItalic = False
        End If
End Sub
```

Run the application to see the result.

Click on the **play** button on the tool bar.

Type some text in the text box.

Click on the **Italic** check box to check it.

The text in the text box should be displayed as italic.

Click on the **stop** button on the tool bar to return to Visual Basic.

Frame Control

A **Frame** control provides an identifiable grouping for controls. You can also use a **Frame** to subdivide a form functionally—for example, to separate groups of **OptionButton** controls.

Syntax

Frame

Remarks

To group controls, first draw the **Frame** control, and then draw the controls inside the **Frame**. This enables you to move the **Frame** and the controls it contains together. If you draw a control outside the **Frame** and then try to move it inside, the control will be on top of the **Frame** and you'll have to move the **Frame** and controls separately.

To select multiple controls in a **Frame**, hold down the CTRL key while using the mouse to draw a box around the controls.

Using the Frame Control

Frame controls are used to provide an identifiable grouping for other controls. For example, you can use frame controls to subdivide a form functionally — to separate groups of option button controls.

In most cases, you will use the frame control passively — to group other controls — and will have no need to respond to its events. You will, however, most likely change its Name, Caption, or Font properties.

For More Information See "Grouping Options with Option Buttons" in "Forms, Controls, and Menus" for a simple demonstration of using the frame control to group option buttons.

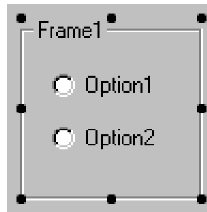
Adding a Frame Control to a Form

When using the frame control to group other controls, first draw the frame control, and then draw the controls inside of it. This enables you to move the frame and the controls it contains together.

Drawing Controls Inside the Frame

To add other controls to the frame, draw them inside the frame. If you draw a control outside the frame, or use the double-click method to add a control to a form, and then try to move it

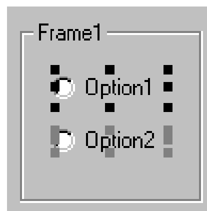
inside the frame control, the control will be on top of the frame and you'll have to move the frame and controls separately.



Note If you have existing controls that you want to group in a frame, you can select all the controls, cut them to the clipboard, select the frame control, and then paste them into the frame control.

Selecting Multiple Controls in a Frame

To select multiple controls in a frame , hold down the CTRL key while using the mouse to draw a box around the controls. When you release the mouse, the controls inside the frame will be selected, as in Figure .



WEEK 7:

LOOPING

During this week you will learn :

- Looping
- Do...Loop Statement
- For...Next Statement

Visual Basic allows a procedure to be repeated as many times as long as the processor could support. This is generally called looping .

Do...Loop Statement

Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

Syntax

Do [{**While** | **Until**} condition]
[statements]
[**Exit Do**]
[statements]

Loop

Or, you can use this syntax:

Do
[statements]
[**Exit Do**]
[statements]

Loop [{**While** | **Until**} condition]

The **Do Loop** statement syntax has these parts:

Part	**Description**
Condition	Optional. Numeric expression or string expression that is **True** or **False**. If condition is Null, condition is treated as **False**.
Statements	One or more statements that are repeated while, or until, condition is **True**.

Remarks

Any number of **Exit Do** statements may be placed anywhere in the **Do…Loop** as an alternate way to exit a **Do…Loop**. **Exit Do** is often used after evaluating some condition, for example, **If…Then**, in which case the **Exit Do** statement transfers control to the statement immediately following the **Loop**.

When used within nested **Do…Loop** statements, **Exit Do** transfers control to the loop that is one nested level above the loop where **Exit Do** occurs.

Example

This example shows how **Do...Loop** statements can be used. The inner **Do...Loop** statement loops 10 times, sets the value of the flag to **False**, and exits prematurely using the **Exit Do** statement. The outer loop exits immediately upon checking the value of the flag.

```
Dim Check, Counter
Check = True: Counter = 0   ' Initialize variables.
Do   ' Outer loop.
   Do While Counter < 20   ' Inner loop.
      Counter = Counter + 1   ' Increment Counter.
      If Counter = 10 Then   ' If condition is True.
         Check = False   ' Set value of flag to False.
         Exit Do   ' Exit inner loop.
      End If
   Loop
Loop Until Check = False   ' Exit outer loop immediately.
```

For...Next Statement

Repeats a group of statements a specified number of times.

Syntax

For counter = start **To** end [**Step** step]
[statements]
[**Exit For**]
[statements]

Next [counter]

The **For…Next** statement syntax has these parts:

Part	**Description**
Counter	Required. Numeric variable used as a loop counter. The variable can't be a Boolean or an array element.
Start	Required. Initial value of counter.

End	Required. Final value of counter.
Step	Optional. Amount counter is changed each time through the loop. If not specified, step defaults to one.
Statements	Optional. One or more statements between **For** and **Next** that are executed the specified number of times.

Remarks

The step argument can be either positive or negative. The value of the step argument determines loop processing as follows:

Value	**Loop executes if**
Positive or 0	counter <= end
Negative	counter >= end

After all statements in the loop have executed, step is added to counter. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute initially), or the loop is exited and execution continues with the statement following the **Next** statement.

Tip Changing the value of counter while inside a loop can make it more difficult to read and debug your code.

Any number of **Exit For** statements may be placed anywhere in the loop as an alternate way to exit. **Exit For** is often used after evaluating of some condition, for example **If...Then**, and transfers control to the statement immediately following **Next**.

You can nest **For...Next** loops by placing one **For...Next** loop within another. Give each loop a unique variable name as its counter. The following construction is correct:

```
For I = 1 To 10
   For J = 1 To 10
      For K = 1 To 10
          ...
      Next K
   Next J
Next I
```

Note If you omit counter in a **Next** statement, execution continues as if counter is included. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

Example

This example uses the **For...Next** statement to create a string that contains 10 instances of the numbers 0 through 9, each string separated from the other by a single space. The outer loop uses a loop counter variable that is decremented each time through the loop.

```
Dim Words, Chars, MyString
For Words = 10 To 1 Step -1     ' Set up 10 repetitions.
   For Chars = 0 To 9   ' Set up 10 repetitions.
      MyString = MyString & Chars   ' Append number to string.
   Next Chars   ' Increment counter
   MyString = MyString & " "   ' Append a space.
Next Words
```

Example

```
For  counter=1 to 10
      display.Text=counter
Next
```

Example

```
For counter=1 to 1000 step 10
     counter=counter+1
Next
```

Example

```
For counter=1000 to 5 step -5
     counter=counter-10
Next
```

WEEK 8:

PROCEDURE AND FUNCTIONS

During this week you will learn :

- Functions and procedures
- Scope of procedures
- Supplied Numeric Functions (Math Functions)
- Supplied String Function

Concept of procedure and function

A procedure is a group of sequential statements that have a name in common and can be executed by calling the group (by name, of course) from some other place in the program. VB. lets you use two distinct types of procedures: Sub procedures and Function procedures. The difference between the two is that a Function procedure returns a calculated value, while a Sub procedure doesn't return a value.
Sub procedures are used extensively to handle events such as the Load event for a page or the Click event for a button. In addition, you can create your own Sub or Function procedures. This often helps you simplify your code by enabling you to break a long Sub procedure into several shorter Sub or Function procedures.
Note that Sub procedures are often called subroutines, and both Sub and Function procedures are often called methods. The only difference between a Sub procedure and a Function procedure is that a Function procedure returns a value, while a Sub procedure does not.

Using Sub Procedures

A Sub procedure begins with a Sub statement and ends with an End Sub statement. The statements that make up the procedure go between the Sub and End Sub statements. The Sub command supplies the name of the procedure and any parameters that can be passed to the subroutine.

Sub Statement

Declares the name, arguments, and code that form the body of a **Sub** procedure.

Syntax

Sub name [(arglist)]
    [statements]
    [Exit Sub]
    [statements]
End Sub

The **Sub** statement syntax has these parts:

Part	Description
Name	Required. Name of the **Sub**; follows standard variable naming conventions.
Arglist	Optional. List of variables representing arguments that are passed to the **Sub** procedure when it is called. Multiple variables are separated by commas.
statements	Optional. Any group of statements to be executed within the **Sub** procedure.

Remarks

The **Exit Sub** keywords cause an immediate exit from a **Sub** procedure. Program execution continues with the statement following the statement that called the **Sub** procedure. Any number of **Exit Sub** statements can appear anywhere in a **Sub** procedure.

Example:

This example uses the **Sub** statement to define the name, arguments, and code that form the body of a **Sub** procedure.

```
' Sub procedure definition.
' Sub procedure with two arguments.
Sub SubComputeArea(Length, TheWidth)
   Dim Area As Double    ' Declare local variable.
   If Length = 0 Or TheWidth = 0 Then
   ' If either argument = 0.
      Exit Sub    ' Exit Sub immediately.
   End If
   Area = Length * TheWidth    ' Calculate area of rectangle.
   Debug.Print Area    ' Print Area to Debug window.
End Sub

For calling SubComputeArea use :
        SubComputeArea(10,20)
```

Here's another example:

Sub SayHello
Response.Write("Hello, World!")
End Sub
Sub SayWhatever(Message As String)
Response.Write(Message)
End Sub

In this example, the SayHello procedure writes the text "Hello, World!" to the page. The SayWhatever message writes the text you pass via a parameter to the page.

Notice that if the procedure uses parameters, you must provide both the name and the type of the parameter (in parentheses) following the procedure

name. For example, in the second procedure above, one parameter is used. The name of the parameter is Message, and its type is String.
You can invoke a Sub procedure simply by listing the procedure's name, almost as if the procedure had become its own VB statement. Here's a simple example:

```
SayHello
```

If the Sub procedure uses parameters, you list the values you want to pass to the parameters following the procedure name, like this:

```
SayWhatever("Greetings, Planet!")
```

Besides literal values, you can also pass variables or complex expressions. For example, both of the following calls are allowed:

```
Dim Msg As String
Dim Part1 As String
Dim Part2 As String
Part1 = "Hello"
Part2 = "World!"
Msg = Part1 & ", " & Part2
SayWhatever(Msg)
SayWhatever(Part1 & ", " & Part2)
```

Here the Sub procedure SayWhatever is called twice. In both cases, the same value is passed to the Message parameter: The first time, the value is

passed via a variable named Msg; the second time, the value is passed as an expression.
If you want to be just a bit of a neatness freak, you can type the keyword Call before the subroutine name when calling the subroutine — as in this example:

```
Call SayWhatever("Hello World!")
```

Okay, the Call keyword is optional, but some VB programmers like to use it to help distinguish user-written subroutines from built-in VB commands.

Scope of a procedure

Sub procedures (as well as Function procedures, described in the next section) can begin with an access modifier that specifies whether the procedure is available to other classes in the application. The three most common access modifiers are

✦ Public: The procedure is visible throughout the application.

✦ Private: The procedure is visible only within the current class, which means it can't be used from other classes.

✦ Protected: The procedure is hidden from other classes in the project, with the exception of any classes derived from the current class

Working with Functions

A Function procedure is similar to a Sub procedure, with one crucial difference:
A Function procedure returns a value.

Declares the name, arguments, and code that form the body of a **Function** procedure.

Syntax

Function name [(arglist)] [**As** type] [statements]
    [name = expression]
    [**Exit Function**]
    [statements]
    [name = expression]
End Function

The **Function** statement syntax has these parts:

Part	**Description**
Name	Required. Name of the **Function**; follows standard variable naming conventions.
Arglist	Optional. List of variables representing arguments that are passed to the **Function** procedure when it is called. Multiple variables are separated by commas.
Type	Optional. Data type of the value returned by the **Function** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, or (except fixed length).
Statements	Optional. Any group of statements to be executed within the **Function** procedure.
Expression	Optional. Return value of the **Function**.

Remarks

To return a value from a function, assign the value to the function name. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to name, the procedure returns a default value: a numeric function returns 0, a string function returns a zero-length string (""), and a **Variant** function returns Empty.

Example:

This example uses the **Function** statement to declare the name, arguments, and code that form the body of a **Function** procedure. The last example uses hard-typed, initialized **Optional** arguments.

```
' The following user-defined function returns the square root of the
' argument passed to it.
Function CalculateSquareRoot(NumberArg As Double) As Double
```

```
   If NumberArg < 0 Then   ' Evaluate argument.
      Exit Function   ' Exit to calling procedure.
   Else
      CalculateSquareRoot = Sqr(NumberArg)   ' Return square root.
   End If
End Function

For calling CalculateSquareRoot use :
        Square_root = CalculateSquareRoot (25)
```

Built in Library functions

This are functions that are provided by the VB compiler and can be called upon at any time to perform specific operation in our program.

Supplied Numeric Functions (Math Functions)

Abs Function

Returns **a value of the same type that is passed to it** specifying the absolute value of a number.

Syntax

Abs(number**)**

The required number argument can be any valid numeric expression. If number contains Null, **Null** is returned; if it is an uninitialized variable, zero is returned.

Remarks

The absolute value of a number is its unsigned magnitude. For example, `ABS(-1)` and `ABS(1)` both return `1`.

Example :

This example uses the **Abs** function to compute the absolute value of a number.

```
Dim MyNumber
MyNumber = Abs(50.3)   ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Int Functions

Returns the integer portion of a number.

Syntax

Int(number)

The required number argument is a Double or any valid numeric expression. If number contains Null, **Null** is returned.

Example :

This example illustrates how the **Int** function return integer portions of numbers.

```
Dim MyNumber

MyNumber = Int(99.8)    ' Returns 99.

MyNumber = Int(-99.8)    ' Returns -100.

MyNumber = Int(-99.2)    ' Returns -100.
```

Rnd Function

Returns a **Single** containing a random number.

Syntax

Rnd[(number)]

The optional number argument is a Single or any valid numeric expression.

Return Values

If number is	Rnd generates
Less than zero	The same number every time, using number as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

Remarks

The **Rnd** function returns a value less than 1 but greater than or equal to zero.
The value of number determines how **Rnd** generates a random number:
For any given initial seed, the same number sequence is generated because each successive call to the **Rnd** function uses the previous number as a seed for the next number in the sequence.
Before calling **Rnd,** use the **Randomize** statement without an argument to initialize the random-number generator with a seed based on the system timer.

To produce random integers in a given range, use this formula:
```
Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
```

Example :

This example uses the **Rnd** function to generate a random integer value from 1 to 6.

```
Dim MyValue
MyValue = Int((6 * Rnd) + 1)    ' Generate random value between 1 and 6.
```

Sqr Function

Returns a **Double** specifying the square root of a number.

Syntax

Sqr(number)
The required number argument is a Double or any valid numeric expression greater than or equal to zero.

Example :

This example uses the **Sqr** function to calculate the square root of a number.

```
Dim MySqr
MySqr = Sqr(4)    ' Returns 2.
MySqr = Sqr(23)    ' Returns 4.79583152331272.
MySqr = Sqr(0)    ' Returns 0.
MySqr = Sqr(-4)    ' Generates a run-time error.
```

Supplied String Function

Len Function

Returns a Long containing the number of characters in a string or the number of bytes required to store a variable.

Syntax

Len(string)
The **Len** function syntax has these parts:

Part	**Description**
String	Any valid string expression. If string contains Null, Null is returned.

Example :

```
Dim MyString, MyLen
MyString = "Hello World"    ' Initialize variable.
MyLen = Len(MyString)    ' Returns 11.
```

LCase and Functions

Returns a String that has been converted to lowercase.

Syntax

LCase(string)
The required string argument is any valid string expression. If string contains Null, Null is returned.

Remarks

Only uppercase letters are converted to lowercase; all lowercase letters and nonletter characters remain unchanged.

Example :

This example uses the **LCase** function to return a lowercase version of a string.

```
Dim UpperCase, LowerCase
Uppercase = "Hello World 1234"    ' String to convert.
Lowercase = Lcase(UpperCase)    ' Returns "hello world 1234".
```

UCase Function

Returns a **Variant** (**String**) containing the specified string, converted to uppercase.

Syntax

UCase(string)
The required string argument is any valid string expression. If string contains Null, **Null** is returned.

Remarks

Only lowercase letters are converted to uppercase; all uppercase letters and nonletter characters remain unchanged

Example :

This example uses the **UCase** function to return an uppercase version of a string.

```
Dim LowerCase, UpperCase
```

```
LowerCase = "Hello World 1234"    ' String to convert.
UpperCase = UCase(LowerCase)   ' Returns "HELLO WORLD 1234".
```

Asc Function

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

Syntax

Asc(string)
The required string argument is any valid string expression. If the string contains no characters, a run-time error occurs.

