



**NEAR EAST UNIVERSITY**

**Faculty of Engineering**

**Department of Computer Engineering**

**Car Service Garage Program  
With Visual Basic**

**Graduation Project  
Com-400**

**Student: Hasan Onbaşı(20033591)**

**Supervisor: Dr Umit İlhan**

**NICOSIA-2008**



## ACKNOWLEDGEMENTS

First,I would like to thank my supervisor Dr UMIT ILHAN for his support and encouragement during this graduation project Second ,I would like to express my gratitude to Near East University for the scholarship that made the work possible. Third , I thank my familiy for their constant encaungement and support during the preparation of this project. Finally.I would also like to thank all my friends For their advice and support



## ABSTRACT

This project including, base topics of visual basic 6.0,chapter1(The visual basic language)

Chapter2(Exploring the visual basic tollbok), chapter3(More Exploration of the Visual Basic Toolbox),chapter4(Error-Handling, Debugging and File Input/Output) , chapter5(Database Access Management),chapter6(SQL) finally chapter7(Car Service Garrage program with visual basic) This program(project) including 3 part. At program used Database Access, SQL, and base commands of visual basic.

## **TABLE OF CONTENTS**

### **Chapter.1 The Visual Basic Language**

1.1>About Visual Basic	1
1.2.Visual Basic Statements and Expressions	2
1.3.Visual Basic Operators	3
1.4.Visual Basic Functions	4
1.5.String Functions	5
1.6Rnd (Random Number) Function	6
Example : Savings Account	
1.7Visual Basic Symbolic Constants	8
1.8Defining Your Own Constants	11
1.9Visual Basic Branching - If Statements	12
1.10Key Trapping	
Example : Savings Account - Key Trapping	13
1.11Select Case - Another Way to Branch	15
1.12The GoTo Statement	16
1.13Visual Basic Looping	18
1.14Visual Basic Counting	19
Example : Savings Account - Decisions	

### **Chapter.2 Exploring the Visual Basic Toolbox**

2.1.The Message Box	20
2.2.Object Methods	22
2.3.The Form Object	23
2.4.Command Buttons	24
2.5.Label Boxes	25
2.6.Text Boxes	25
Example : Password Validation	28
2.7.Check Boxes	28
2.8.Option Buttons	29
2.9.Arrays	29
2.10.Frames	31
Example 3-2: Pizza Order	
2.11List Boxes	36
2.12.Combo Boxes	37

### **Chapter.3 More Exploration of the Visual Basic Toolbox**

3.1.Display Layers	41
3.2.Line Tool	43
3.3.Shape Tool	43
3.4.Horizontal and Vertical Scroll Bars	44
Example : Temperature Conversion	
3.5.Image Boxes	47
3.6.Quick Example: Picture and Image Boxes	48
3.7.Drive List Box	49
3.8.Directory List Box	49
3.9.File List Box	50
3.10.Synchronizing the Drive, Directory, and File List Boxes	51
Example : Image Viewer	
3.11.Common Dialog Boxes	55
3.12.Open Common Dialog Box	56
3.13.Save As Common Dialog Box	59
Quick Example: The Save As Dialog Box	

### **Chapter.4 Error-Handling, Debugging and File Input/Output**

4.1.ErrorTypes	60
4.2.Run-Time Error Trapping and Handling	61
4.3.General Error Handling Procedure	63
Example - Simple Error Trapping	
4.4.Debugging Visual Basic Programs	66
Example - Debugging Example.	
4.5.Using the Debugging Tools	68
4.6.Debugging Strategies	71
4.7.Sequential Files	72
4.8.Sequential File Output (Variables)	73
4.9.Sequential File Input (Variables)	74
4.10.Writing and Reading Text Using Sequential Files	75
4.11.Random Access Files	77
4.12.User-Defined Variables	79
4.13.Writing and Reading Random Access Files	80
4.14.Using the Open and Save Common Dialog Boxes	81

## **Chapter.5 Database Access Management**

5.1.Database Structure and Terminology	84
5.2.ADO Data Control (ADODC)	86
5.3.Data Links	87
5.4.Assigning Tables	88
5.5.Bound Data Tools	89
Example - Accessing the Books Database	
5.6.Creating a Virtual Table	91
5.7.Finding Specific Records	92

## **Chapter.6 SQL**

6.1.Select statement	94
6.2.Conditional selection	95
6.3.Join	96
6.4.Function	98
6.5.Deleting Data	99
6.6.Updating data	99

## **Chapter.7 Car Service Garrage Program with visual basic**

7.1.About project	100
7.2.Advantages of program	101
7.3.step by step program	102
7.4.conclusion	112

CONCLUSION	113
REFERENCES	114



## Introduction

Visual Basic is a highly popular language in the commercial world because it allows for the rapid development of Windows based programs. VB is particularly strong at creating front ends for databases. This can be done in amazing time through the use of wizards. This page does not cover all aspects of VB, it does not show how to do the basics like layout a form, neither does it cover all the built in functions, as there is already plenty of help provided for these, and a lot of it is self-evident. A more limited version of Visual Basic is also included in several other Microsoft Applications such as MS Access. Most of the information here applies to that version. There is also VB Script for creating web pages. Much of the information on this page applies, but VB Script only has one basic data type - the Variant type. This project consists of introduction, 7 chapters.

Chapter 1 describes the The Visual Basic Language

Chapter 2 describes the Exploring the Visual Basic Toolbox

Chapter 3 describes the More Exploration of the Visual Basic Toolbox

Chapter 4 describes the Error-Handling, Debugging and File Input/Output

Chapter 5 describes the Database Access Management

Chapter 6 describes the SQL with visual basic

Chapter 7 describes the Car Service Garage Program with visual basic

## 1.1 About Visual Basic

Visual Basic (VB) is a third-generation event driven programming language and associated development environment (IDE) from Microsoft for its COM programming model.<sup>[1]</sup> Visual Basic was derived from BASIC and enables the rapid application development (RAD) of graphical user interface (GUI) applications, access to databases using DAO, RDO, or ADO, and creation of ActiveX controls and objects. Scripting languages such as VBA and VBScript are syntactically similar to Visual Basic, but perform differently.<sup>[2]</sup>

A programmer can put together an application using the components provided with Visual Basic itself. Programs written in Visual Basic can also use the Windows API, but doing so requires external function declarations.

### A Brief History of Basic

Language developed in early 1960's at Dartmouth College:

**B** (eginner's)  
**A** (All-Purpose)  
**S** (Symbolic)  
**I** (Instruction)  
**C** (Code)

Answer to complicated programming languages (FORTRAN, Algol, Cobol ...). First timeshare language.



## 1.2. Visual Basic Statements and Expressions

The simplest statement is the **assignment** statement. It consists of a variable name, followed by the assignment operator (=), followed by some sort of **expression**. The variable (or property) on the left hand side of the assignment operator is **replaced** by the value of the expression on the right hand side of the operator.