Example :

This example uses the **Asc** function to return a character code corresponding to the first letter in the string.

```
Dim MyNumber
MyNumber = Asc("A")    ' Returns 65.
MyNumber = Asc("a")    ' Returns 97.
MyNumber = Asc("Apple")    ' Returns 65.
```

Chr Function

Returns a String containing the character associated with the specified character code.

Syntax

Chr(charcode)
The required charcode argument is a Long that identifies a character.

Example :

This example uses the **Chr** function to return the character associated with the specified character code.

```
Dim MyChar
MyChar = Chr(65)    ' Returns A.
MyChar = Chr(97)    ' Returns a.
MyChar = Chr(62)    ' Returns >.
MyChar = Chr(37)    ' Returns %.
```

Supplied Time And Date Functions

Now Function

Returns a **Variant** (**Date**) specifying the current date and time according your computer's system date and time.

Syntax

Now

Example :

This example uses the **Now** function to return the current system date and time.

```
Dim Today
Today = Now   ' Assign current system date and time.
```

Date Function

Returns a **Variant** (**Date**) containing the current system date.

Syntax

Date

Remarks

To set the system date, use the **Date** statement.

Example :

This example uses the **Date** function to return the current system date.

```
Dim MyDate
MyDate = Date   ' MyDate contains the current system date.
```

Time Function

Returns a **Variant** (**Date**) indicating the current system time.

Syntax

Time

Remarks

To set the system time, use the **Time** statement.
Example :

This example uses the **Time** function to return the current system time.

```
Dim MyTime
MyTime = Time   ' Return current system time.
```

WEEK 9:

CONCEPT OF ARRAY

During this week you will learn :

- Non-Arrays data values
- Arrays data values
- Declaring Fixed-Size Arrays
- Multidimensional Arrays

Non-Arrays data values

The following list of four variables doesn't count as an array:

```
curSales sngbonus98 strFirstName intCtr
```

This list doesn't define an array because each variable has a different name. You may wonder how more than one variable can have the same name; this convention seems to violate the rules of variables. If two variables have the same name, how does Visual Basic know which one you want when you use its name?

Arrays data values

An array is a list of more than one variable with the same name. Not every list of variables is an array.
Arrays allow you to refer to a series of variables by the same name and to use a number (an index) to tell them apart. This helps you create smaller and simpler code in many situations, because you can set up loops that deal efficiently with any number of cases by using the index number. Arrays have both upper and lower bounds, and the elements of the array are contiguous within those bounds. Because Visual Basic allocates space for each index number, avoid declaring an array larger than necessary.

All the elements in an array have the same data type. Of course, when the data type is Variant, the individual elements may contain different kinds of data (objects, strings, numbers, and so on). You can declare an array of any of the fundamental data types.

Example :

Suppose that you want to process 35 people's names and monthly dues from your local neighborhood association. The dues are different for each person. All this data fits nicely in a table of data, but suppose that you also want to hold, at one time, all the data in variables so

that you can perform calculations and print various statistics about the members by using Visual Basic.

Without arrays, you find yourself having to store each of the 35 names in 35 different variables, and each of their dues in 35 different variables. But doing so makes for a complex and lengthy program. To enter the data, you have to store the data in variables with names such as the following:

```
strFamilyName1    curFamilyDues1

strFamilyName2    curFamilyDues2

strFamilyName3    curFamilyDues3

strFamilyName4    curFamilyDues4
```

The list continues until you use different variable names for all the 35 names and dues. With array we need only one array to hold 35 names.

Declaring Fixed-Size Arrays

For declaring arrays, the format of the Public and Dim statements varies only in the keyword of the command and its placement in the module. Here are the syntaxes of the two statements:

```
Dim arName(intSub) [As dataType][, arName(intSub) [As dataType]]...
```

You name arrays (arName) just as you do regular variables. You can create an array of any data type, so dataType can be Integer, Single, or any of the data types with which you're familiar. The intSub portion of the commands describes the number of elements and how you refer to those array elements.

Using Option Base

Declaring an array is easiest when you specify only the upper subscript bound. All array subscripts begin at 0 unless the following statement appears in the module's Declarations section:

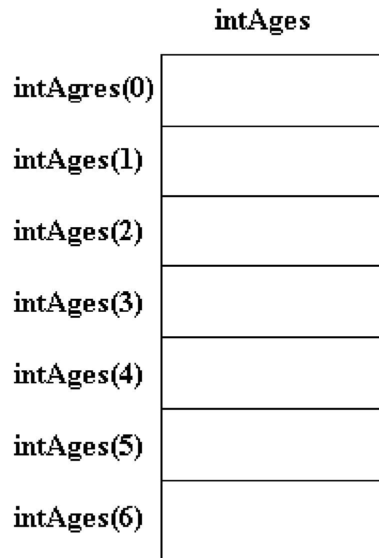
```
Option Base 1
```

The Option Base command is rather outdated. (If you want to change the lower bounds of an array, you should consider using the more advanced Low To option.)

The following Dim statement declares seven elements of an Integer array named intAges:

```
Dim intAges(6) ' Reserves 7 elements
```

The subscript, 6, is the upper subscript, and the lower subscript is 0 (without an Option Base 1 appearing elsewhere, which would force the beginning subscript to 1). Figure next illustrates just what's declared with this statement. An array of seven Integer values, all with the same name (intAges), is reserved for use. Each variable is distinguished by its subscript number; intAges(2) is a completely different variable from intAges(6).



The intAges array contains seven elements.

Example

Based on the previous discussion, you can declare the strFamilyName and curFamilyDues arrays as follows:

```
Dim strFamilyName(35) As String ' Reserves 36 names
Dim curFamilyDues(35) As Currency ' Reserves 36 dues
```

Actually, the subscript 35 is the upper bound, and the subscript 0 is automatically the lower bound. Therefore, these statements each dimension 36 elements in each array. The previous discussion mentioned 35 members in the neighborhood association, so the 0 subscript isn't used.

Because Dim was used here, the arrays have procedure-level scope. Only the code within the procedure that contains these two statements can use the two arrays unless the procedure passes the arrays to other procedures.

Sometimes, specifying the lower and upper bounds of the array subscripts makes sense. As you've seen, if you specify Option Base 1, the lower array subscript is 1. If you specify Option Base 0 or nothing at all, the lower array subscript bounds are zero. By using the expanded array declaration statements with the To keyword, however, you can specify the upper and lower bounds of your array subscripts.

Next

These Dim statements each do the same thing:

```
Dim Amounts(0 To 50) ' Subscripts 0 to 50
Dim Amounts(50) ' Subscripts 0 to 50
```

And so do these pairs:

```
Option Base 1
Dim Balances(75) ' Subscripts 1 to 75
Option Base 0
Dim Balances(1 To 75) ' Subscripts 1 to 75
```

You can see how the Option Base statement affects the arrays you declare. Now that you can declare arrays, you'll now learn how to use them.

Multidimensional Arrays

A multidimensional array is an array with more than one subscript. A single-dimensional array is a list of values, whereas a multidimensional array simulates a table of values or even multiple tables of values. The most commonly used table is a two-dimensional table (an array with two subscripts).

Suppose that a softball team wants to keep track of its players' hits. The team played eight games, and 10 players are on the team.

Table : A Softball Team's Hit Record

Player	Game1	Game2	Game3	Game4	Game5	Game6	Game7	Game8
Adeniyi	2	1	0	0	2	3	3	1
Berryhill	1	0	3	2	5	1	2	2
Edwards	0	3	6	4	6	4	5	3
Grady	1	3	2	0	1	5	2	1
Howard	3	1	1	1	2	0	1	0
Powers	2	2	3	1	0	2	1	3
Smith	1	1	2	1	3	4	1	0
Townsend	0	0	0	0	0	0	1	0
Ulmer	2	2	1	1	2	1	1	2
Williams	2	3	1	0	1	2	1	1

Do you see that the softball table is a two-dimensional table? It has rows (the first dimension) and columns (the second dimension). Therefore, you call it a two-dimensional table with 10 rows and eight columns. (Generally, the number of rows is specified first.) Each row has a player's name, and each column has a game number associated with it, but these headings aren't part of the data. The data consists of only 80 values (10 rows times eight columns). The data in a table, like the data in an array, always is the same type of data (in this case, every value is an integer). If the table contains names, it's a string table, and so on.

The number of dimensions-in this case, two-corresponds to the dimensions in the physical world. The first dimension represents a line. The single-dimensional array is a line, or list, of values. Two dimensions represent both length and width. You write on a piece of paper in two dimensions; two dimensions represent a flat surface. Three dimensions represent width, length, and depth. You may have seen three-dimensional movies; not only do the images have width and height, but they also (appear to) have depth.

It's difficult to visualize more than three dimensions. You can, however, think of each dimension after three as another occurrence. In other words, you can store a list of one player's season hit record in an array. The team's hit record (as shown in Table 12.2) is two-dimensional. Their league, made up of several teams' hit records, represents a three-dimensional table. Each team (the depth of the table) has rows and columns of hit data. If there's more than one league, you can consider leagues another dimension.

Visual Basic lets you work with up to 60 dimensions, although real-world data rarely requires more than two or three dimensions.

Example

The following statement declares a two-dimensional 10-by-10 array within a procedure:

```
Static MatrixA(9, 9) As Double
```

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

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

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

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

This declaration creates an array that has three dimensions with sizes 4 by 10 by 15. The total number of elements is the product of these three dimensions, or 600.

WEEK 10:

CONCEPT OF CLASS/LIST BOX CONTROL

During this week you will learn :

- Classes and instances of classes
- creation of methods
- List boxes
- List Boxes Controls That Work Like Arrays
- The list box control's events.
- List box method

The Concept Of Objects

Object oriented Programming has been a technical word for quite some time. Object oriented programming is a Programming structure that encapsulates data and functionality as a single unit and for which the only public access is through the programming structure's interface (Properties, Methods and Events). There are different types of object, all objects share specific characteristics, such as properties and methods.

The most commonly used object. In visual Basic are the form object, and the control objects (Already Discussed). another less technical example is the use of pets, Dog and cat are different entities (object) but of the same category of Pets Object. Similarly text boxes and command buttons are each a unique type of object, but both considered a control object every object belong to a class as cat and dog belong to the class of pet.

Concept Of Classes

Classes enable you to develop application using object oriented programming (OOP) techniques. Classes are templates that define objects. When you create a new form in a visual Basic Project, you are actually creating a class that defines a form Instantiated runtime are derived from the class. to truly realize the benefit of OOP, you must create your own class.

Creating an Object Interface

To create an Object from a class, the class must expose an Interface. Interface is a mean by which client code communicates with the object derived from the class.

The Interface of a class consists of one or more of the following: members. Proprieties, methods and events. Clients interact with an object via the object's interface. You can create class properties using property procedures.

Property procedure enable you to execute code when a properly is changed, to validate property values, and to dictate whether a properly is read-only, write –only or both readable and writable. Declaring a properly procedure is similar to declaring a standard functions sub procedure but with some differences. The structure of a property procedure looks like this.

Public property Propertyname ( ) as long
Get
‘Code to return the property’s value goes here
End Get
Set (Byval Value As long)
‘Code that accept a new value goes here
End set
End property

The first word in the property declaration simply designate the scope of the property (Public or Private)

Properties declare with public are available to code outside of the class i.e they can be accessed by the client code.

Properties declared as private are available only to code within the class.

The word immediately following the public or private word property, tells VB that you are creating a property Procedure rather than a sub or function procedure. Next come the property name and data type. Entry the declaration statement of property procedure causes VB to complete the procedure template for you for instance, Type the following two statements into your class.

Private M- IngHeight As Long
Public property Height ( ) As Long

Press enter Key, and VB fill in the rest of the procedure temperature for you.

Construction and Destructor

The Get construct is used to place code that returns a value for the property when ready by the client, if you remove the Get and End Get Statement; Client won't be able to read the value of the properly.

The set construct is where you place code that accepts a new properly value from client code. If you remove the set and End set statement, client won't be able to change value of properly. Leaving the Get construct and removing the set construct create a read-only property. i.e Client can retrieve the value of the property but they cannot change it.

You can create new object when dimensioning a variable as follows:

Dim Objmyobject As new Clsmyclass( )

When an object is no longer needed, it should be destroyed so that all the resources used by the object can be reclaimed. Objects are destroyed automatically when the last reference to the object is released.

To explicitly release an object, set the object variable equal to nothing like this

Objmyobject = Nothing

Creating instance of an object from a class, after obtaining a reference to an object and assign it to a variable you can manipulate the object using an object variable. Consider the following codes.

Dim Objmyobject As Object
Objmyobject = New Clsmyclass ( )
Msgbox (objmyobject. AddTwo Numbers (1,2)

The first statement creates a variable of type object.
In the second statement, the variable appears on the left hand side of the equal sign is a variable being set to same value, you want to place a reference to an object in the variable, but no object has yet been created, the New Keyword tell VB to create a new object and the text following New is the name of the class used to derive the object.

The last statement calls the AddTwo numbers method of your class and displays the result in a message box.

Try Run the Program, and save it.

Note; VB.NET handles Classes better.

List boxes

What is a list box?

A list box is a list, which appears on the screen, from which the user can choose items. In this chapter you will create the text box and list box shown below and program the add, delete, clear buttons.

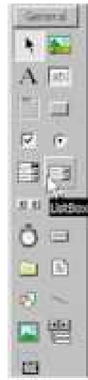
The add button will add the contents of the text box to the list. The delete button will remove the item in the list which the user has selected. The clear button will delete all the items from the list.

Creating a list box

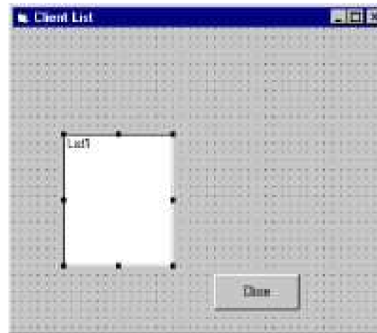
Creating a list box is just like creating any other Visual Basic control.

Select View Form for frmClientList from the Project window.

Double click on the list box control in the tool box.

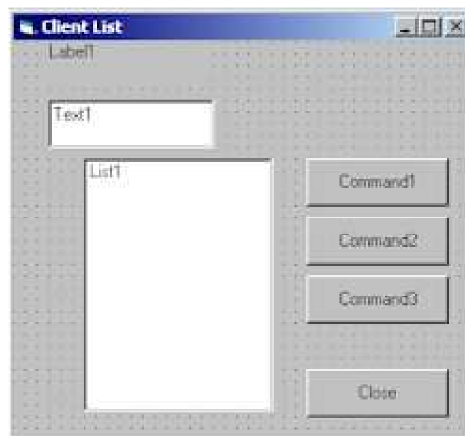


Move and size the list box.



Adding the other controls on the form

Add a label, text box and three more command buttons to the form, as shown below.



Name each of the new controls by selecting the control on your form and changing the name property in the properties window.

Control	Name
Text1	txtInput
List1	lstClientList
Command1	cmdAdd

Command2	cmdDelete
Command3	cmdClear

The controls also require the captions to be changed.

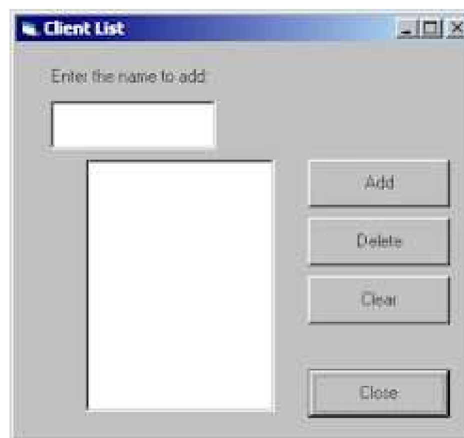
Change the Caption properties as shown below.

Control	**Name**
Label1	Enter the name to add:
cmdAdd	Add
cmdDelete	Delete
cmdClear	Clear

Clear the Text property of the txtInput control.

Check the appearance of the form at this stage.

Click on the play button on the tool bar. Click on the Client List button. The second form should then appear as shown below.



Click on the Close and Quit buttons once you have checked your form.

The add button

The Add button will add what the user has entered in the text box into the list.

Double click on the Add button on your form to display the code window.

Enter the code shown below. Remember that the Private Sub and End Sub statements will have been entered for you by Visual Basic.

```
Private Sub cmdAdd_Click ()
        lstClientList.AddItem txtInput.Text
        txtInput.Text = ""
```

```
        txtInput.SetFocus
End Sub
```

The first line of the code adds an item to the list box (remember the name of the list box is lstClientList). The item which is added is the current value of the Text property of the txtInput control. This will be what the user has entered into the text box.

The next line resets the Text property of the text box to a blank string, ready for the next entry by the user.