**Examples:**

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

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

```
StartTime = Now : EndTime = StartTime + 10
```

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

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

```
Months = Log(Final * IntRate / Deposit + 1) _  
/ Log(1 + IntRate)
```

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

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

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

### 1.3. Visual Basic Operators

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

Operator	Operation
<sup>^</sup>	Exponentiation
* /	Multiplication and division
\	Integer division (truncates decimal portion)
Mod	Modulus
+ -	Addition and subtraction

**Parentheses** around expressions can change precedence.

To **concatenate** two strings, use the **&** symbol or the **+** symbol:

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

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

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

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

The result of a comparison operation is a Boolean value (**True** or **False**).

We will use three **logical** operators

<b>Operator</b>	<b>Operation</b>
Not	Logical not
And	Logical and
Or	Logical or

The **Not** operator simply negates an operand. It is very useful for 'toggling' Boolean variables.

The **And** operator returns a True if both operands are True. Else, it returns a False.

The **Or** operator returns a True if either of its operands is True, else it returns a False.

Logical operators follow arithmetic operators in precedence.

#### **1.4. Visual Basic Functions**

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

<b>Function</b>	<b>Value Returned</b>
Abs	Absolute value of a number
Asc	ASCII or ANSI code of a character
Chr	Character corresponding to a given ASCII or ANSI code
Cos	Cosine of an angle
Format	Date or number converted to a text string
Instr	Locates a substring in another string
Left	Selected left side of a text string
Len	Number of characters in a text string
Mid	Selected portion of a text string
Now	Current time and date
Right	Selected right end of a text string
Rnd	Random number
Sin	Sine of an angle
Sqr	Square root of a number
Str	Number converted to a text string
Timer	Number of seconds elapsed since midnight
Trim	Removes leading and trailing spaces from string
Val	Numeric value of a given text string

## 1.5.String Functions

Visual Basic offers a powerful set of functions to work with string type variables, which are very important in Visual Basic. The **Caption** property of the label control and the **Text** property of the text box control are string types. You will find you are constantly converting string types to numeric data types to do some math and then converting back to display the information.

To convert a string type to a numeric value, use the **Val** function. As an example, to convert the Text property of a text box control named txtExample to a number, use:

```
Val(txtExample.Text)
```

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

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

```
MyNumber = 3.14159  
txtExample.Text = Str(MyNumber)
```

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

To find a substring within a string variable, use the **Instr** function. Three arguments are used: starting position in String1 (optional), **String1** (the variable), **String2** (the substring to find). The function will return the location of the first character of the substring (it will return 0 if the substring is not found). For our example:

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

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



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

**Asc("A")**

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

**Chr(48)**

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

## 1.6.Rnd (Random Number) Function

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

$$I = \text{Int}((\text{Imax} - \text{Imin} + 1) * \text{Rnd}) + \text{Imin}$$

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

### Randomize Seed

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

### Randomize

Place the above statement in the **Form\_Load** event procedure.

### Examples:

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

$$\text{NumberSpots} = \text{Int}(6 * \text{Rnd}) + 1$$

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

$$\text{Number} = \text{Int}(101 * \text{Rnd}) + 100$$

## Example 2-1

### 1.7.Savings Account

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

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

where

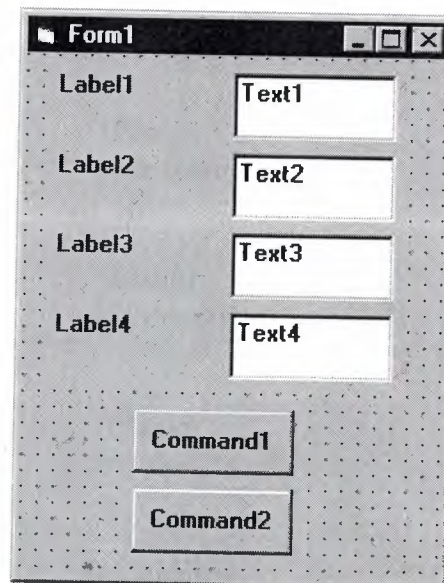
F - Final amount

D - Monthly deposit amount

I - Monthly interest rate

M - Number of months

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



The screenshot shows a standard Windows application window titled "Form1". Inside the window, there is a grid of controls. On the left side, there are four labels: "Label1", "Label2", "Label3", and "Label4", arranged vertically. To the right of each label is a corresponding text box: "Text1", "Text2", "Text3", and "Text4", also arranged vertically. Below these text boxes, at the bottom of the form, are two command buttons: "Command1" and "Command2", stacked vertically. The form has a dotted grid background and standard Windows window controls (minimize, maximize, close) in the title bar.

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

**Form1:**

BorderStyle	1-Fixed Single
Caption	Savings Account
Name	frmSavings

**Label1:**

Caption	Monthly Deposit
---------	-----------------

**Label2:**

Caption	Yearly Interest
---------	-----------------

**Label3:**

Caption	Number of Months
---------	------------------

**Label4:**

Caption	Final Balance
---------	---------------

**Text1:**

Text	[Blank]
Name	txtDeposit

**Text2:**

Text	[Blank]
Name	txtInterest

**Text3:**

Text	[Blank]
Name	txtMonths

**Text4:**

Text	[Blank]
Name	txtFinal

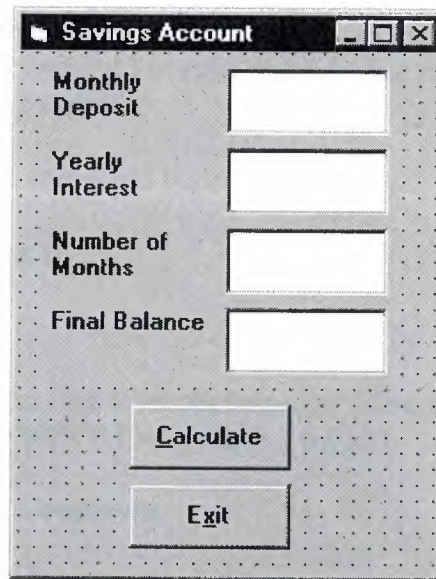
**Command1:**

Caption	&Calculate
Name	cmdCalculate

**Command2:**

Caption	E&xit
Name	cmdExit

Now, your form should look like this:



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

**Option Explicit**

**Dim Deposit As Single**

**Dim Interest As Single**

**Dim Months As Single**

**Dim Final As Single**

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

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

**Private Sub cmdCalculate\_Click ()**

**Dim IntRate As Single**

**'Read values from text boxes**

**Deposit = Val(txtDeposit.Text)**

**Interest = Val(txtInterest.Text)**

**IntRate = Interest / 1200**

**Months = Val(txtMonths.Text)**

**'Compute final value and put in text box**

**Final = Deposit \* ((1 + IntRate) ^ Months - 1) / IntRate**

**txtFinal.Text = Format(Final, "#####0.00")**

**End Sub**



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

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

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

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

### 1.7. Visual Basic Symbolic Constants

Many times in Visual Basic, functions and objects require data arguments that affect their operation and return values you want to read and interpret. These arguments and values are constant numerical data and difficult to interpret based on just the numerical value. To make these constants more understandable, Visual Basic assigns names to the most widely used values - these are called **symbolic constants**. Appendix I lists many of these constants.