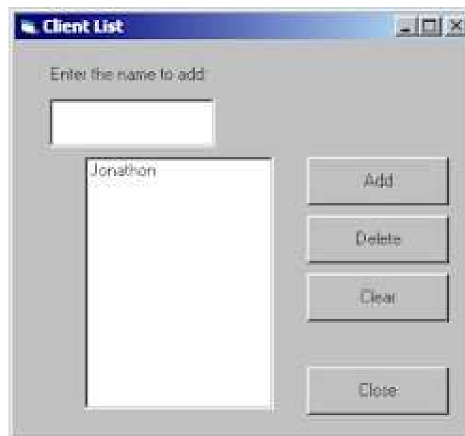
The last line repositions the cursor inside the text box so the user can begin typing the next entry.

Run the application to test the Add button code.

Type your name as the first client to add to the list.

Click on the Add button.

Your name should appear in the list box and the text box should appear empty. The cursor will be positioned in the text box ready for the next entry.



Use the Close and Quit buttons on your forms to stop the application when you have finished.

The Delete button

The Delete button will delete the item that the user has selected in the list box from the list.

Double click on the Delete button on your form to display the code window.

Enter the code shown below. Remember that the Private Sub and End Sub statements will have been entered for you by Visual Basic.

```
Private Sub cmdDelete_Click ()
        If lstClientList.ListIndex >= 0 Then
                lstClientList.RemoveItem lstClientList.ListIndex
```

```
        Else
                Beep
        End If
End Sub
```

The code uses an If statement to test whether the user has selected an item in the list.

The ListIndex property for the lstClientList control holds a number indicating which item in the list has been selected by the user.

If ListIndex is zero then the first item in the list is selected. If ListIndex is equal to one then the next item in the list is selected and so on.

If ListIndex is less than zero it means that nothing is selected.

If something is selected the next line of the code will remove it from the list. Visual Basic will remove the item at list position lstClientList.ListIndex.

If nothing is selected a Beep is sounded.

You should run the application to test the Delete button code.

Click on the play button on the tool bar and click the Client List button.

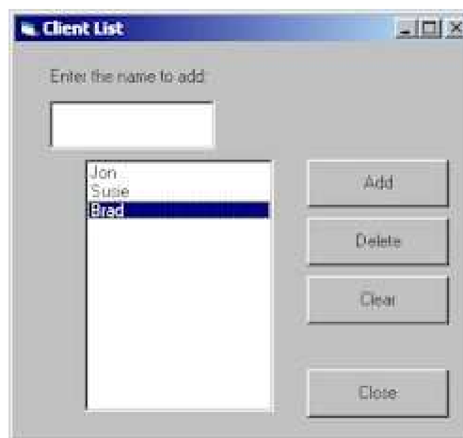
Firstly, test to see what happens if you click the Delete button when the list is empty.

Click the Delete button.

A beep should sound to indicate that nothing in the list is selected for deletion. The action should not result in an error message.

Next, test that the Delete button correctly deletes list items.

Add several items to the list by entering them in the text box and clicking on the Add button.



Click on one of the items to select it.

The item you select will become highlighted.

Click on the Delete button.

The item will be removed from the list.

Try selecting and deleting the other list items.

Stop the application running when you have finished testing.

The Clear button

The Clear button will delete all of the items in the list.

Double click on the Clear button on your form to display the code window.

Enter the code shown below. Remember that the Private Sub and End Sub statements will have been entered for you by Visual Basic.

```
Private Sub cmdClear_Click ()
        lstClientList.Clear
End Sub
```

The code uses the Clear method to delete all the items from the list.

Run the application to test the Clear button code.

Click on the play button on the tool bar. Click on Client List.

Add several items to the list using the Add button.

Click on the Clear button.

All of the items in the list will be deleted.

Stop the application running.

Save the project.

Selecting items from a list

The following example demonstrates how to use a list box to allow your user to make a selection from the list.

The items in this list box example are added at design time.

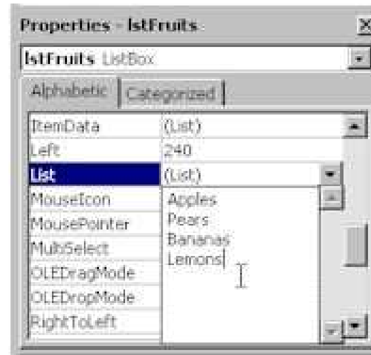
Start a new project.

Add a list box to the form. Change its name to lstFruits.

Select the List property for the list box and click on the drop down arrow.

Type **apples** then press Ctrl Enter

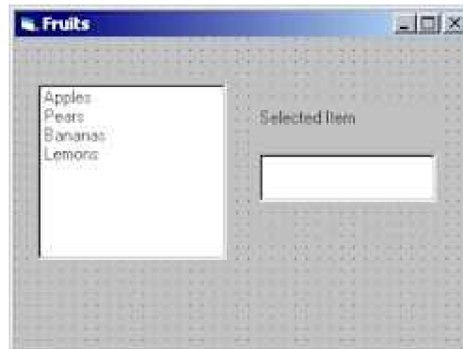
Type the other fruit names shown pressing Ctrl Enter after each one.



If you cannot see the full list of fruit in the list box on your form, make the list box larger.

Add a label with the caption shown and a text box to your form

Change the name of the text box to **txtSelection**.



The List property returns the contents of the list for the specified index.

Add the following code to the click event for the list box.

```
Private Sub lstFruits_Click()
        txtSelection.Text = lstFruits.List(lstFruits.ListIndex)
End Sub
```

Run the application and test that clicking on an item in the list box will cause it to be displayed in the text box.

Stop the application.

List Boxes: Controls That Work Like Arrays

The list box control works a lot like an active array on your form that the user can scroll through, seeing all the items in the list. Often, programmers initialize list boxes with data from arrays. Unlike most controls, you can't add values to a list box control through the Property window, but must add the values at runtime through code.

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

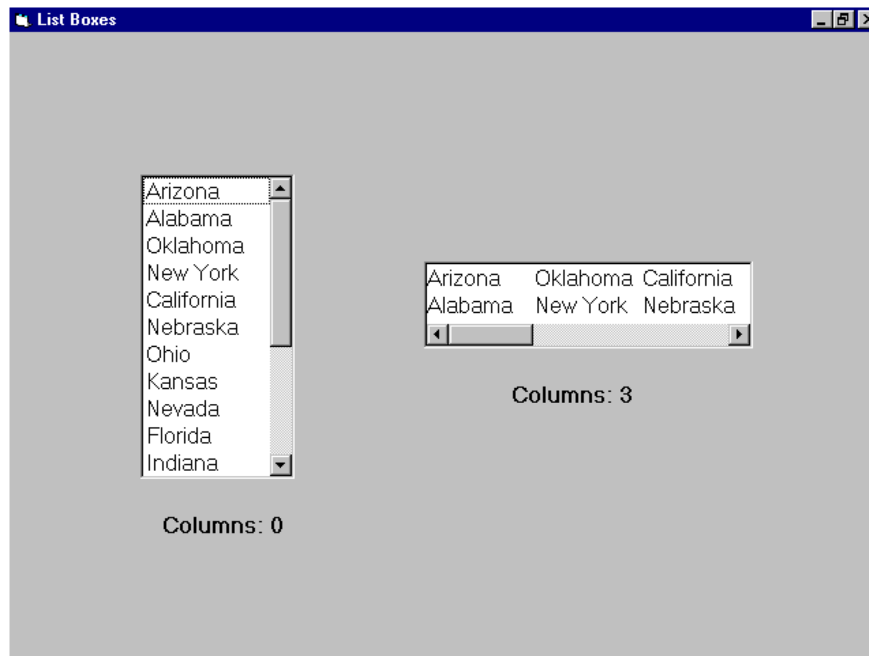
If no item is selected, the **ListIndex** property value is -1. The first item in the list is **ListIndex** 0, and the value of the **ListCount** property is always one more than the largest **ListIndex** value.

The following table contains the list of property values that you can set for list box controls. You've seen many of the properties before because several controls share many of the same properties.

The list box properties.

Property	Description
BackColor	The background color of the list box. It's a hexadecimal number representing one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you're ready to set the BackColor property. The default background color is the same as the form's default background color.
Columns	If 0 (the default), the list box scrolls vertically in a single column. If 1 or more, the list box items appear in the number of columns specified (one or more columns) which the user scrolls the list box horizontally to see all the items if needed. Figure 11.4 shows two identical list boxes, one with a Columns property of 0 and one with a Columns property of 3.
DragIcon	The icon that appears when the user drags the list box control around on the form. (You'll only rarely allow the user to move a list box control, so the Drag... property settings aren't usually relevant.)
DragMode	Either contains 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the list box control but that you, through code, can initiate the dragging if needed.
Enabled	If set to True (the default), the list box control can respond to events. Otherwise, Visual Basic halts event processing for that particular control.
FontBold	True (the default) if the list values are to display in boldfaced characters; False otherwise.
FontItalic	True (the default) if the list values are to display in italicized characters; False otherwise.

FontName	The name of the list box's text style. Typically, you'll use the name of a Windows TrueType font.
FontSize	The size, in points, of the font used for the list box values.
FontStrikethru	True (the default) if the list values are to display in strikethru letters (characters with a dash through each one); False otherwise.
FontUnderline	True (the default) if the list box values are to display in underlined letters; False otherwise.
ForeColor	The color of the values inside the list box.
Height	The height, in twips, of the list box control.
HelpContextID	If you add advanced, context-sensitive help to your application), the HelpContextID provides the identifying number for the help text.
Index	If the list box control is part of a control array, the Index property provides the numeric subscript for each particular list box control (see the next unit).
Left	The number of twips from the left edge of the Form window to the left edge of the list box control.
MousePointer	The shape that the mouse cursor changes to if the user moves the mouse cursor over the list box control. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take.
MultiSelect	If 0-None (the default), the user can select only one list box item. If 1-Simple, the user can select more than one item by clicking with the mouse or by pressing the spacebar over items in the list. If 2-Extended, the user can select multiple items using Shift+click and Shift+arrow to extend the selection from a previously selected item to the current item. Ctrl+click either selects or deselects an item from the list.
Name	The name of the control. By default, Visual Basic generates the names List1, List2, and so on as you add subsequent list box controls to the form.
Sorted	If True, Visual Basic doesn't display the list box values sorted numerically or alphabetically. If False (the default), the values appear in the same order in which the program added them to the list.
TabIndex	The focus tab order begins at 0 and increments every time that you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value.
TabStop	If True, the user can press Tab to move the focus to this list box. If False, the list box can't receive the focus.
Tag	Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the list box control.
Top	The number of twips from the top edge of a list box control to the top of the form.
Visible	True or False, indicating whether or not the user can see (and, therefore, use) the list box control.
Width	The number of twips wide that the list box control consumes.



When placing a list box control on the form, decide how tall you want the list box to be by resizing the control to the size that fits the form best. Remember that if all the list box values don't all fit within the list box, Visual Basic adds scroll bars to the list box so that the user can scroll through the values.

This following table contains all the list box events that you can program. Table contains all the list box events that you can use in a program. You'll rarely write event procedures for list box controls, however. Most of the time, you'll let the user scroll through the list box values to see information they need; programs don't need to respond to list box events as often as they need to respond to command buttons and text boxes.

The list box control's events.

Event	Description
Click	Occurs when the user clicks the list box control
DblClick	Occurs when the user double-clicks the list box control
DragDrop	Occurs when a dragging operation of the list box completes
DragOver	Occurs during a drag operation
GotFocus	Occurs when the list box receives the focus
KeyDown	Occurs when the user presses a key as long as the KeyPreview property is set to True for the controls on the form; otherwise, the form gets the KeyDown event
KeyPress	Occurs when the user presses a key over the list box
KeyUp	Occurs when the user releases a key over the list box
LostFocus	Occurs when the list box loses the focus to another object
MouseDown	Occurs when the user presses a mouse button over the list box
MouseMove	Occurs when the user moves the mouse over the list box

MouseUp	Occurs when the user releases a mouse button over the list box

The following table contains a list of list box control methods that you'll need to use for initializing, analyzing, and removing items from a list box control. Methods works like miniature programs that operate on controls. Here is the format of a method's use on a list box named lstItems:

```
lstItems.AddItem "Arizona"
```

The control name always precedes the method and the dot operator. Any data needed by the method appears to the right of the method.

List box methods.

Method Name	Description
AddItem	Adds a single item to the list box
Clear	Clears all items from the list
List	A string array that holds each item within the list box
ListCount	The total number of items in a list box
RemoveItem	Removes a single item from a list box
Selected	Determines whether the user has selected a particular item in the list box

Use the AddItem method to add properties to a list box control. Suppose that you want to add a few state names to a list box named lstStates. The following code adds the state names:

```
'
Add several states to a list box control

lstStates.AddItem "Arizona"

lstStates.AddItem "Alabama"

lstStates.AddItem "Oklahoma"

lstStates.AddItem "New York"

lstStates.AddItem "California"


lstStates.AddItem "Nebraska"

lstStates.AddItem "Ohio"

lstStates.AddItem "Florida"

lstStates.AddItem "Texas"

lstStates.AddItem "South Dakota"

lstStates.AddItem "Nevada"

lstStates.AddItem
```

```
"Illinois"

lstStates.AddItem "New Mexico"
```

This code would most likely appear in the Form_Load() event procedure so that the list boxes are initialized with their values before the form appears and before the list boxes are seen on the form.

Each item in a list box has a subscript just as each element in an array has a subscript. The first item that you add to a list box has a subscript of 0, the second has a subscript of 1, and so on. To remove the third item from the list box, therefore, your code can apply the RemoveItem method, as follows:

```
lstStates.RemoveItem(2) ' 3rd item has a subscript of 2
```

If you want to remove all items from the list box, use the Clear method. The following simple method removes all the state names from the state list box:

```
lstStates.Clear ' Remove all items
```

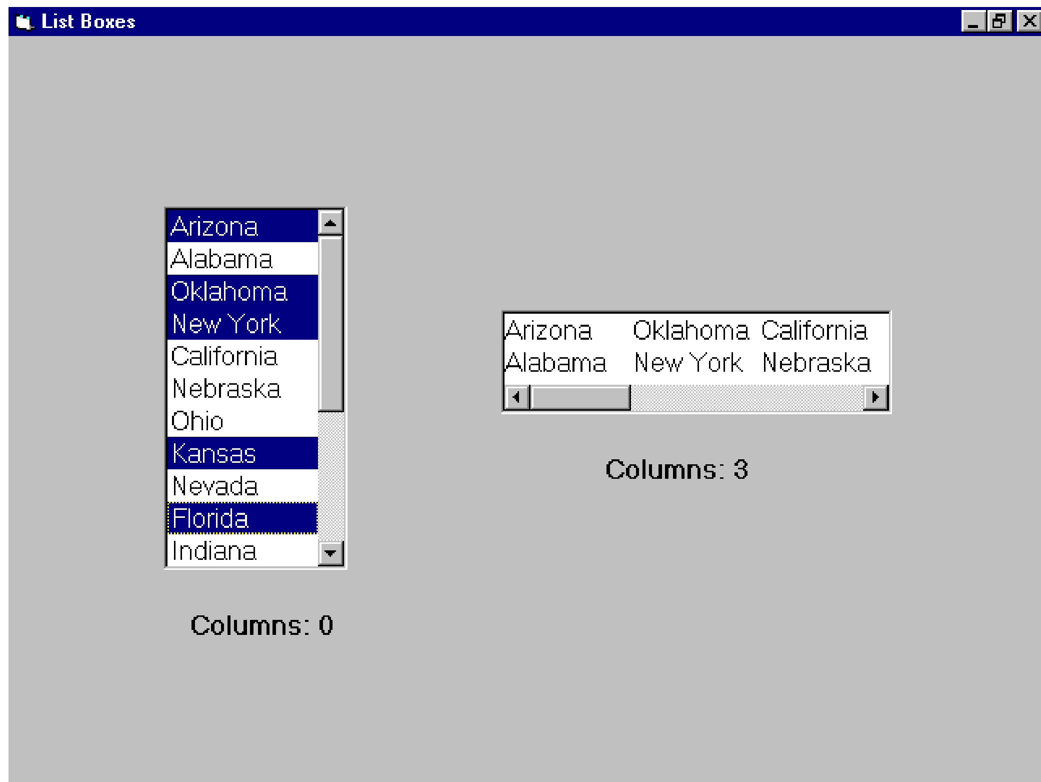
You can assign individual items from a list box control that contains data by using the List method. You must save list box values in string or variant variables unless you convert list box items to a numeric data type using Val() first. The following assignment statements store the first and fourth list box item in two string variables:

```
FirstStringVar = lstStates.List(0)

SecondStringVar = lstStates.List(3)
```

The ListCount method always provides the total number of items in the list box control. For example, the following statement stores the number of list box items in a numeric variable named Num:

```
Num = lstStates.ListCount
```

The Selected method returns either a true or false value that determines whether a user has selected a list box item. The Selected method returns true for possibly more than one list box item if the MultiSelect property is set to either 1-Simple or 2-Extended. Those properties indicate that the user can select more than one item at once



The code in Listing stores every selected item of a list box named lstStates in a string array. The string array is defined with enough elements to hold all the items if the user happens to have selected all fifty items that were first added to the list box.