As an example, to set the background color of a form named **frmExample** to blue, we could type:

```
frmExample.BackColor = 0xFF0000
```

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

```
frmExample.BackColor = vbBlue
```

It is strongly suggested that the symbolic constants be used instead of the numeric values, when possible. You should agree that **vbBlue** means more than the value **0xFF0000** when selecting the background color in the above example. You do not need to do anything to define the symbolic constants - they are built into Visual Basic.

### 1.8. Defining Your Own Constants

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

```
Const PI = 3.14159
```

**User-defined constants** should be written in all upper case letters to distinguish them from variables. The scope of constants is established the same way a variables' scope is. That is, if defined within a procedure, they are local to the procedure. If defined in



the general declarations of a form, they are global to the form. To make constants global to an application, use the format:

**Global Const PI = 3.14159**

within the general declarations area of a module.

### 1.9. Visual Basic Branching - If Statements

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

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

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

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

**If/Then/End If** blocks to allow multiple statements:

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

In this case, if Balance - Check is less than zero, two lines of information are printed. Or, **If/Then/Else/End If** blocks:

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

Here, the same two lines are printed if you are overdrawn (Balance - Check < 0), but, if you are not overdrawn (**Else**), your new Balance is computed.

Or, we can add the **ElseIf** statement:

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

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

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

## 1.10.Key Trapping

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

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

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

What happens in this procedure is that every time a key is pressed in the corresponding text box, the ASCII code for the pressed key is passed to this procedure in the argument list (i.e. **KeyAscii**). If **KeyAscii** is an acceptable value, we would do nothing. However, if **KeyAscii** is not acceptable, we would set **KeyAscii** equal to zero and exit the procedure. Doing this has the same result of not pressing a key at all. ASCII values for all keys are available via the on-line help in Visual Basic. And some keys are also defined by symbolic constants. Where possible, we will use symbolic constants; else, we will use the ASCII values.

As an example, say we have a text box (named **txtExample**) and we only want to be able to enter upper case letters (ASCII codes 65 through 90, or, correspondingly, symbolic

constants **vbKeyA** through **vbKeyZ**). The key press procedure would look like (the **Beep** causes an audible tone if an incorrect key is pressed):

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

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

### Example 2-2

#### **1Savings Account - Key Trapping**

1. Note the acceptable ASCII codes are 48 through 57 (numbers), 46 (the decimal point), and 8 (the backspace key). In the code, we use symbolic constants for the numbers and backspace key. Such a constant does not exist for the decimal point, so we will define one with the following line in the **general declarations** area:

```
Const vbKeyDecPt = 46
```

2. Add the following code to the three procedures: **txtDeposit\_KeyPress**, **txtInterest\_KeyPress**, and **txtMonths\_KeyPress**.

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

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



**End Sub**

**Private Sub txtMonths\_KeyPress (KeyAscii As Integer)**

**'Only allow number keys, decimal point, or backspace**

**If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then**

**Exit Sub**

**Else**

**KeyAscii = 0**

**Beep**

**End If**

**End Sub**

(In the If statements above, note the word processor causes a line break where there really shouldn't be one. That is, there is no line break between the words **Or KeyAscii** and **= vbKeyDecPt**. One appears due to page margins. In all code in these notes, always look for such things.)

3. Rerun the application and test the key trapping performance. Save the application (**Example2-2** in the **LearnVB6/VB Code/Class 2** folder).

## **1.11.Select Case - Another Way to Branch**

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

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

**If Age = 5 Then**

**Category = "Five Year Old"**

**ElseIf Age >= 13 and Age <= 19 Then**

**Category = "Teenager"**

**ElseIf (Age >= 20 and Age <= 35) Or Age = 50 Or (Age >= 60 and Age <= 65) Then**

**Category = "Special Adult"**

**ElseIf Age > 65 Then**

**Category = "Senior Citizen"**

**Else**

**Category = "Everyone Else"**

**End If**

The corresponding code with **Select Case** would be:

**Select Case Age**

**Case 5**

**Category = "Five Year Old"**

**Case 13 To 19**

**Category = "Teenager"**

**Case 20 To 35, 50, 60 To 65**

**Category = "Special Adult"**

**Case Is > 65**

**Category = "Senior Citizen"**

**Case Else**


**Category = "Everyone Else"**

**End Select**

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

## 1.12. The GoTo Statement

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

**Line10:**  **:**  
**GoTo Line10** **:**

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

## Visual Basic Looping

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

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

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

**Do Until/Loop Example:**



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

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

**Do/Loop While** Example:

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

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

**Do/Loop Until** Example:

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

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

Make sure you can always get out of a loop! Infinite loops are never nice. If you get into one, try **Ctrl+Break**. That sometimes works - other times the only way out is rebooting your machine!

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

### 1.13. Visual Basic Counting

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

#### Example

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

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

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

#### Example 2-3

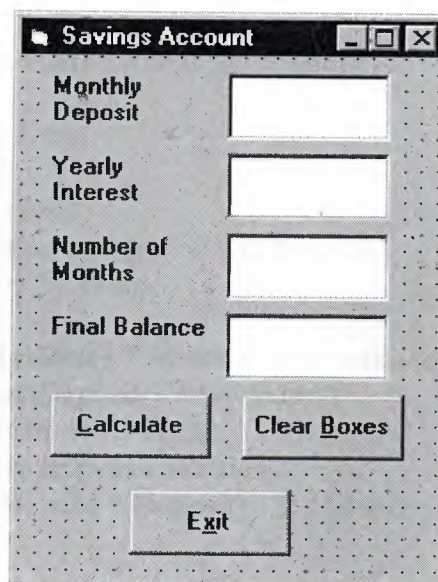
##### Savings Account - Decisions

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

#### Command3:

Caption	Clear &Boxes
Name	cmdClear

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



The screenshot shows a Visual Basic form titled "Savings Account". It contains four text boxes for input, labeled "Monthly Deposit", "Yearly Interest", "Number of Months", and "Final Balance". Below the text boxes are three command buttons: "Calculate", "Clear Boxes", and "Exit". The "Calculate" button has the letter 'C' underlined, and the "Clear Boxes" button has the letter 'B' underlined. The "Exit" button is centered at the bottom of the form.

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

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

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

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

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

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

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

```
Private Sub cmdCalculate_Click()  
    Dim IntRate As Single  
    Dim IntNew As Single  
    Dim Fcn As Single, FcnD As Single  
    'Read the four text boxes  
    Deposit = Val(txtDeposit.Text)  
    Interest = Val(txtInterest.Text)  
    IntRate = Interest / 1200  
    Months = Val(txtMonths.Text)  
    Final = Val(txtFinal.Text)  
    'Determine which box is blank  
    'Compute that missing value and put in text box  
    If Trim(txtDeposit.Text) = "" Then  
        'Deposit missing  
        Deposit = Final / (((1 + IntRate) ^ Months - 1) / IntRate)  
        txtDeposit.Text = Format(Deposit, "#####0.00")  
    ElseIf Trim(txtInterest.Text) = "" Then  
        'Interest missing - requires iterative solution  
        IntNew = (Final / (0.5 * Months * Deposit) - 1) / Months  
        Do  
            IntRate = IntNew  
            Fcn = (1 + IntRate) ^ Months - Final * IntRate / Deposit - 1
```



```

    FcnD = Months * (1 + IntRate) ^ (Months - 1) - Final / Deposit
    IntNew = IntRate - Fcn / FcnD
    Loop Until Abs(IntNew - IntRate) < 0.00001 / 12
    Interest = IntNew * 1200
    txtInterest.Text = Format(Interest, "##0.00")
ElseIf Trim(txtMonths.Text) = "" Then
    'Months missing
    Months = Log(Final * IntRate / Deposit + 1) / Log(1 + IntRate)
    txtMonths.Text = Format(Months, "###.0")
ElseIf Trim(txtFinal.Text) = "" Then
    'Final value missing
    Final = Deposit * ((1 + IntRate) ^ Months - 1) / IntRate
    txtFinal.Text = Format(Final, "#####0.00")
End If
End Sub

```

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

If the **Interest** value is missing, for the mathematically-inclined, we have to solve an *M*th-order polynomial using something called Newton-Raphson iteration - a good example of using a Do loop. If you're not mathematically inclined, you should see that finding the **Interest** value is straightforward. What we do is guess at what the interest is, compute a better guess, and repeat the process (loop) until the old guess and the new guess are close to each other. You can see each step in the code. Don't be intimidated by the code in this example. Upon study, you should see that it is just a straightforward list of instructions for the computer to follow based on input from the user.

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

## CHAPTER2 (Exploring the Visual Basic Toolbox)

### 1.1.The Message Box

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

#### MsgBox Message, Type, Title

where

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

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

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

#### **Dim Response as Integer**

**Response = MsgBox(Message, Type, Title)**

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

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

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

<b>Value</b>	<b>Meaning</b>	<b>Symbolic Constant</b>
0	No icon	(None)
16	Critical icon	vbCritical
32	Question mark	vbQuestion
48	Exclamation point	vbExclamation
64	Information icon	vbInformation

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

<b>Value</b>	<b>Meaning</b>	<b>Symbolic Constant</b>
0	First button default	vbDefaultButton1
256	Second button default	vbDefaultButton2
512	Third button default	vbDefaultButton3

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

<b>Value</b>	<b>Meaning</b>	<b>Symbolic Constant</b>
0	Application modal	vbApplicationModal
4096	System modal	vbSystemModal

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



Note for each option in **Type**, there are numeric values listed and symbolic constants.

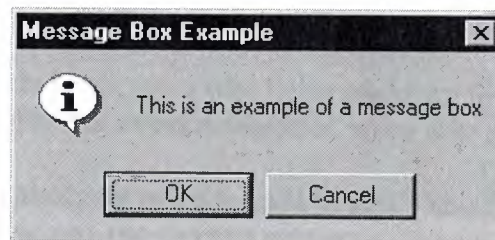
Recall, it is strongly suggested that the symbolic constants be used instead of the numeric values. You should agree that **vbOKOnly** means more than the number 0 when selecting the button type.

The value returned by the function form of the message box is related to the button clicked:

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

Message Box Example:

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



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

## 2.2.Object Methods

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

ObjectName.Method {optional arguments}

Note this is another use of the dot notation.

## 2.3.The Form Object

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

Form Properties:

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

Form Events:

- Activate** Form\_Activate event is triggered when form becomes the active window.
- Click** Form\_Click event is triggered when user clicks on form.
- DblClick** Form\_DblClick event is triggered when user double-clicks on form.
- Load** Form\_Load event occurs when form is loaded. This is a good place to initialize variables and set any run-time properties.

Form Methods:

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

### Examples

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

## 2.4.Command Buttons



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

Command Button Properties:

**Appearance** Selects 3-D or flat appearance.

**Cancel** Allows selection of button with **Esc** key (only one button on a form can have this property True).

**Caption** String to be displayed on button.

**Default** Allows selection of button with **Enter** key (only one button on a form can have this property True).

**Font** Sets font type, style, size.

Command Button Events:

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

## 2.5.Label Boxes



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

Label Properties:

**Alignment** Aligns caption within border.

**Appearance** Selects 3-D or flat appearance.

**AutoSize** If True, the label is resized to fit the text specified by the caption property. If False, the label will remain the size defined at design time and the text may be clipped.

**BorderStyle** Determines type of border.

**Caption** String to be displayed in box.

**Font** Sets font type, style, size.

**WordWrap** Works in conjunction with AutoSize property. If AutoSize = True, WordWrap = True, then the text will wrap and label will expand vertically to fit the Caption. If AutoSize = True, WordWrap = False, then the text will not wrap and the label expands horizontally to fit the Caption. If AutoSize = False, the text will not wrap regardless of WordWrap value.

Label Events:

**Click** Event triggered when user clicks on a label.

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

## 2.6.Text Boxes



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

Text Box Properties:

**Appearance** Selects 3-D or flat appearance.

**BorderStyle** Determines type of border.

**Font** Sets font type, style, size.

**MaxLength** Limits the length of displayed text (0 value indicates unlimited length).

**MultiLine** Specifies whether text box displays single line or multiple lines.

**PasswordChar** Hides text with a single character.

**ScrollBars** Specifies type of displayed scroll bar(s).

**SelLength** Length of selected text (run-time only).

**SelStart** Starting position of selected text (run-time only).

**SelText** Selected text (run-time only).

**Tag** Stores a string expression.

**Text** Displayed text.



## Text Box Events:

- Change** Triggered every time the **Text** property changes.
- LostFocus** Triggered when the user leaves the text box. This is a good place to examine the contents of a text box after editing.
- KeyPress** Triggered whenever a key is pressed. Used for key trapping, as seen in last class.

## Text Box Methods:

- SetFocus** Places the cursor in a specified text box.

## Example

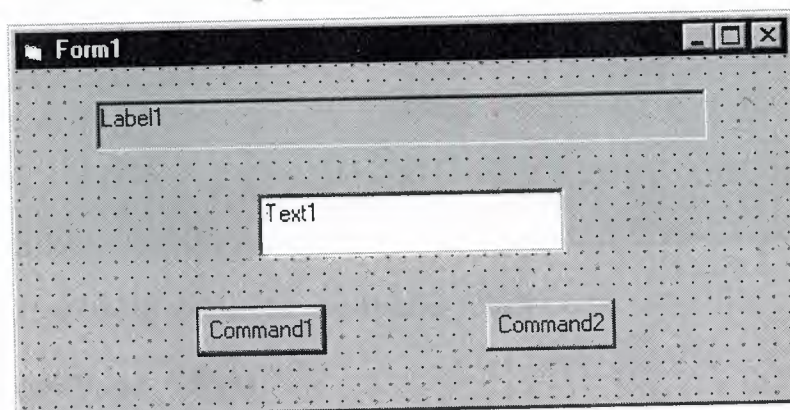
`txtExample.SetFocus ' moves cursor to txtExample`

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

## Example 3-1

### Password Validation

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



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

**Form1:**

BorderStyle	1-Fixed Single
Caption	Password Validation
Name	frmPassword

**Label1:**

Alignment	2-Center
BorderStyle	1-Fixed Single
Caption	Please Enter Your Password:
FontSize	10
FontStyle	Bold