A list box control holds one or more values through which the user can scroll. Unlike many other controls, the user can only look at and select items in the list box, but not add items to the list box. Through methods, you can, however, add items, delete items, and check for selected items in the list box by using the methods inside your code procedures

WEEK 11:

CONCEPT OF DATA FILE IN VB/ ADVANCED VB CONTROLS

During this week you will learn :

- Creating a sequential file
- Creating a random file
- Combo Boxes
- The Timer Control
- Scrolling the Scroll Bars

CONCEPT OF FILES

A Data file can be described as the collection of related records that is stored in an auxiliary storage media such as magnetic tape or disk with a file name for example all records about all the students in Kaduna polytechnic form a student file. In a data file a line of information is known as a logical record. A file can be organized serially, sequentially or randomly.

Visual Basic can handle both Sequential and Random access files.

SEQUENTIAL FILE PROCESSING COMMANDS IN VISUAL BASIC

visual basic uses some of the traditional basic file processing commands but with little variations. Some of these commands and statements are highlighted below:

1**. OPEN STATEMENT:** The link between the external disk filename and the internal file number is made through the open statement.

Syntax:

Open filename for mode As # file number:

The filename is a string expression or variable

The modes can either be input, output , append or binary

eg. Open "stud.dat" for input as #1

2. **CLOSE STATEMENT:** This is used to close an opened file at the end of the file session.

Syntax

Close # file number

Eg. Close #1

3. **Print # and Write # statements;** These are used to write into an opened file

Syntax:

Print #, Filenumber,{variable list}

Eg. Print # 1, Sname,Saddress,Sage,Sex,……

Write# 1,Sname,Saddress,Sage,Sex,……

Write# and Print # Statements differes only in the way they handle strings

4. **Input # statement:** This read Data back from an existing file

Syntax;

Input # file number,{Variable list}

Eg. Input# 2, Sname,Saddress,Sage,Sex,……

5. **Kill Statements: T**his Delete a file from the disk

Syntax: Kill"filename"

Eg. Kill"Temp.dat"

6. **EOF Function: T**his is used to test for End of file

Syntax: EOF(Channel)

Eg.

```
Do While NOT EOF(1)

   Loop
```

THE RANDOM FILE

Opening Random file is a little different from that of a sequential file. We use the "For Random" option with some other options as described below.

Open "stud.dat" for random as #2 len=len(myrecord)

In a random file user defined data type are used to group all the related variables together into one record as shown below

```
Private sub form_load()

        Type single-Record

        Record-key as integer

        Sname As String*20

        Snum as string*10

End type

Dim myrecord As single-record

Open"stud.txt"for random as #2, Len = len(myrecord)

End sub
```

The len parameter is used by the open statement to specify the total number of character in a record

Put Statement: The put statement is used to write data into a random file instead of the print statement used in a sequential file.

Eg Put #2, 1,Myrecord

The Get statement: This is used to read data from the random file, instead of input statement in sequential file. in addition, it specify the record number to be read and the name of the variable into which data is to be copied.

Eg. Get #1,myrecord.

Note: Database gives you a better way to handle your data records.

Combo Boxes

The combo box control is a combination of a list box and a text box. Items are stored in a list and viewed using a drop down menu.

The contents of the Text property of the combo box is displayed at the top of the combo box.

Combo boxes work a little like list boxes except that the user can add items to a combo box at runtime. There are three kinds of combo boxes determined by the Style property. The AddItem and RemoveItem methods are popular for combo boxes, although all of the list box methods that you learned in the previous lesson apply to combo boxes as well.

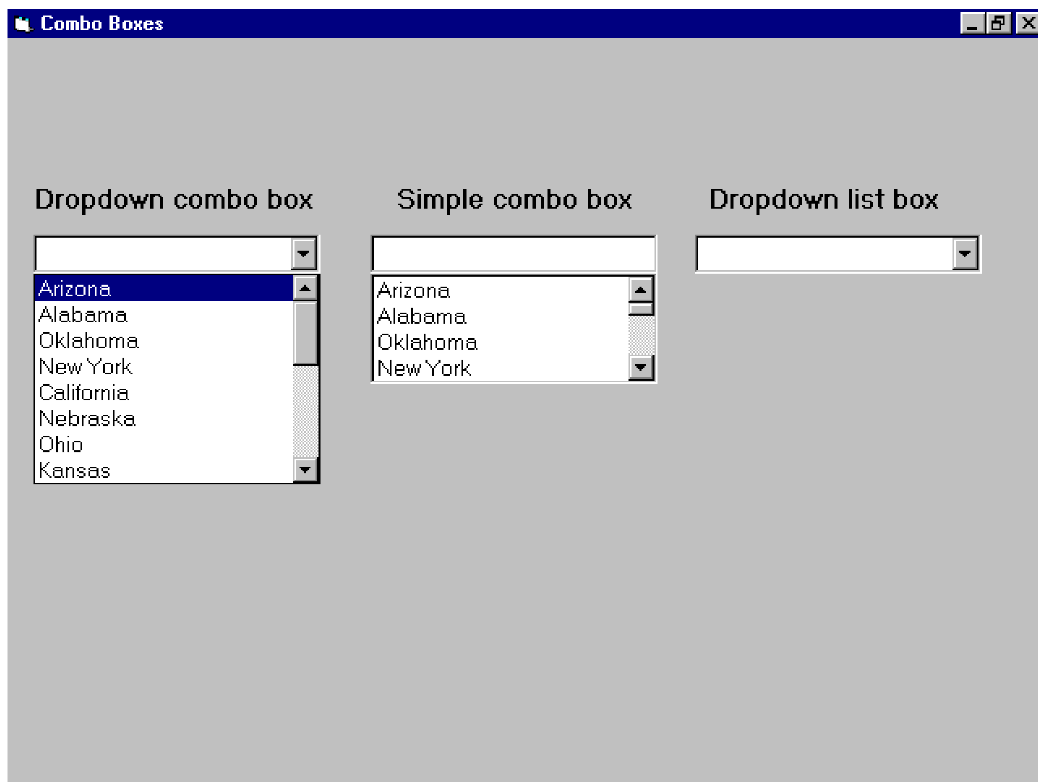
Figure shows the location of the combo box control on the Toolbox window. No matter what kind of combo box you want to place on the form, you'll use the same combo box control on the toolbox to add the combo box to the form.

There are three kinds of combo boxes, as follows:

- A dropdown combo box takes up only a single line on the form unless the user opens the combo box (by pressing the combo box's down arrow) to see additional values. The user can enter additional items at the top of the dropdown combo box and select items from the combo box.

- A simple combo box always displays items as if they were in a list box. The user can add items to the combo box list as well.

- A dropdown list box is a special list box that the user can't enter new items into, but that normally appears closed to a single line until the user clicks the down arrow button to open the list box to its full size. Technically, dropdown list boxes are not combo box controls but work more like list boxes. The reason dropdown list boxes fall inside the combo box control family is that you place dropdown list boxes on forms by clicking the combo box control and setting the appropriate combo box property (Style).



The following table contains a description of every combo list property value that you can set in the Property window.

The combo box properties.

Property	Description
BackColor	The background color of the combo box. This is a hexadecimal number representing one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you're ready to set the BackColor property. The default background color is the same as the form's default background color.

DragIcon	The icon that appears when the user drags the combo box control around on the form. (You will only rarely allow the user to move a combo box control, so the Drag... property settings aren't usually relevant.)
DragMode	Either contains 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the combo box control but that you, through code, can initiate the dragging if needed.
Enabled	If set to True (the default), the combo box control can respond to events. Otherwise, Visual Basic halts event processing for that particular control.
FontBold	True (the default) if the combo values are to display in boldfaced characters; False otherwise.
FontItalic	True (the default) if the combo values are to display in italicized characters; False otherwise.
FontName	The name of the combo box's text style. Typically, you'll use the name of a Windows TrueType font.
FontSize	The size, in points, of the font used for the combo box values.
FontStrikethru	True (the default) if the combo values are to display in strikethru letters (characters with a dash through each one); False otherwise.
FontUnderline	True (the default) if the combo box values are to display in underlined letters; False otherwise.
ForeColor	The color of the values inside the combo box.
Height	The height, in twips, of the combo box control.
HelpContextID	If you add advanced, context-sensitive help to your application), the HelpContextID provides the identifying number for the help text.
Index	If the combo box control is part of a control array, the Index property provides the numeric subscript for each particular combo box control. (See the next unit).
Left	The number of twips from the left edge of the Form window to the left edge of the combo box control.
MousePointer	The shape that the mouse cursor changes to if the user moves the mouse cursor over the combo box control. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take.
Name	The name of the control. By default, Visual Basic generates the names Combo1, Combo2, and so on as you add subsequent combo box controls to the form.
Sorted	If True, Visual Basic doesn't display the combo box values sorted numerically or alphabetical. If False (the default), the values appear in the same order in which the program added them to the list.
Style	The default, 0-Dropdown Combo, produces a dropdown combo box control. 1-Simple Combo turns the combo box into a simple combo box control. 2-Dropdown list turns the combo box into a dropdown list box control.
TabIndex	The focus tab order begins at 0 and increments every time that you add a new control. You can change the focus order by changing the controls'

	TabIndex to other values. No two controls on the same form can have the same TabIndex value.
TabStop	If True, the user can press Tab to move the focus to this combo box. If False, the combo box can't receive the focus.
Tag	Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the combo box control.
Text	The initial value only that the user sees in the combo box.
Top	The number of twips from the top edge of a combo box control to the top of the form.
Visible	True (the default) or False, indicating whether the user can see (and, therefore, use) the combo box control.
Width	The number of twips wide that the combo box control consumes.

Note: Notice that there is no MultiSelect combo box property as there is with list box controls. The user can select only one combo box item at any one time.

The following table contains a list of the combo box events for which you can write matching event procedures when your program must react to a user's manipulation of a combo box.

The combo box control's events.

Event	Description
Change	Occurs when the user changes the value in the data entry portion of the dropdown combo or the simple combo box; not available for dropdown list boxes because the user can't change data in them
Click	Occurs when the user clicks the combo box control
DblClick	Occurs when the user double-clicks the combo box control
DragDrop	Occurs when a dragging operation of the combo box completes
DragOver	Occurs during a drag operation
DropDown	Occurs when the user opens a dropdown combo box or a dropdown list box
GotFocus	Occurs when the combo box receives the focus
KeyDown	Occurs when the user presses a key as long as the KeyPreview property is set to True for the controls on the form; otherwise, the form gets the KeyDown event
KeyPress	Occurs when the user presses a key over the combo box
KeyUp	Occurs when the user releases a key over the combo box
LostFocus	Occurs when the combo box loses the focus to another object

The combo box controls support the same methods that the list box controls support. Therefore, you can add, remove, count, and select items from the combo box if you apply the methods seen in The following table.

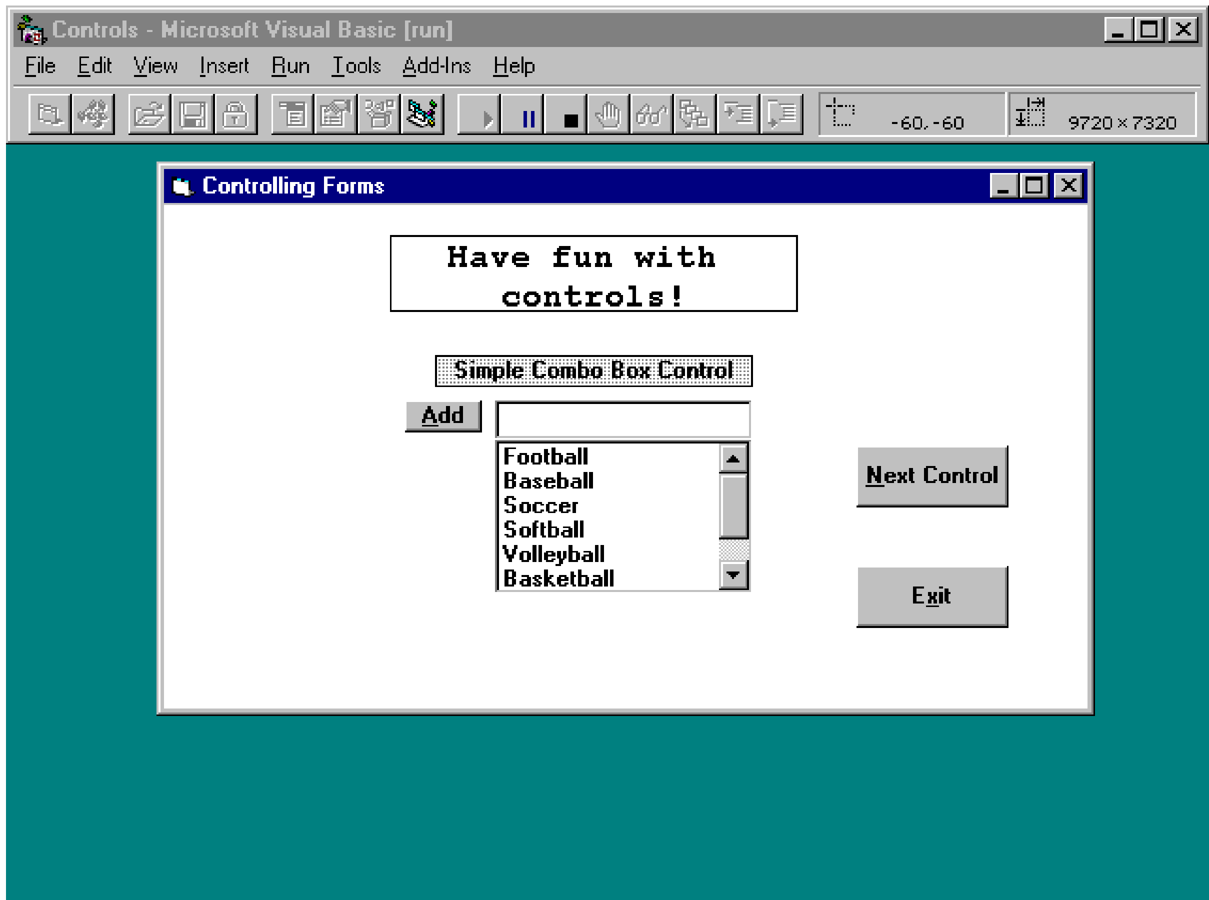
The following list contains a command button's event procedure that adds a value to a combo box's list of items. Unlike text box controls, you'll need to provide the user with some data entry mechanism, such as a command button, that informs the program when to use the AddItem method to add an item to a combo box.

Note: Listing assumes that another procedure, such as the Form_Load() procedure, added the initial values to the combo box.

Listing A command button's event procedure that adds an item to a combo box.

```
Sub cmdSimple_Click ()

        ' Add the user's item to the simple combo

        comSCtrl.AddItem comSCtrl.Text

        comSCtrl.Text = ""

        comSCtrl.SetFocus

End Sub
```

Output: Figure contains a screen that you saw in the second lesson of the book. The CONTROLS.VBP application demonstrates the simple combo box and shows how you can set up an application to add items to a list of values.



Analysis: The command button in Figure 11.8 is named cmdSimple, so clicking the command button executes the event procedure shown in Listing 11.3. Line 3 stores the combo box's Text property value to that combo box's list of items. The combo box will not contain a user's entry in the upper data entry portion of the combo box until an AddItem method adds that entry to the list. The Text property always holds the current value shown in the data entry portion of the combo box, but the AddItem method must add that value to the list.

As soon as the user's entry is added, line 4 erases the data entry portion of the combo box. After all, the user's text will now appear in the lower listing portion of the combo box (thanks to line 3), so line 4 clears the data entry area for more input. In addition, line 5 sets the focus back to the combo box (the focus appears in the data entry area that line 4 cleared) so that the user is ready to add an additional item to the combo box.

The Timer Control

A timer control allows you to generate events at specified time intervals. For example, you could build your own version of the Windows Clock application by displaying the time in a label and using a timer control to update the display every second. Your application might look like the following:

Starting and stopping a timer control

A timer is started by setting the **Enabled** property to TRUE and giving the **Interval** property a value greater than 0.

The timer can be stopped while the application is running by setting the **Enabled** property to FALSE or setting the **Interval** property to 0.