**Text1:**

FontSize	14
FontStyle	Regular
Name	txtPassword
PasswordChar	*
Tag	[Whatever you choose as a password]
Text	[Blank]

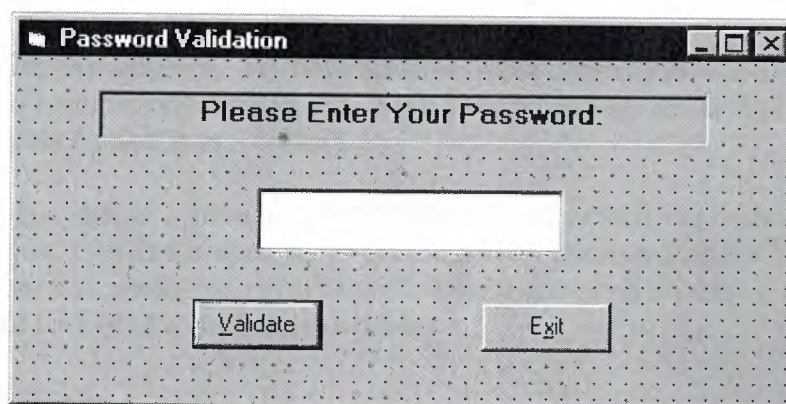
**Command1:**

Caption	&Validate
Default	True
Name	cmdValid

**Command2:**

Cancel	True
Caption	E&xit
Name	cmdExit

Your form should now look like this:



4. Attach the following code to the **cmdValid\_Click** event.

```
Private Sub cmdValid_Click()
'This procedure checks the input password
Dim Response As Integer
If txtPassword.Text = txtPassword.Tag Then
'If correct, display message box
```

```

    MsgBox "You've passed security!", vbOKOnly + vbExclamation, "Access
Granted"
Else
    'If incorrect, give option to try again
    Response = MsgBox("Incorrect password", vbRetryCancel + vbCritical, "Access
Denied")
    If Response = vbRetry Then
        txtPassword.SelStart = 0
        txtPassword.SelLength = Len(txtPassword.Text)
    Else
        End
    End If
End If
txtPassword.SetFocus
End Sub

```

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

5. Attach the following code to the **Form\_Activate** event.

```

Private Sub Form_Activate()
txtPassword.SetFocus
End Sub

```

6. Attach the following code to the **cmdExit\_Click** event.

```

Private Sub cmdExit_Click()
End
End Sub

```

7. Try running the program. Try both options: input correct password (note it is case sensitive) and input incorrect password. Save your project (saved as **Example3-1** in the **LearnVB6/VB6 Code/Class 3** folder).

If you have time, define a constant, **TRYMAX = 3**, and modify the code to allow the user to have just **TRYMAX** attempts to get the correct password. After the final try, inform the user you are logging him/her off. You'll also need a variable that counts the number of tries (make it a **Static** variable).

## 2.7.Check Boxes



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



**Caption** Identifying text next to box.

**Font** Sets font type, style, size.

**Value** Indicates if unchecked (0, vbUnchecked), checked (1, vbChecked), or grayed out (2, vbGrayed).

- Check Box Events:

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

## 2.8.Option Buttons



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

- Option Button Properties:

**Caption** Identifying text next to button.

**Font** Sets font type, style, size.

**Value** Indicates if selected (True) or not (False). Only one option button in a group can be True. One button in each group of option buttons should always be initialized to True at design time.

- Option Button Events:

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

## 2.9.Arrays

- Up to now, we've only worked with regular variables, each having its own unique name. Visual Basic has powerful facilities for handling arrays, which provide a way to store a large number of variables under the same name. Each variable, called an element, in an array must have the same data type, and they are distinguished from each other by an array index. In this class, we work with one-dimensional arrays, although multi-dimensional arrays are possible.

- Arrays are declared in a manner identical to that used for regular variables. For example, to declare an integer array named '**Item**', with dimension **9**, at the procedure level, we use:

**Dim Item(9) as Integer**



If we want the array variables to retain their value upon leaving a procedure, we use the keyword **Static**:

### **Static Item(9) as Integer**

At the **form** or **module** level, in the general declarations area of the Code window, use:

### **Dim Item(9) as Integer**

And, at the module level, for a **global** declaration, use:

### **Global Item(9) as Integer**

The index on an array variable begins at 0 and ends at the dimensioned value. For example, the **Item** array in the above examples has **ten** elements, ranging from Item(0) to Item(9). You use array variables just like any other variable - just remember to include its name and its index. For example, to set Item(5) equal to 7, you simply write:

```
Item(5) = 7
```

To sum all the array elements, use:

```
Sum = 0
For I = 0 to 9
    Sum = Sum + Item(I)
Next I
```

### **Control Arrays**

Similar to variable arrays, **control arrays** are groups of like controls with the same name and referred to by individual index values. Use of control arrays depends on the application. For example, option buttons are almost always grouped in control arrays.

Control arrays are a convenient way to handle groups of controls that perform a similar function. All of the events available to the single control are still available to the array of controls, the only difference being an argument indicating the **Index** (a property of control arrays) of the selected array element is passed to the event. Hence, instead of writing individual procedures for each control (i.e. not using control arrays), you only have to write one procedure for each array.

Another advantage to control arrays is that you can add or delete array elements at run-time. You cannot do that with controls (objects) not in arrays. Refer to the **Load** and **Unload** statements in on-line help for the proper way to add and delete control array elements at run-time.

Two ways to **create** a control array:

1. Create an individual control and set desired properties. Copy the control using the editor, then paste it on the form. Visual Basic will pop-up a dialog box that will ask you

if you wish to create a control array. Respond yes and the array is created (an **Index** value is assigned to the control).

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

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

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

We'll use control arrays in the next example.

## 2.10.Frames



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

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

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

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

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

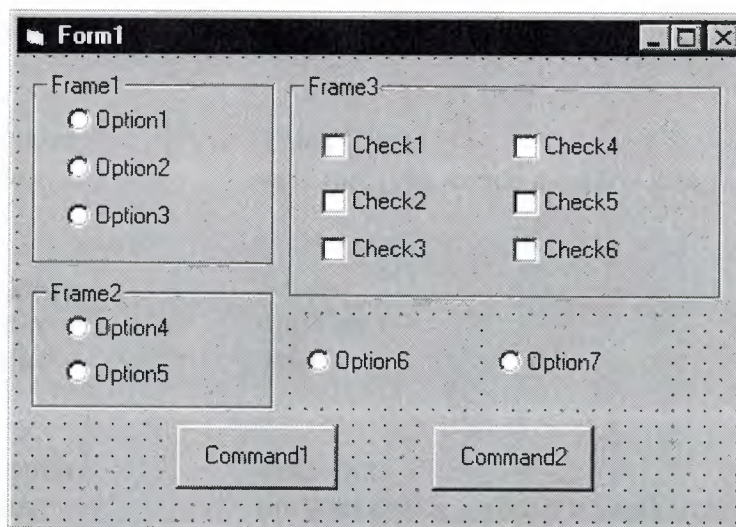
Frame Properties:

- Caption** Title information at top of frame.  
**Font** Sets font type, style, size.  
**Enabled** If False, **all** controls within frame are disabled

### Example 2-2

#### **Pizza Order**

1. Start a new project. We'll build a form where a pizza order can be entered by simply clicking on check boxes and option buttons.
2. Draw three frames. In the first, draw three option buttons, in the second, draw two option buttons, and in the third, draw six check boxes. Draw two option buttons on the form. Add two command buttons. Make things look something like this.