Designing the Clock application

Start a new project and add two labels with Name property set to **lblTime** and **lblDate** respectively. Change the FontName for the labels to Courier New (a non-proportional font) and choose a suitable FontSize for each label. Change the caption of the form to **Clock**.

Add a Timer Control to the form. It doesn't matter where you put this control because it is invisible when the project is running.

Select the Properties window for the Timer Control and set Enabled to **True** and the Interval property to **1000**.

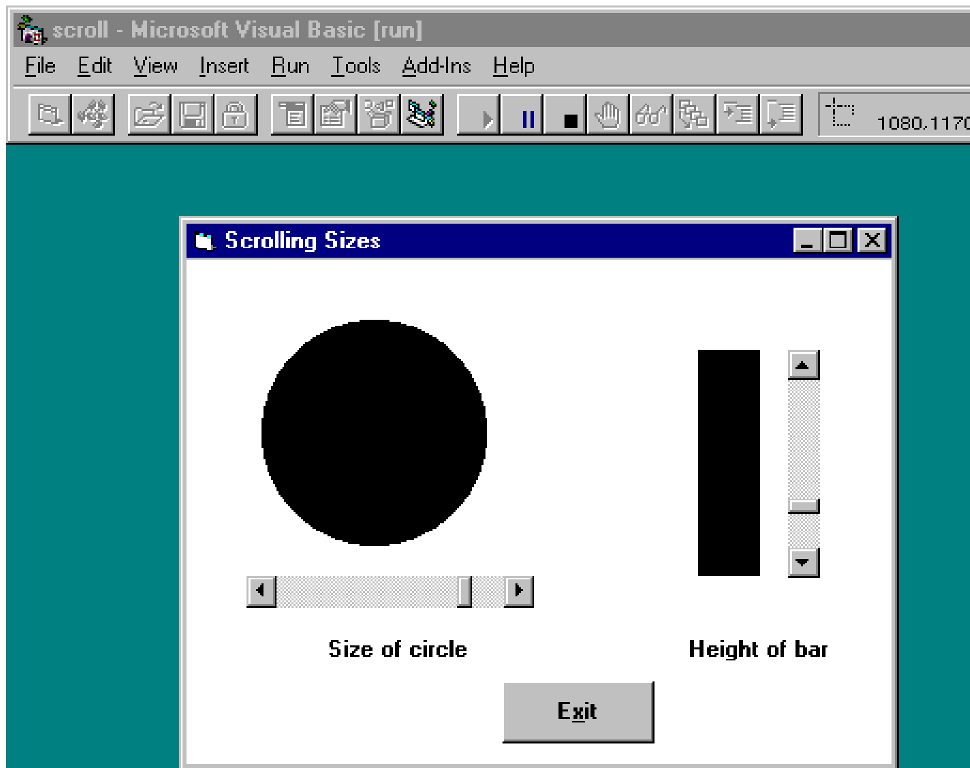
The units for the interval are milliseconds. With these settings, the code in the event Timer1_Timer will be executed approximately every second.

Add the following event procedure code and try out your application. `Private Sub`

```
Form_Load()
        lblDate.Caption = Date
End Sub

Private Sub Timer1_Timer()
        lblTime.Caption = Time
End Sub
```

Scrolling the Scroll Bars

The scroll bars give the user the ability to control changing values. Rather than type specific values, the user can move the scroll bars with the mouse to specify relative positions within a range of values.



The following table contains a list of the scroll bar properties. The most unique and important property values for a scroll bar are the LargeChange, Max, Min, and SmallChange.

The scroll bar properties.

Property	Description
DragIcon	Specifies the icon that appears when the user drags the scroll bar around on the form. (You'll only rarely allow the user to move a scroll bar, so the Drag... property settings aren't usually relevant.)
DragMode	Contains either 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the scroll bar but that you, through code, can initiate the dragging if needed.
Enabled	If set to True (the default), the scroll bar can respond to events. Otherwise, Visual Basic halts event processing for that particular control.
Height	Contains the height, in twips, of the scroll bar.

HelpContextID	If you add advanced, context-sensitive help to your application, the HelpContextID provides the identifying number for the help text.
Index	If the scroll bar is part of a control array, the Index property provides the numeric subscript for each particular scroll bar.
LargeChange	Specifies the amount that the scroll bar changes when the user clicks within the scroll bar's shaft area.
Left	Holds the number of twips from the left edge of the Form window to the left edge of the scroll bar.
Max	Indicates the maximum number of units that the scroll bar value represents at its highest setting. The range is from 1 to 32767 (the default).
Min	Indicates the minimum number of units the scroll bar value represents at its lowest setting. The range is from 1 (the default) to 32767.
MousePointer	Contains the shape that the mouse cursor changes to if the user moves the mouse cursor over the scroll bar. The possible values are from 0 to 12, and represent a range of different shapes that the mouse cursor can take on.
Name	Contains the name of the control. By default, Visual Basic generates the names VScroll1, VScroll2, and so on (for vertical scroll bars), and HScroll1, HScroll2, and so on (for horizontal scroll bars) as you add subsequent scroll bars to the form.
SmallChange	Specifies the amount that the scroll bar changes when the user clicks an arrow at either end of the scroll bar.
TabIndex	Determines that the focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value.
TabStop	If True, the user can press Tab to move the focus to this scroll bar. If False, the scroll bar can't receive the focus.
Tag	Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the scroll bar.
Top	Holds the number of twips from the top edge of a scrollbar to the top of the form.
Value	Contains the unit of measurement currently represented by the position of the scroll bar.
Visible	Contains either True or False, indicating whether the user can see (and, therefore, use) the scroll bar.
Width	Holds the number of twips wide that the scroll bar consumes.

Tip: Prefix the names of your horizontal scroll bars with the hsb prefix and your vertical scroll bars with the vsb prefix so that you can easily distinguish them from each other.

When you place a scroll bar on a form, you must decide at that time what range of values the scroll bar is to represent. The scroll bar's full range can extend from 1 to 32767. Set the Min property to the lowest value represented by the scroll bar. Set the Max property to the highest value represented by the scroll bar.

When the user eventually uses the scroll bar, the scroll bar arrows control small movements in the scroll bar's value determined by the SmallChange property. Clicking the empty shaft on either side of the scroll box produces a positive or negative change in the value represented by the LargeChange property. The user can drag the scroll bar itself to any position within the scroll bar shaft to jump to a specific location instead of changing the value gradually.

Suppose, for example, that a horizontal scroll bar was to represent a range of whole dollar amounts from $5 to $100. When the user clicks the scroll arrows, the scroll bar's value is to change by one dollar. When the user clicks the empty shaft on either side of the scroll box, the scroll bar's value is to change by five dollars. Here are the property values that you would set that determine how Visual Basic interprets each click of the scroll bar:

Min: 5

Max: 100

SmallChange: 1

LargeChange: 5

The physical size of the scroll bar has no bearing on the scroll bar's returned values when the user selects from the scroll bar. Adjust the scroll bars on your form so that the scroll bars are wide enough or tall enough to look appropriate sizes for the items that they represent.

The following Listing contains the SCROLL.MAK code that you can load and run to adjust the circle and bar sizes that you saw in above Figure

Review: There are two scroll bars, a horizontal scroll bar and a vertical scroll bar, that give the user the ability to select from a range of possible values without having to enter individual values.

Listing. The code for the SCROLL.MAK application.

```
Option Explicit

Sub Form_Load ()

' Set initial scroll bar values

        hsbBar.Value = 1800 ' Circle's default width

        vsbBar.Value = 1800 ' Bar's default height

End Sub


Sub hsbBar_Change ()

        ' As user clicks the scroll bar,

        ' the width of the circle adjusts

        shpCircle.Width = hsbBar.Value
```

```
End Sub

Sub vsbBar_Change ()
        ' As user clicks the scroll bar,
        ' the height of the bar adjusts
        shpBar.Height = vsbBar.Value
End Sub
```

Analysis: Here are the vital horizontal scroll bar properties that were set during the design of the form:

Min: 50

Max: 2100

SmallChange: 50

LargeChange: 100

The shape control that contains the circle (named shpCircle) has its Width property set to 2100, so the largest that the circle could appear within the control was 2100 twips. Hence the use of 2100 for the Max property.

Here are the vital vertical scroll bar properties that were set during the design of the form:

Min: 50

Max: 2300

SmallChange: 50

LargeChange: 100

The shape control that contains the bar (named shpBar) has its Height property set to 2300, so the tallest that the bar could appear within the control was 2300 twips. Hence the use of 2300 for the Max property.

The two Min properties of 50 keep both the circle and bar from shrinking entirely when the user minimizes either control.

Lines 5 and 6 set the initial values for the circle's width and the bar's height to 1800, so both shapes are fairly large to begin with. Actually, the lines set the initial runtime values for both the scroll bars, which, in turn, generates the Change() event procedures that follow in the code.

Line 12 ensures that the circle's width increases or decreases, depending on the value of the horizontal scroll bar's Value property. Line 18 ensures that the bar's height increases or decreases, depending on the value of the vertical scroll bar's Value property.

WEEK 12:

MANAGING KEYBOARD AND SCREEN I/O

During this week you will learn :

- MsgBox Function
- InputBox Function

In Windows-based applications, dialog boxes are used to prompt the user for data needed by the application to continue or to display information to the user.

MsgBox Function

Displays a message in a dialog box, waits for the user to click a button, and returns an **Integer** indicating which button the user clicked.

Syntax

MsgBox(prompt[, **buttons**] [, **title**] [, **helpfile**, **context**])

The **MsgBox** function syntax has these named arguments:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of **prompt** is approximately 1024 characters, depending on the width of the characters used. If **prompt** consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13)), a linefeed character (**Chr**(10)), or carriage return – linefeed character combination (**Chr**(13) & **Chr**(10)) between each line.
buttons	Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for **buttons** is 0.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit **title**, the application name is placed in the title bar.
helpfile	Optional. String expression 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	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If **context** is provided, **helpfile** must also be provided.

Settings

The **buttons** argument settings are:

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 default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	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 systems

The first group of values (0–5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the **buttons** argument, use only one number from each group.

Note These constants are specified by Visual Basic for Applications. As a result, the names can be used anywhere in your code in place of the actual values.

Return Values

Constant	Value	Description
vbOK	1	**OK**
vbCancel	2	**Cancel**
vbAbort	3	**Abort**
vbRetry	4	**Retry**
vbIgnore	5	**Ignore**
vbYes	6	**Yes**
vbNo	7	**No**

Remarks

When both **helpfile** and **context** are provided, the user can press F1 to view the Help topic corresponding to the **context**. Some host applications, for example, Microsoft Excel, also automatically add a **Help** button to the dialog box.
If the dialog box displays a **Cancel** button, pressing the ESC key has the same effect as clicking **Cancel**. If the dialog box contains a **Help** button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

Example :

This example uses the **MsgBox** function to display a critical-error message in a dialog box with Yes and No buttons. The No button is specified as the default response. The value returned by the **MsgBox** function depends on the button chosen by the user. This example assumes that `DEMO.HLP` is a Help file that contains a topic with a Help context number equal to `1000`.

```
Dim Msg, Style, Title, Help, Ctxt, Response, MyString
Msg = "Do you want to continue ?"   ' Define message.
Style = vbYesNo + vbCritical + vbDefaultButton2   ' Define buttons.
Title = "MsgBox Demonstration"   ' Define title.
Help = "DEMO.HLP"   ' Define Help file.
Ctxt = 1000   ' Define topic
      ' context.
      ' Display message.
Response = MsgBox(Msg, Style, Title, Help, Ctxt)
If Response = vbYes Then   ' User chose Yes.
   MyString = "Yes"   ' Perform some action.
Else   ' User chose No.
   MyString = "No"   ' Perform some action.
End If
```

InputBox Function

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns a String containing the contents of the text box.

Syntax

InputBox(prompt[, **title**] [, **default**] [, **xpos**] [, **ypos**] [, **helpfile**, **context**])

The **InputBox** function syntax has these named arguments:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of **prompt** is approximately 1024 characters, depending on the width of the characters used. If **prompt** consists of more than one line, you can separate the lines using a carriage return character (**Chr**(13)), a linefeed character (**Chr**(10)), or carriage return–linefeed character combination (**Chr**(13) & **Chr**(10)) between each line.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit **title**, the application name is placed in the title bar.
default	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit **default**, the text box is displayed empty.
xpos	Optional. Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If **xpos** is omitted, the dialog box is horizontally centered.
ypos	Optional. Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If **ypos** is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.
helpfile	Optional. String expression 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	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If **context** is provided, **helpfile** must also be provided.

Remarks

When both **helpfile** and **context** are provided, the user can press F1 to view the Help topic corresponding to the **context**. Some host applications, for example, Microsoft Excel, also automatically add a **Help** button to the dialog box. If the user clicks **OK** or presses ENTER , the **InputBox** function returns whatever is in the text box. If the user clicks **Cancel**, the function returns a zero-length string ("").

Example :

This example shows various ways to use the **InputBox** function to prompt the user to enter a value. If the x and y positions are omitted, the dialog box is automatically centered for the respective axes. The variable `MyValue` contains the value entered by the user if the user clicks **OK** or presses the ENTER key . If the user clicks **Cancel**, a zero-length string is returned.

```
Dim Message, Title, Default, MyValue
Message = "Enter a value between 1 and 3"   ' Set prompt.
Title = "InputBox Demo"   ' Set title.
Default = "1"   ' Set default.
' Display message, title, and default value.
MyValue = InputBox(Message, Title, Default)

' Use Helpfile and context. The Help button is added automatically.
MyValue = InputBox(Message, Title, , , , "DEMO.HLP", 10)

' Display dialog box at position 100, 100.
MyValue = InputBox(Message, Title, Default, 100, 100)
```

WEEK 13:

DATABASE MANAGEMENT IN VISUAL BASIC

During this week you will learn :

- The Basic Elements of a Database
- Using Data Control
- Data Access Objects (DAO)
- DAO Advantages and Disadvantages
- Visual Basic Wizard
- Data Form Wizard

All applications use structured information of one kind or another, whether it is accounting data, scientific measurements, employee information, or a list of recipes. Data access in Microsoft Visual Basic gives you the tools to create and use structured database systems to manage your application's data.

These tools include the Microsoft Jet database engine, the Data control, and the data access objects (DAO) programming interface.

The Basic Elements of a Database

Element	Description
Database	A group of data tables that contain related information.
Table	A group of data records, each containing the same type of information. In the phone book example, the book itself is a table.
Record	A single entry in a table, consisting of a number of data fields. In a phone book, a record is one of the single-line entries.
Field	A specific piece of data contained in a record. In a phone book, at least four fields can be identified: last name, first name, address, and phone number.
Index	A special type of table that contains the values of a key field or fields and contains pointers to the location of the actual record. These values and pointers are stored in a specific order and can be used to present data in that order. For the phone book example, one index might be used to sort the information by last and first name; another index might be used to sort the information by street address; and a third might be used to sort the information by phone number.
Query	A command, based on a specific set of conditions or criteria, designed to retrieve a certain group of records from one or more tables or to perform an operation on a table. For example, you would write a query that could show all the students in a class whose last name begins with S and who have a grade point average of more than 3.0.
Recordset	A group of records, created by a query, from one or more tables in a database. The records in a recordset are typically a subset of all the records in a table. When the recordset is created, the number of records and the order in which they're presented can be controlled by the query that creates the recordset.

The Microsoft Jet database engine provides the means by which Visual Basic interacts with databases. You use it with Visual Basic to access databases and database functionality. The Jet engine is shared by Visual Basic, Microsoft Access, and other Microsoft products, and it lets you work with a wide variety of data types, including several types of text and numeric fields. These different data types give you a great deal of flexibility in designing database applications.

Using Data Control

In the following example, we will create a simple database application which enable one to browse customers' names.

To create this application, insert the data control into the new form.

Place the data control somewhere at the bottom of the form. Name the data control as data_navigator.

To be able to use the data control, we need to connect it to any database. We can create a database file using any database application but I suggest we use the database files that come with VB6. Let select NWIND.MDB as our database file.

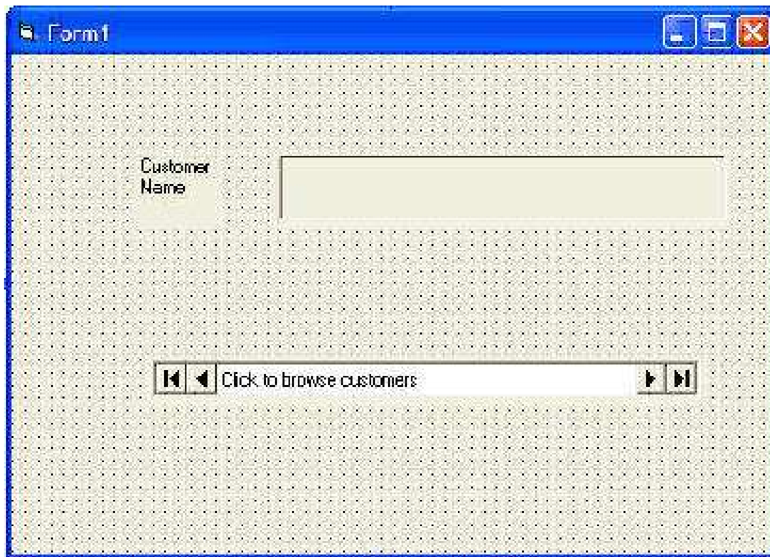
To connect the data control to this database, double-click the **DatabaseName** property in the properties window and select the above file, i.e NWIND.MDB.

Next, double-click on the **RecordSource** property to select the customers table from the database. You can also change the caption of the data control to anything but I use "Click to browse Customers" here. After that, we will place a label and change its caption to Customer Name.

Last but not least, insert another label and name it as cus_name and leave the label empty as customers' names will appear here when we click the arrows on the data control. We need to bind this label to the data control for the application to work.

To do this, open the label's **DataSource** and select data_navigator that will appear automatically. One more thing that we need to do is to bind the label to the correct field so that data in this field will appear on this label. To do this, open the **DataField** property and select ContactName. Now, press F5 and run the program. You should be able to browse all the customers' names by clicking the arrows on the data control.

The Design Interface.



The Runtime Interface

You can also add other fields using exactly the same method. For example, you can add adress, City and telephone number to the database browser.

Data Access Objects (DAO)

If you use either the Professional or Enterprise editions, you can learn how to program Visual Basic by using DAO (Data Access Objects). Data Access Objects are database objects that you create and manage with your program code.
The primary reason for mastering DAO is because it offers several advantages over the Data control. DAO gives you more control and speed in accessing databases. Although using DAO takes a little more knowledge than using the Data control and its related controls, DAO is the choice among most VB programmers due to its powerful advantages.

DAO Advantages and Disadvantages

The Data control is simple to use but doesn't offer extremely fast database access. Although today's computers run quickly, you'll notice speed degradation when you use the Data control in large database tables, especially ODBC-based databases.

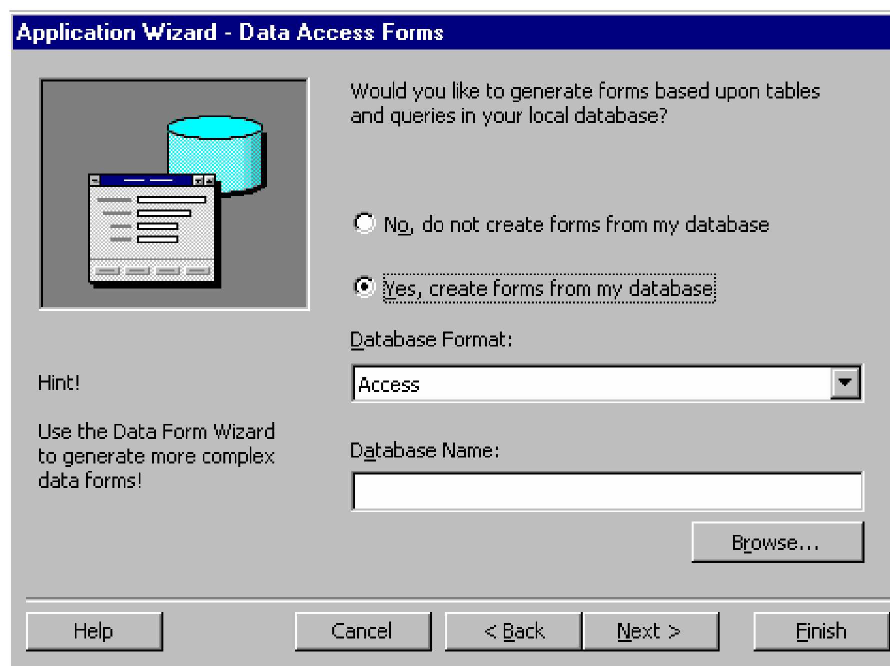
When you use DAO, you must write more program code than you have to write with the Data control. As you saw in the first topic section, you can program the Data control primarily through setting property values. Although you can write code that accesses various Data control methods, straightforward database access is less involved with the Data control.

The DAO lets you control the data access in a much stricter way than with the Data control. The ease of the Data control reflects its inability to be flexible. Also, the overhead of the Data control doesn't burden DAO-based programs. DAO uses the recordset concept for most of its operations, and you can create a recordset variable just as you can create other kinds of object variables in Visual Basic.

Visual Basic Wizard

The preceding topic section only scratched the surface of DAO, but you now know the fundamental requirements and issues that surround DAO programming. You deserve a break, so this final topic section demonstrates a way that you can let Visual Basic do all the work--writing a database application by responding to a few dialog boxes from the VB Application Wizard.

The VB Application Wizard lets you add database capabilities to your project without extensive programming. As you'll see when you follow this topic section's example, the resulting code is fairly complete and forms the basis for a true database application.
The New Project dialog box contains the VB Application Wizard that you've used a few times throughout this book. When you get to the dialog box shown in Figure, you can click the Yes option to add database support to your application.



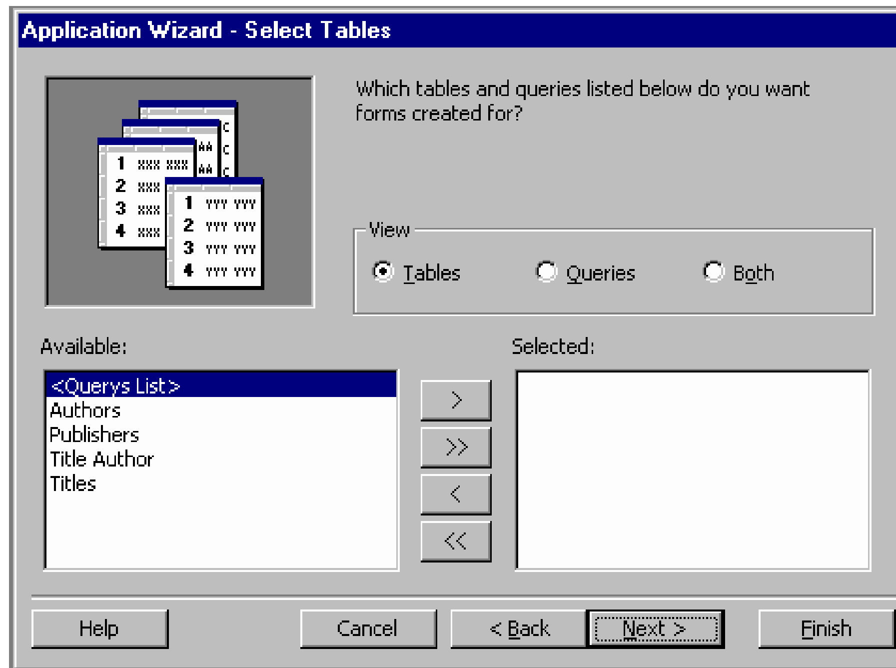
The VB Application Wizard creates database-based applications.
After you select the database option, the rest of the wizard changes dramatically from the dialog boxes that you've seen so far. The next "Example" and "Next Step" sections describe the wizard in more detail.

Example

From the File menu choose New Project, double-click the VB Application Wizard icon, and click Next six times to accept all the VB Application Wizard defaults and to display the dialog box you saw earlier in Figure. Perform these steps to begin the database-aware application:

1. Click Yes to request that the wizard add a database form to the application. You must now select a database on which the wizard will base the form's fields.
2. Click the Browse button and locate the Biblio.mdb database located in the VB directory. Notice that you can select from a wide variety of database systems by

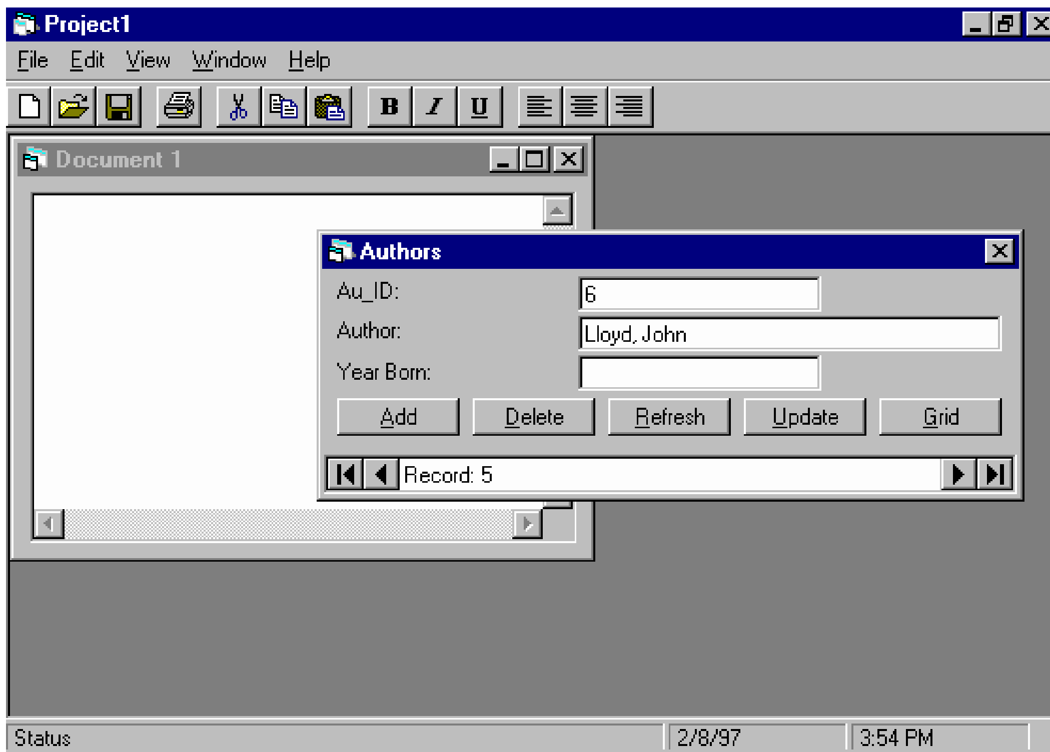
opening the Database Format drop-down list. (For this application, retain the default database, Access.) Click Next to display the Select Tables dialog box



Visual Basic must know the table you want to access.

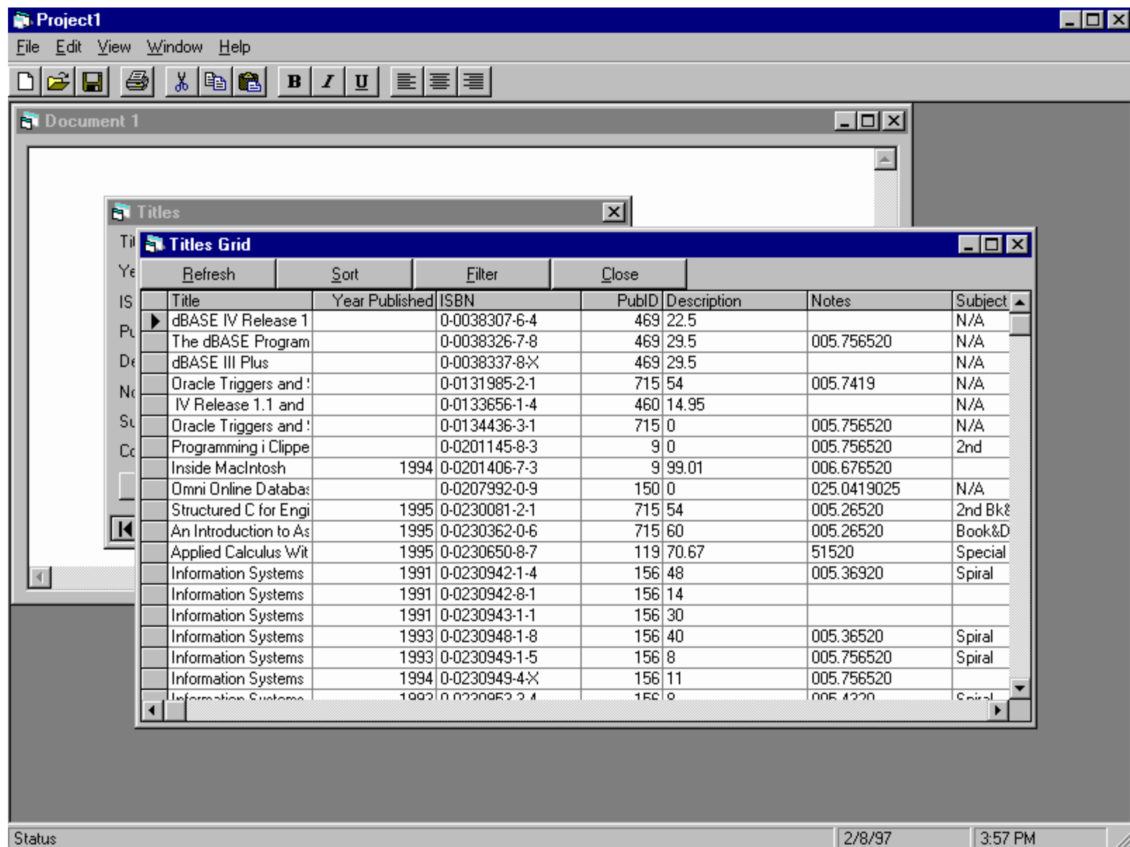
3. Your application can retrieve data directly from the database tables or from any defined queries. As you might recall from earlier in this lesson, a query is a predefined selection criterion for records and fields. A query is just a named instruction set that produces a subset of the database data. The big advantage of queries over table access is that a query, if predefined by users of the database system that generated the database file, can contain data from multiple database tables. The Biblio.mdb database contains only one query, named All Titles (you can click the Queries option to see the query name).
4. This dialog box lets you add multiple tables, so you can select data from more than one table even if no query exists, but the query is often more efficient if defined properly. For this application, however, you'll select two tables: Authors and Titles.
5. Click the Authors table and then click the > button to send the Authors table to the Selected list. Now send the Titles table to the Selected list.
6. Click Next and Finish to finish the wizard and create the project.

The wizard generates a substantial amount of database code for you and designs forms that make the two tables available to the application. When you run the application, choose Authors from the View menu to display the Form window shown in Figure.



The wizard created the form that updates the Authors table.

From the View menu, choose Titles to open the Titles dialog box. Perhaps the most surprising feature of the application appears when you click Grid. Visual Basic formats the table's data into the worksheet-like format shown in Figure. You can resize columns, sort on columns (by clicking a column title), and scroll through the grid to view multiple records at one time. The grid comes from the DBGrid control that the wizard added to the application's Toolbox window during the project's creation.

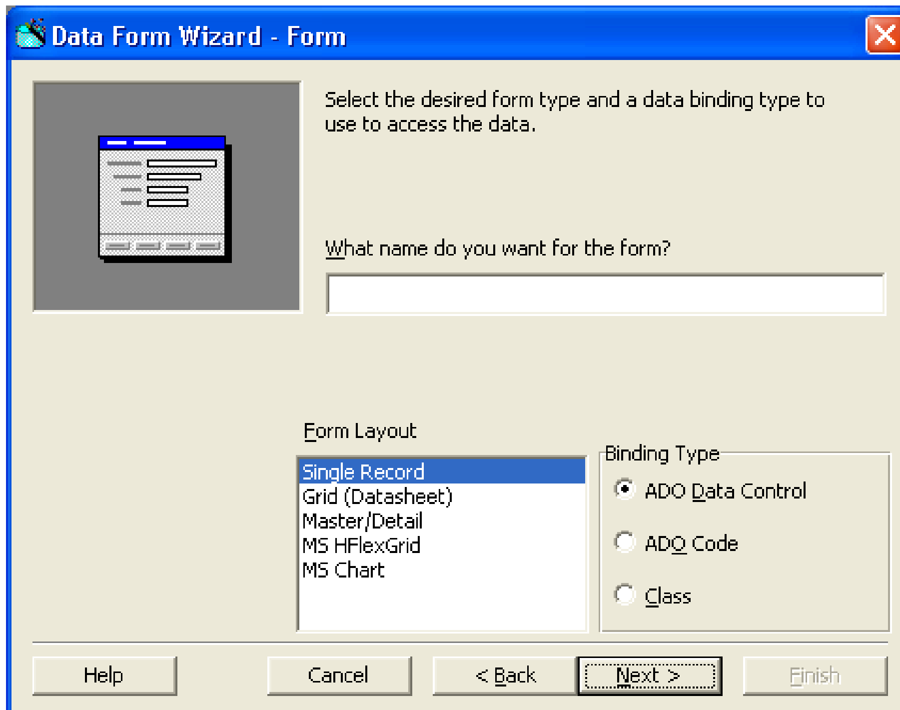


The grid shows more information at one time.