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

#### **Form1:**

BorderStyle	1-Fixed Single
Caption	Pizza Order
Name	frmPizza

#### **Frame1:**

Caption	Size
---------	------

#### **Frame2:**

Caption	Crust Type
---------	------------

#### **Frame3**

Caption	Toppings
---------	----------



**Check4:**

Caption	Onions
Name	chkTop

**Check5:**

Caption	Green Peppers
Name	chkTop

**Check6:**

Caption	Tomatoes
Name	chkTop

**Command1:**

Caption	&Build Pizza
Name	cmdBuild

**Command2:**

Caption	E&xit
Name	cmdExit

The form should look like this now:

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

**Option Explicit****Dim PizzaSize As String****Dim PizzaCrust As String****Dim PizzaWhere As String**

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



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

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

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

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

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

```
Private Sub optCrust_Click(Index As Integer)  
'Read crust type  
PizzaCrust = optCrust(Index).Caption  
End Sub
```

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

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

7. Attach this code to the **cmdBuild\_Click** event.

```
Private Sub cmdBuild_Click()  
'This procedure builds a message box that displays your pizza type  
Dim Message As String  
Dim I As Integer  
Message = PizzaWhere + vbCr  
Message = Message + PizzaSize + " Pizza" + vbCr  
Message = Message + PizzaCrust + vbCr  
For I = 0 To 5  
    If chkTop(I).Value = vbChecked Then Message = Message + chkTop(I).Caption +  
    vbCr  
Next I  
MsgBox Message, vbOKOnly, "Your Pizza"  
End Sub
```

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

7. Attach this code to the **cmdExit\_Click** event.

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

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

9. If you have time, try these modifications:

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

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

## 2.11. List Boxes



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

List Box Properties:

**Appearance** Selects 3-D or flat appearance.

**List** Array of items in list box.

**ListCount** Number of items in list.

**ListIndex** The number of the most recently selected item in list. If no item is selected, **ListIndex** = -1.

**MultiSelect** Controls how items may be selected (0-no multiple selection allowed, 1-multiple selection allowed, 2-group selection allowed).

**Selected** Array with elements set equal to True or False, depending on whether corresponding list item is selected.

**Sorted** True means items are sorted in 'Ascii' order, else items appear in order added.

**Text** Text of most recently selected item.

List Box Events:

**Click** Event triggered when item in list is clicked.

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

List Box Methods:

**AddItem** Allows you to insert item in list.

**Clear** Removes all items from list box.

**RemoveItem** Removes item from list box, as identified by index of item to remove.

### Examples

**lstExample.AddItem "This is an added item"**

**lstExample.Clear**

**lstExample.RemoveItem 4 ' removes lstExample.List(4)**

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

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

## 2.12.Combo Boxes



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

Combo Box Properties:

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

**Appearance** Selects 3-D or flat appearance.

**List** Array of items in list box portion.

**ListCount** Number of items in list.

**ListIndex** The number of the most recently selected item in list. If no item is selected, ListIndex = -1.

**Sorted** True means items are sorted in 'Ascii' order, else items appear in order added.

**Style** Selects the combo box form.

Style = 0, Dropdown combo; user can change selection.

Style = 1, Simple combo; user can change selection (make sure to resize default box so dropdown area appears).

Style = 2, Dropdown combo; user cannot change selection.

**Text** Text of most recently selected item.



### Combo Box Events:

**Click** Event triggered when item in list is clicked.

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

### Combo Box Methods:

**AddItem** Allows you to insert item in list.

**Clear** Removes all items from list box.

**RemoveItem** Removes item from list box, as identified by index of item to remove.

### Examples

**cboExample.AddItem "This is an added item"**

**cboExample.Clear**

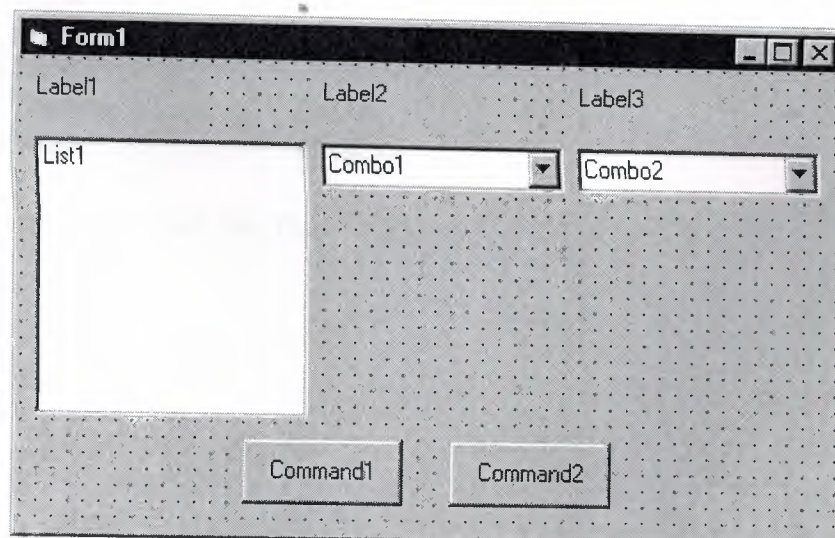
**cboExample.RemoveItem 4 ' removes cboExample.List(4)**

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

### Example 3-3

#### Flight Planner

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





3. Set the form and object properties:

**Form1:**

BorderStyle	1-Fixed Single
Caption	Flight Planner
Name	frmFlight

**List1:**

Name	lstCities
Sorted	True

**Combo1:**

Name	cboSeat
Style	2-Dropdown List

**Combo2:**

Name	cboMeal
Style	1-Simple
Text	[Blank]

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

**Label1:**

Caption	Destination City
---------	------------------

**Label2:**

Caption	Seat Location
---------	---------------

**Label3:**

Caption	Meal Preference
---------	-----------------

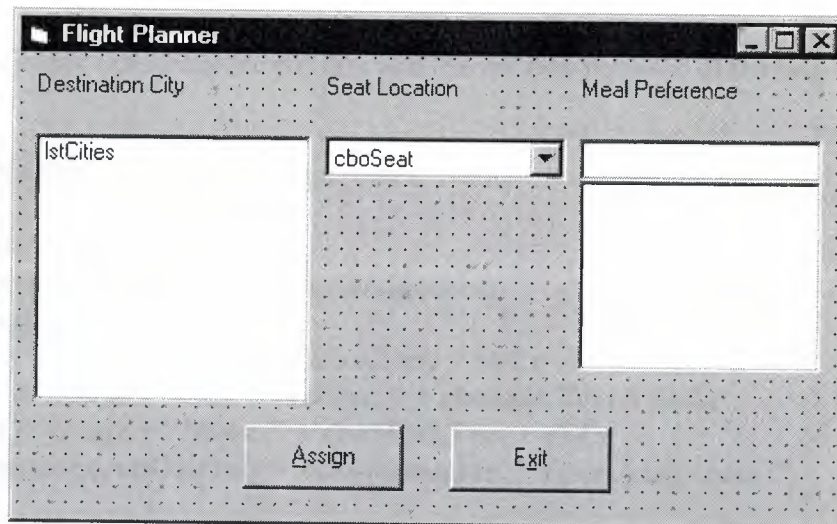
**Command1:**

Caption	&Assign
Name	cmdAssign

**Command2:**

Caption	E&xit
Name	cmdExit

Now, the form should look like this:



4. Attach this code to the **Form\_Load** procedure:

```

Private Sub Form_Load()
'Add city names to list box
lstCities.Clear
lstCities.AddItem "San Diego"
lstCities.AddItem "Los Angeles"
lstCities.AddItem "Orange County"
lstCities.AddItem "Ontario"
lstCities.AddItem "Bakersfield"
lstCities.AddItem "Oakland"
lstCities.AddItem "Sacramento"
lstCities.AddItem "San Jose"
lstCities.AddItem "San Francisco"
lstCities.AddItem "Eureka"
lstCities.AddItem "Eugene"
lstCities.AddItem "Portland"
lstCities.AddItem "Spokane"
lstCities.AddItem "Seattle"
lstCities.ListIndex = 0

'Add seat types to first combo box
cboSeat.AddItem "Aisle"
cboSeat.AddItem "Middle"
cboSeat.AddItem "Window"
cboSeat.ListIndex = 0

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

```

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

5. Attach this code to the **cmdAssign\_Click** event:

```
Private Sub cmdAssign_Click()  
    'Build message box that gives your assignment  
    Dim Message As String  
    Message = "Destination: " + lstCities.Text + vbCrLf  
    Message = Message + "Seat Location: " + cboSeat.Text + vbCrLf  
    Message = Message + "Meal: " + cboMeal.Text + vbCrLf  
    MsgBox Message, vbOKOnly + vbInformation, "Your Assignment"  
End Sub
```

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

5. Attach this code to the **cmdExit\_Click** event:

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

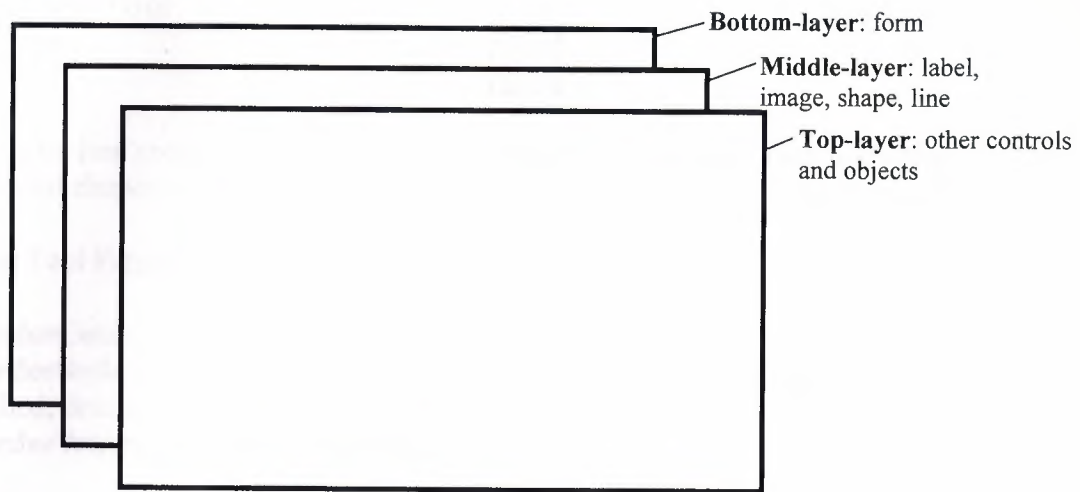
6. Run the application. Save the project (saved as **Example3-3** in **LearnVB6/VB Code/Class 3** folder).

## Chapter3 More Exploration of the Visual Basic Toolbox

### 3.1.Display Layers

In this class, we will look at our first graphic type controls: line tools, shape tools, picture boxes, and image boxes. And, with this introduction, we need to discuss the idea of display **layers**.

Items shown on a form are not necessarily all on the same layer of display. A form's display is actually made up of three layers as sketched below. All information displayed directly on the form (by printing or drawing with graphics methods, discussed in Chapter 7) appears on the **bottom-layer**. Information from label boxes, image boxes, line tools, and shape tools, appears on the **middle-layer**. And, all other objects are displayed on the **top-layer**.



What this means is you have to be careful where you put things on a form or something could be covered up. For example, a command button placed on top of it would hide text printed on the form. Things drawn with the shape tool are covered by all controls except the image box, line control and label control.

The next question then is what establishes the relative location of objects in the same layer. That is, say two command buttons are in the same area of a form - which one lies on top of which one? The order in which objects in the same layer overlay each other is called the **Z-order**. This order is first established when you draw the form (you can also establish it in code using the **Zorder** property). Items drawn last lie over items drawn earlier. Once drawn, however, clicking on the desired object and choosing **Bring to Front** from Visual Basic's **Edit** menu can modify the Z-order. The **Send to Back** command has the opposite effect. Note these two commands only work within a layer; middle-layer objects will always appear behind top-layer objects and lower layer objects will always appear behind middle-layer objects.



### 3.2.Line Tool



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

Line Tool Properties:

**BorderColor** Determines the line color.

**BorderStyle** Determines the line 'shape'. Lines can be transparent, solid, dashed, dotted, and combinations.

**BorderWidth** Determines line width.

There are no events or methods associated with the line tool.

Since the line tool lies in the middle-layer of the form display, any lines drawn will be obscured by all controls except the shape tool, label box or image box.

### 3.3.Shape Tool



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

Shape Tool Properties:

**BackColor** Determines the background color of the shape (only used when FillStyle not Solid).

**BackStyle** Determines whether the background is transparent or opaque.

**BorderColor** Determines the color of the shape's outline.

**BorderStyle** Determines the style of the shape's outline. The border can be transparent, solid, dashed, dotted, and combinations.

**BorderWidth** Determines the width of the shape border line.

**FillColor** Defines the interior color of the shape.

**FillStyle** Determines the interior pattern of a shape. Some choices are: solid, transparent, cross, etc.

**Shape** Determines whether the shape is a square, rectangle, circle, or some other choice.

Like the line tool, events and methods are not used with the shape tool.

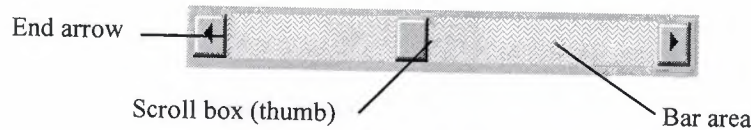
Shapes are covered by all objects except perhaps line tools, label boxes and image boxes (depends on their Z-order) and printed or drawn information. This is a good feature in that you usually use shapes to contain a group of control objects and you'd want them to lie on top of the shape.