Data Form Wizard

Visual Basic contains a special Data Form Wizard that lets you design advanced forms that you can insert into your applications. These forms offer unique DAO and RDO (Remote Data Objects that might appear on a network) capabilities that you can select by answering the wizard's dialog box questions. The Data Form Wizard is an add-in program that you have to set up before you can use it. (You must also have the Professional or Enterprise VB editions before you can add the Data Form Wizard.)
Perform these steps to add the Data Form Wizard to your Add-Ins menu:

1. From the Add-Ins menu, choose Add-In Manager to display the list of add-ins you can add to your Visual Basic development environment.
2. Select the Data Form Wizard and click OK. When you open theAdd-Ins menu again, the Data Form Wizard appears on the menu.
3. Choose Add-Ins, Data Form Wizard to start the wizard. After reading the opening dialog box, click Next to select a database type. You'll connect the Microsoft Access Biblio.mdb database to the wizard, so keep Access selected and click Next again.
4. Select the Biblio.mdb database and click Next to display the Form dialog box in Figure. The three options determine the style of the form you want the wizard to generate.



Select the kind of form you want the wizard to create from the data.

5. The Single Record style displays records one form at a time and is most useful for adding new records to the database from the application. Grid presents the grid view (called the datasheet view) of the data in a worksheet format like you saw in Figure. The Master/Detail view provides an interesting combination for the data display. As you click each option, the dialog box updates its icon to show you what the resulting form will look like.
6. Select Master/Detail and click Next. The next dialog box determines which table and fields you want to see in the resulting application's form. Select the Publishers table and then move the following fields to the Selected Fields list: PubID, Company Name, and State.
7. You've entered the Master section of the form, and you must now click Next to move to the next dialog box and enter the Detail section of the form. The Master section will show one selected record, and the Detail section will list the multiple records beneath the Master record.
8. Select the Titles table for the Detail section. A title list will appear beneath the publisher shown in the upper half of the form. Send these fields to the Selected Fields list: Title, Year Published, ISBN, and PubID.
9. Select the Titles field in the Column to Sort By list box. The form will display the titles in alphabetical order by title and not in the table's actual order, which may be different. Click Next to move to the next dialog box.
10. The Record Source Relation dialog box lets you connect the Master section to the Details section by selecting a common field. Click the PubID field in each column, and then click Finish to complete the wizard and generate the project.

The Data Form Wizard generates only the form, not a complete project. Therefore, you can add the form into whatever project that needs the form. To try the form, choose Properties from the Project menu and change Startup Object to frmTitles. When you run the application, you'll see the resulting Master/Detail form. As you click through the records, publishers with multiple titles in the database will appear as shown in Figure. Although you

normally wouldn't show the PubID field (because the field will contain the same value for each record), you can see now how the PubID field connects the Master record to the Detail record.

Publishers

PubID: 15

Company Name: CAMBRIDGE UNIV PR

State: MA

Title	Year Published	ISBN	PubID
Ada	1990	0-5213942-2-4	15
Ada : Experiences and Prospects : Proceedings of the Ad	1990	0-5213952-2-4	15
Australian Popular Culture (Australian Cultural Studies	1994	0-5214666-7-9	15
Basic Simple Type Theory (Cambridge Tracts in Theoretic	1995	0-5214651-8-4	15
Beowulf : An Introduction	1959	0-5210461-5-7	15
C by Example (Cambridge Computer Science Texts, No 2S	1994	0-5214502-3-3	15
C by Example (Cambridge Computer Science Texts, No 2S	1994	0-5214565-0-9	15
Computational Geometry in C	1994	0-5214403-4-3	15
Computational Geometry in C	1994	0-5214459-2-2	15
Concurrency in Ada	1995	0-5214147-1-7	15
Database Design : A Classified and Annotated Bibliograph	1986	0-5213112-3-3	15

Add Delete Refresh Update Close

Record: 15

The Master and Detail record are synchronized.
After designing the form, you can save it and add it to future projects that need access to the data.

WEEK 14

GENERATE REPORT USING DATA REPORT IN VISUAL BASIC

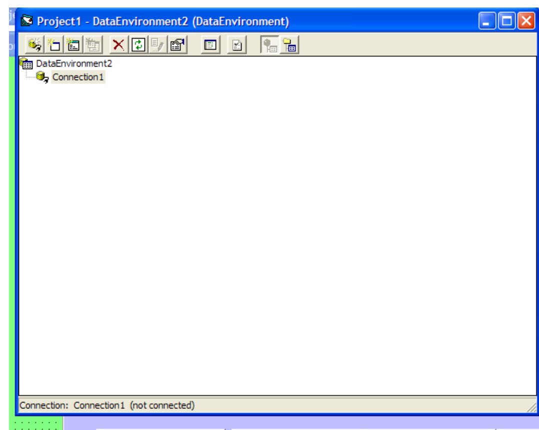
During this week you will learn:

. Add data environment
. Create the logical view of your report
. Add data report
. Create Report format
. Print report

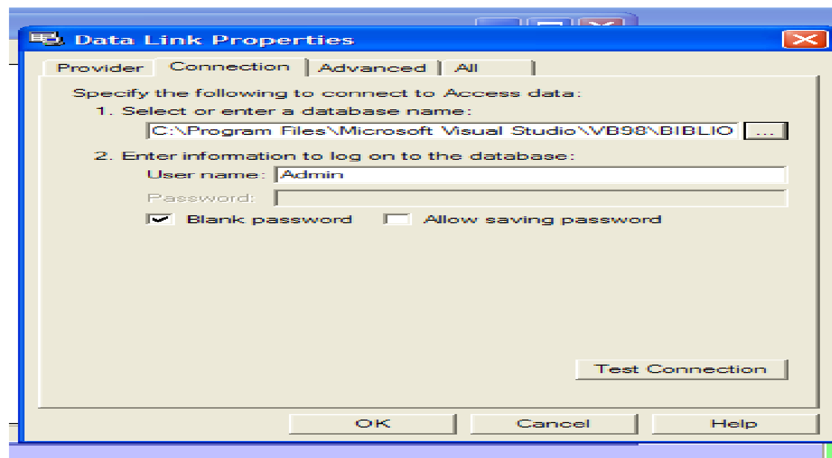
To generate the report from your database follow the steps below

1. On the Menu bar click project
2. Select Add data environment

   The Data environment window is displayed as shown below:



3. Right click on connection 1
4. Select properties in the Data link pup-up menu windows displayed
5. Specify the provide or the Database engine. By selecting 3.51 OLE DB provide for the Database Structure created within vb environment or using MS Access '97
6. Click next, this takes you to connector Data link Properties windows



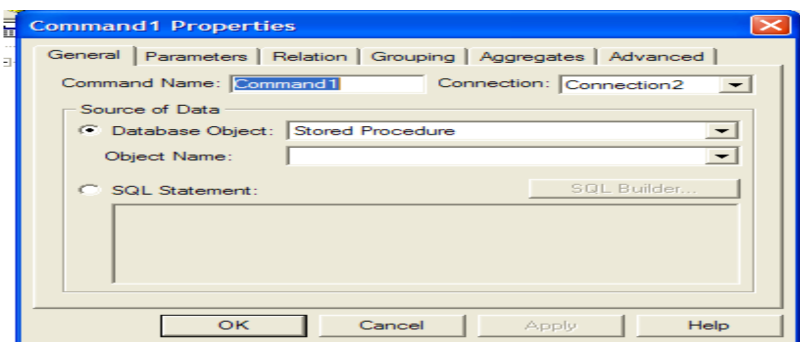
7. Select or Enter a database name

8. Click Test connector, to be sure you can successfully connected

9. Click OK. This takes you back to the Data environment window

10. Create the logical view of your report i.e. Table or fields that you need in your database. To do this;

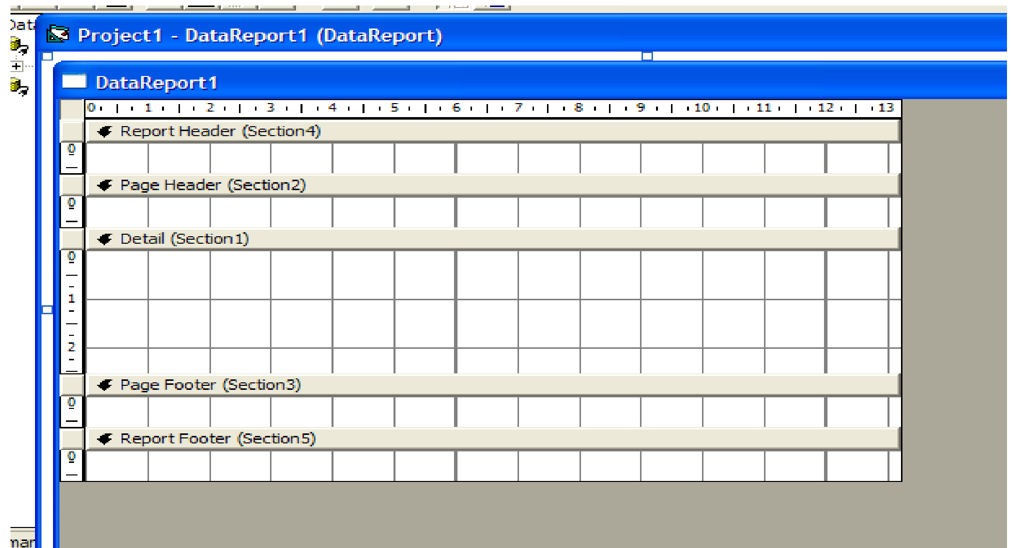
    - Click on connector
    - Click on Add command Icon on the Data environment tool bar, Command 1 is created
    - Right Click the Command 1 created
    - Click on property from the PUP-UP menu displayed
    - Command properties window is displayed



    - Specify the command name e.g. (mystock)
    - Select Database Object
    - Select Table
    - Select Object name
    - Select mytable
    - Click apply
    - Click OK

11. Now create the Report format as follows:

- Click project
    - Click Add Data Report, the Data Project Properties is displayed



    - Click Report Header for the Heading that appears once in the Report
    - Click Page Header for the Column Title
    - Click Detail Section for Fields in the Report
    - Go To Properties window, select Data Source
    - Select data source, eg Destock
    - Click Data Member, select Command Object for the Data Members i.e. (mdstock)
    - Right click any area within the Data report window
    - From the PUP-UP displayed, select retrieve structure, to link our report to the command structure created earlier in the Data environment
    - Click Yes
    - Select the band (i.e. the Page Header)
    - Double Click label on the tool bar
    - Set the properties e.g.

      Change Caption to "STOCK REPORT".
- Click on Details report
- Double Click the Report Text box

- Click Unbound

a. Click Data Member in Properties Text box

    b. Select (cmdstock)

    c. Select Data field Itemnum

12. Give your Data report a name e.g. drpstock
13. Set caption for the windows
14. Set window state to Maximise
15. Save the project to Update all the work
16. Then Go back to the interface form
17. Double click the Command Report
18. Type the program statements as shown below

```
Private Sub Cmd_report_Click()
    drpStoc.ref
    drpStock.Show
End Sub
```

- Run the Program, then Click on Report Command, the report is displayed
- Click Print Icon on the report to Print the report on paper

WEEK 15:

CREATING MENUS

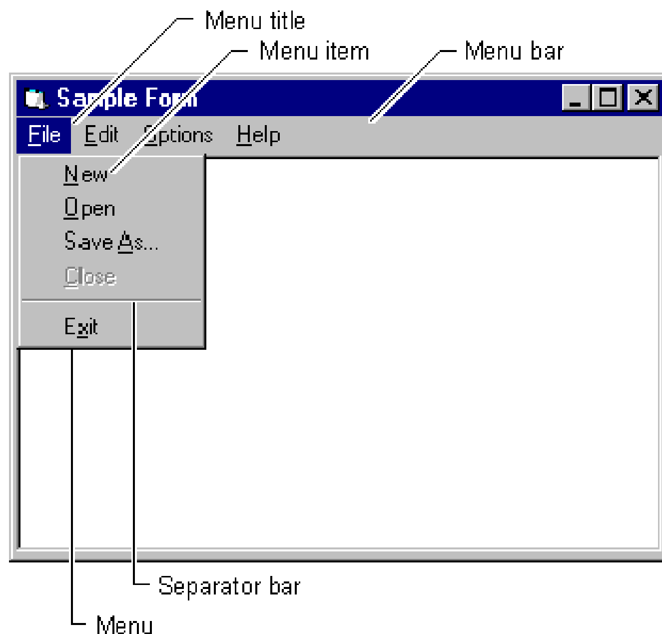
During this week you will learn :

- Menu Basics
- Menu Control
- Menu Editor Dialog Box
- Creating Menus with the Menu Editor
- Writing Code for Menu Controls
- MsgBox Function
- InputBox Function

Visual Basic makes creating and placing menu bar items into your application as easy as pushing command buttons and typing a few keystrokes. The Menu Design window contains menu description tools that enable you to create the application's menu bar, menu commands, and shortcut access keys to all of your applications.

Menu Basics

If you want your application to provide a set of commands to users, menus offer a convenient and consistent way to group commands and an easy way for users to access them. Figure Illustrates the elements of a menu interface on an untitled form.



The menu bar appears immediately below the title bar on the form and contains one or more menu titles. When you click a menu title (such as File), a menu containing a list of menu items drops down. Menu items can include commands (such as New and Exit), separator bars, and submenu titles. Each menu item the user sees corresponds to a menu control you define in the Menu Editor (described later in this chapter).

To make your application easier to use, you should group menu items according to their function. In Figure above, for example, the file-related commands New, Open, and Save As… are all found on the File menu.
Some menu items perform an action directly; for example, the Exit menu item on the File menu closes the application. Other menu items display a dialog box — a window that requires the user to supply information needed by the application to perform the action. These menu items should be followed by an ellipsis (…). For example, when you choose Save As… from the File menu, the Save File As dialog box appears.
A menu control is an object; like other objects it has properties that can be used to define its appearance and behavior. You can set the Caption property, the Enabled and Visible properties, the Checked property, and others at design time or at run time. Menu controls contain only one event, the Click event, which is invoked when the menu control is selected with the mouse or using the keyboard.

Menu Control

A **Menu** control displays a custom menu for your application. A menu can include commands, submenus, and separator bars. Each menu you create can have up to four levels of submenus.

Syntax

Menu

Remarks

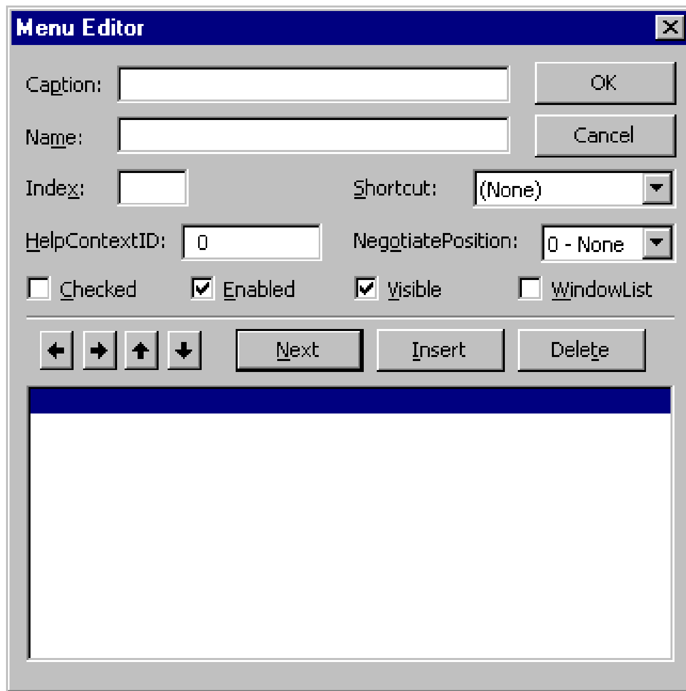
To create a **Menu** control, use the Menu Editor. Enter the name of the **Menu** control in the Caption box. To create a separator bar, enter a single hyphen (-) in the Caption box. To display a check mark to the left of a menu item, select the Checked box.
While you can set some **Menu** control properties using the Menu Editor, all **Menu** control properties are displayed in the Properties window. To display the properties of a **Menu** control, select the menu name in the Objects list at the top of the Properties window.
When you create an MDI application, the menu bar on the MDI child form replaces the menu bar on the **MDIForm** object when the child form is active.

Menu Editor Command (Tools Menu)

Displays the Menu Editor dialog box.
Use the Menu Editor command to create custom menus for your application ,and to define some of their properties.

Toolbar shortcut: . Keyboard shortcut: CTRL+E.

Menu Editor Dialog Box



Allows you to create custom menus for your application and to define their properties.

Dialog Box Options

Caption

Allows you to enter the menu or command name that you want to appear on your menu bar or in a menu.
If you want to create a separator bar in your menu, type a single hyphen (-) in the Caption box.
To give the user keyboard access to a menu item, insert an ampersand (&) before a letter. At run time, this letter is underlined (the ampersand is not visible), and the user can access the menu or command by pressing ALT and the letter. If you need an ampersand to show in the menu, put two consecutive ampersands in the caption.

Name

Allows you to enter a control name for the menu item. A control name is an identifier used only to access the menu item in code; it doesn't appear in a menu.

Index

Allows you to assign a numeric value that determines the control's position within a control array. This position isn't related to the screen position.

Shortcut

Allows you to select a shortcut key for each command.

HelpContextID

Allows you to assign a unique numeric value for the context ID. This value is used to find the appropriate Help topic in the Help file identified by the **HelpFile** property.

NegotiatePosition

Allows you to select the menu's **NegotiatePosition** property. This property determines whether and how the menu appears in a container form.

Checked

Allows you to have a check mark appear initially at the left of a menu item. It is generally used to indicate whether a toggle option is turned on or off.

Enabled

Allows you to select whether you want the menu item to respond to events, or clear if you want the item to be unavailable and appear dimmed.

Visible

Allows you to have the menu item appear on the menu.

WindowList

Determines if the menu control contains a list of open MDI child forms in an MDI application.

Right Arrow

Moves the selected menu down one level each time you click it. You can create up to four levels of submenus.

Left Arrow

Moves the selected menu up one level each time you click it. You can create up to four levels of submenus.

Up Arrow

Moves the selected menu item up one position within the same menu level each time you click it.

Down Arrow

Moves the selected menu item down one position within the same menu level each time you click it.

Menu List

A list box that displays a hierarchical list of menu items. Submenu items are indented to indicate their hierarchical position or level.

Next

Moves selection to the next line.

Insert

Inserts a line in the list box above the currently selected line

Delete

Deletes the currently selected line.

OK

Closes the Menu Editor and applies all changes to the last form you selected. The menu is available at design time, but selecting a menu at design time opens the Code window for that menu's Click event rather than executing any event code.

Cancel

Closes the Menu Editor and cancels all changes.

Creating Menus with the Menu Editor

You can use the Menu Editor to create new menus and menu bars, add new commands to existing menus, replace existing menu commands with your own commands, and change and delete existing menus and menu bars.

To display the Menu Editor

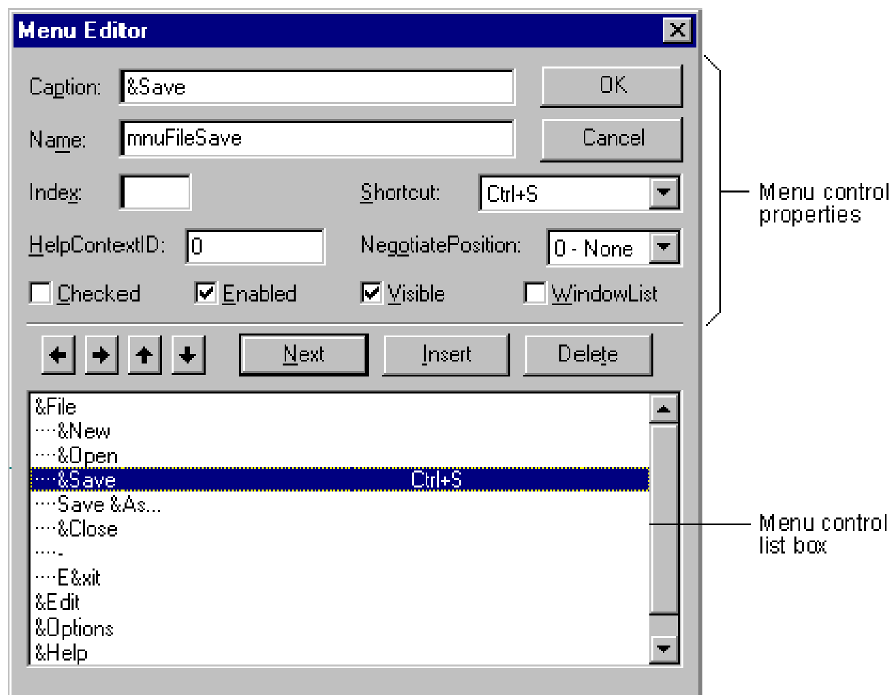
- From the **Tools** menu, choose **Menu Editor**.

  –or–

  Click the **Menu Editor** button on the toolbar.

This opens the Menu Editor, shown in Figure.

The Menu Editor



While most menu control properties can be set using the Menu Editor, all menu properties are available in the Properties window. The two most important properties for menu controls are:

- Name — This is the name you use to reference the menu control from code.

- Caption — This is the text that appears on the control.

Other properties in the Menu Editor, including Index, Checked, and NegotiatePosition, are described later in this chapter.

Using the List Box in the Menu Editor

The menu control list box (the lower portion of the Menu Editor) lists all the menu controls for the current form. When you type a menu item in the Caption text box, that item also appears in the menu control list box. Selecting an existing menu control from the list box allows you to edit the properties for that control.
For example, Figure shows the menu controls for a File menu in a typical application. The position of the menu control in the menu control list box determines whether the control is a menu title, menu item, submenu title, or submenu item:

- A menu control that appears flush left in the list box is displayed on the menu bar as a menu title.
- A menu control that is indented once in the list box is displayed on the menu when the user clicks the preceding menu title.
- An indented menu control followed by menu controls that are further indented becomes a submenu title. Menu controls indented below the submenu title become items of that submenu.
- A menu control with a hyphen (-) as its Caption property setting appears as a separator bar. A separator bar divides menu items into logical groups.

Note A menu control cannot be a separator bar if it is a menu title, has submenu items, is checked or disabled, or has a shortcut key.
To create menu controls in the Menu Editor

1. Select the form.
2. From the **Tools** menu, choose **Menu Editor**.

   –or–

   Click the **Menu Editor** button on the toolbar.
3. In the **Caption** text box, type the text for the first menu title that you want to appear on the menu bar. Also, place an ampersand (&) before the letter you want to be the access key for that menu item. This letter will automatically be underlined in the menu.

   The menu title text is displayed in the menu control list box.
4. In the **Name** text box, type the name that you will use to refer to the menu control in code. See "Menu Title and Naming Guidelines" later in this chapter.
5. Click the left arrow or right arrow buttons to change the indentation level of the control.

6. Set other properties for the control, if you choose. You can do this in the Menu Editor or later, in the Properties window.

7. Choose **Next** to create another menu control.

   –or–

   Click **Insert** to add a menu control between existing controls.

   You can also click the up arrow and down arrow buttons to move the control among the existing menu controls.

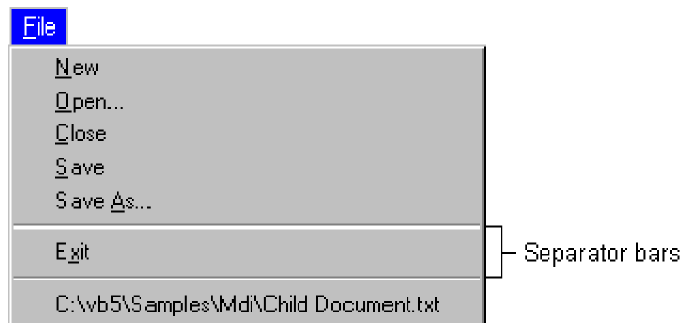
8. Choose **OK** to close the Menu Editor when you have created all the menu controls for that form.

   The menu titles you create are displayed on the form. At design time, click a menu title to drop down its corresponding menu items.

Separating Menu Items

A separator bar is displayed as a horizontal line between items on a menu. On a menu with many items, you can use a separator bar to divide items into logical groups. For example, the File menu in Visual Basic uses separator bars to divide its menu items into three groups, as shown in Figure

Separator bars



To create a separator bar in the Menu Editor

1. If you are adding a separator bar to an existing menu, choose **Insert** to insert a menu control between the menu items you want to separate.

2. If necessary, click the right arrow button to indent the new menu item to the same level as the menu items it will separate.

3. Type a hyphen (-) in the **Caption** text box.

4. Set the **Name** property.

5. Choose **OK** to close the Menu Editor.

Note Although separator bars are created as menu controls, they do not respond to the Click event, and users cannot choose them.

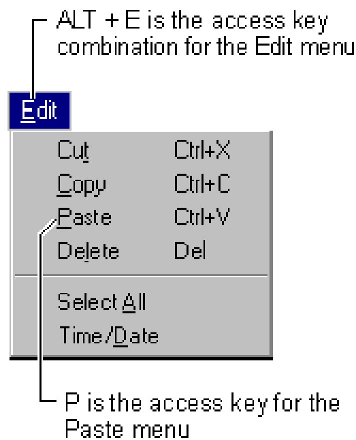
Assigning Access Keys and Shortcut Keys

You can improve keyboard access to menu commands by defining access keys and shortcut keys.

Access Keys

Access keys allow the user to open a menu by pressing the ALT key and typing a designated letter. Once a menu is open, the user can choose a control by pressing the letter (the access key) assigned to it. For example, ALT+E might open the Edit menu, and P might select the Paste menu item. An access-key assignment appears as an underlined letter in the menu control's caption, as shown in Figure

Access keys



To assign an access key to a menu control in the Menu Editor

1. Select the menu item to which you want to assign an access key.
2. In the **Caption** box, type an ampersand (&) immediately in front of the letter you want to be the access key.

For example, if the Edit menu shown in Figure is open, the following Caption property settings respond to the corresponding keys.

Menu control caption	Caption property	Access keys
Cut	Cu&t	t
Copy	C&opy	o
Paste Delete Select All Time/Date	&Paste De&lete Select &All Time/&Date	p l a d

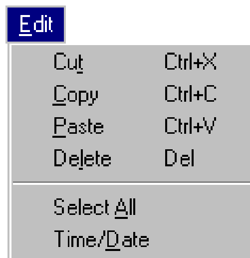
Note Do not use duplicate access keys on menus. If you use the same access key for more than one menu item, the key will not work. For example, if C is the access key for both Cut

and Copy, when you select the Edit menu and press C, the Copy command will be selected, but the application will not carry out the command until the user presses ENTER. The Cut command will not be selected at all.

Shortcut Keys

Shortcut keys run a menu item immediately when pressed. Frequently used menu items may be assigned a keyboard shortcut, which provides a single-step method of keyboard access, rather than a three-step method of pressing ALT, a menu title access character, and then a menu item access character. Shortcut key assignments include function key and control key combinations, such as CTRL+F1 or CTRL+A. They appear on the menu to the right of the corresponding menu item, as shown in Figure .

Shortcut keys



To assign a shortcut key to a menu item

1. Open the **Menu Editor**.
2. Select the menu item.
3. Select a function key or key combination in the **Shortcut** combo box.

   To remove a shortcut key assignment, choose "(none)" from the top of the list.

Note Shortcut keys appear automatically on the menu; therefore, you do not have to enter CTRL+key in the Caption box of the Menu Editor.

Writing Code for Menu Controls

When the user chooses a menu control, a Click event occurs. You need to write a Click event procedure in code for each menu control. All menu controls except separator bars (and disabled or invisible menu controls) recognize the Click event.
The code that you write in a menu event procedure is no different than that which you would write in any other control's event procedure. For example, the code in a File, Close menu's Click event might look like this:

```
Sub mnuFileClose_Click()
   Unload Me
End Sub
```

Visual Basic displays a menu automatically when the menu title is chosen; therefore, it is not necessary to write code for a menu title's Click event procedure unless you want to perform another action, such as disabling certain menu items each time the menu is displayed.

Note At design time, the menus you create are displayed on the form when you close the Menu Editor. Choosing a menu item on the form displays the Click event procedure for that menu control.

In Windows-based applications, dialog boxes are used to prompt the user for data needed by the application to continue or to display information to the user.

MsgBox Function

Displays a message in a dialog box, waits for the user to click a button, and returns an **Integer** indicating which button the user clicked.

Syntax

MsgBox(prompt[, **buttons**] [, **title**] [, **helpfile**, **context**])

The **MsgBox** function syntax has these named arguments:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of **prompt** is approximately 1024 characters, depending on the width of the characters used. If **prompt** consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr**(10)), or carriage return – linefeed character combination (**Chr**(13) & **Chr**(10)) between each line.
buttons	Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for **buttons** is 0.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit **title**, the application name is placed in the title bar.
helpfile	Optional. String expression 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	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If **context** is provided, **helpfile** must also be provided.

Settings

The **buttons** argument settings are:

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 default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	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 systems

The first group of values (0–5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the **buttons** argument, use only one number from each group.

Note These constants are specified by Visual Basic for Applications. As a result, the names can be used anywhere in your code in place of the actual values.

Return Values

Constant	Value	Description
vbOK	1	**OK**
vbCancel	2	**Cancel**
vbAbort	3	**Abort**
vbRetry	4	**Retry**
vbIgnore	5	**Ignore**
vbYes	6	**Yes**
vbNo	7	**No**

Remarks

When both **helpfile** and **context** are provided, the user can press F1 to view the Help topic corresponding to the **context**. Some host applications, for example, Microsoft Excel, also automatically add a **Help** button to the dialog box.
If the dialog box displays a **Cancel** button, pressing the ESC key has the same effect as clicking **Cancel**. If the dialog box contains a **Help** button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

Example :

This example uses the **MsgBox** function to display a critical-error message in a dialog box with Yes and No buttons. The No button is specified as the default response. The value returned by the **MsgBox** function depends on the button chosen by the user. This example assumes that `DEMO.HLP` is a Help file that contains a topic with a Help context number equal to `1000`.

```
Dim Msg, Style, Title, Help, Ctxt, Response, MyString
Msg = "Do you want to continue ?"   ' Define message.
Style = vbYesNo + vbCritical + vbDefaultButton2   ' Define buttons.
Title = "MsgBox Demonstration"   ' Define title.
Help = "DEMO.HLP"   ' Define Help file.
Ctxt = 1000   ' Define topic
      ' context.
      ' Display message.
Response = MsgBox(Msg, Style, Title, Help, Ctxt)
If Response = vbYes Then   ' User chose Yes.
   MyString = "Yes"   ' Perform some action.
Else   ' User chose No.
   MyString = "No"   ' Perform some action.
End If
```

InputBox Function

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns a String containing the contents of the text box.

Syntax

InputBox(prompt[, **title**] [, **default**] [, **xpos**] [, **ypos**] [, **helpfile**, **context**])

The **InputBox** function syntax has these named arguments:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of **prompt** is approximately 1024 characters, depending on the width of the characters used. If **prompt** consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13)), a linefeed character (**Chr(**10)), or carriage return–linefeed character combination (**Chr(**13) & **Chr(**10)) between each line.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit **title**, the application name is placed in the title bar.
default	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit **default**, the text box is displayed empty.
xpos	Optional. Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If **xpos** is omitted, the dialog box is horizontally centered.
ypos	Optional. Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If **ypos** is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.
helpfile	Optional. String expression 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	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If **context** is provided, **helpfile** must also be provided.

Remarks

When both **helpfile** and **context** are provided, the user can press F1 to view the Help topic corresponding to the **context**. Some host applications, for example, Microsoft Excel, also automatically add a **Help** button to the dialog box. If the user clicks **OK** or presses ENTER , the **InputBox** function returns whatever is in the text box. If the user clicks **Cancel**, the function returns a zero-length string ("").

Example :

This example shows various ways to use the **InputBox** function to prompt the user to enter a value. If the x and y positions are omitted, the dialog box is automatically centered for the respective axes. The variable `MyValue` contains the value entered by the user if the user clicks **OK** or presses the ENTER key . If the user clicks **Cancel**, a zero-length string is returned.

```
Dim Message, Title, Default, MyValue
Message = "Enter a value between 1 and 3"   ' Set prompt.
Title = "InputBox Demo"   ' Set title.
Default = "1"   ' Set default.
' Display message, title, and default value.
MyValue = InputBox(Message, Title, Default)

' Use Helpfile and context. The Help button is added automatically.
MyValue = InputBox(Message, Title, , , , "DEMO.HLP", 10)

' Display dialog box at position 100, 100.
MyValue = InputBox(Message, Title, Default, 100, 100)
```