### 3.4.Horizontal and Vertical Scroll Bars



Horizontal and vertical **scroll bars** are widely used in Windows applications. Scroll bars provide an intuitive way to move through a list of information and make great input devices.

Both type of scroll bars are comprised of three areas that can be clicked, or dragged, to change the scroll bar value. Those areas are:



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

Scroll Bar Properties:

**LargeChange** Increment added to or subtracted from the scroll bar **Value** property when the bar area is clicked.

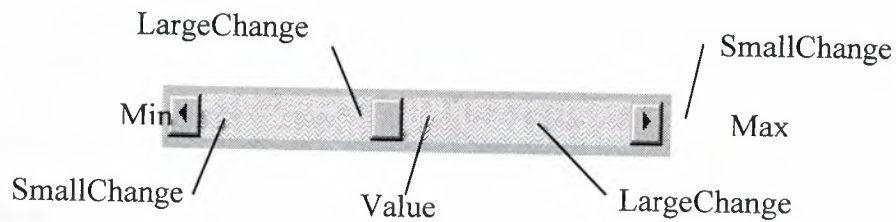
**Max** The value of the horizontal scroll bar at the far right and the value of the vertical scroll bar at the bottom. Can range from -32,768 to 32,767.

**Min** The other extreme value - the horizontal scroll bar at the left and the vertical scroll bar at the top. Can range from -32,768 to 32,767.

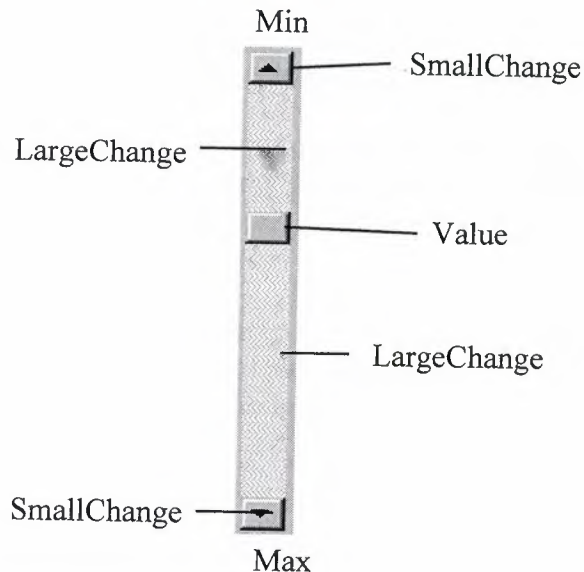
**SmallChange** The increment added to or subtracted from the scroll bar **Value** property when either of the scroll arrows is clicked.

**Value** The current position of the scroll box (thumb) within the scroll bar. If you set this in code, Visual Basic moves the scroll box to the proper position.

Properties for horizontal scroll bar:



Properties for vertical scroll bar:



A couple of important notes about scroll bars:

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



## Scroll Bar Events:

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

**Scroll** Event triggered continuously whenever the scroll box is being moved.

## Picture Boxes



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

Picture boxes lie in the top layer of the form display. They behave very much like small forms within a form, possessing most of the same properties as a form.

### Picture Box Properties:

**AutoSize** If True, box adjusts its size to fit the displayed graphic.

**Font** Sets the font size, style, and size of any printing done in the picture box.

**Picture** Establishes the graphics file to display in the picture box.

### Picture Box Events:

**Click** Triggered when a picture box is clicked.

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

### Picture Box Methods:

**Cls** Clears the picture box.

**Print** Prints information to the picture box.

## Examples

**picExample.Cls** ' clears the box **picExample**

**picExample.Print** "a picture box" ' prints text string

### Picture Box LoadPicture Procedure:

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

### Example

```
picExample.Picture = LoadPicture("c:\pix\sample.bmp")
```

This command loads the graphics file c:\pix\sample.bmp into the Picture property of the picExample picture box. The argument in the LoadPicture function must be a legal, complete path and file name, else your program will stop with an error message.



Five types of graphics files can be loaded into a picture box:

**Bitmap** An image represented by pixels and stored as a collection of bits in which each bit corresponds to one pixel. Usually has a **.bmp** extension. Appears in original size.

**Icon** A special type of bitmap file of maximum 32 x 32 size. Has a **.ico** extension. We'll create icon files in Class 5. Appears in original size.

**Metafile** A file that stores an image as a collection of graphical objects (lines, circles, polygons) rather than pixels. Metafiles preserve an image more accurately than bitmaps when resized. Has a **.wmf** extension. Resizes itself to fit the picture box area.

**JPEG** JPEG (Joint Photographic Experts Group) is a compressed bitmap format which supports 8 and 24 bit color. It is popular on the Internet. Has a **.jpg** extension and appears in original size.

**GIF** GIF (Graphic Interchange Format) is a compressed bitmap format originally developed by CompuServe. It supports up to 256 colors and is popular on the Internet. Has a **.gif** extension and appears in original size.

### 3.5. Image Boxes



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

Image boxes appear in the middle-layer of form display, hence they could be obscured by picture boxes and other objects. Image box graphics can be resized by using the **Stretch** property.

Image Box Properties:

**Picture** Establishes the graphics file to display in the image box.

**Stretch** If False, the image box resizes itself to fit the graphic. If True, the graphic resizes to fit the control area.

Image Box Events:

**Click** Triggered when a image box is clicked.

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

The image box does not support any methods, however it does use the **LoadPicture** function. It is used in exactly the same manner as the picture box uses it. And image boxes can load the same file types: bitmap (.bmp), icon (.ico), metafiles (.wmf), GIF files (.gif), and JPEG files (.jpg). With **Stretch = True**, all five graphic types will expand to fit the image box area. Metafiles, GIF files and JPEG files scale nicely using the **Stretch** property.

### 3.6.Drive List Box



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

Drive List Box Properties:

**Drive** Contains the name of the currently selected drive.

Drive List Box Events:

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

### 3.7.Directory List Box



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

Directory List Box Properties:

**Path** Contains the current directory path.

Directory List Box Events:

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

### 3.8.File List Box



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

File List Box Properties:

**FileName** Contains the currently selected file name.

**ListCount** Number of files listed

**List** Array of file names in list box

**MultiSelect** Allows multiple selection in list box

**Path** Contains the current path directory.

**Pattern** Contains a string that determines which files will be displayed. It supports the use of \* and ? wildcard characters. For example, using \*.dat only displays files with the .dat extension.

File List Box Events:

**DbClick** Triggered whenever a file name is double-clicked.

**PathChange** Triggered whenever the path changes in a file list box.

### 3.10.Synchronizing the Drive, Directory, and File List Boxes

The drive, directory, and file list boxes are almost always used together to obtain a file name. As such, it is important that their operation be synchronized to insure the displayed information is always consistent.

When the drive selection is changed (drive box **Change** event), you should update the directory path. For example, if the drive box is named `drvExample` and the directory box is `dirExample`, use the code:

```
dirExample.Path = drvExample.Drive
```

When the directory selection is changed (directory box **Change** event), you should update the displayed file names. With a file box named `filExample`, this code is:

```
filExample.Path = dirExample.Path
```

Once all of the selections have been made and you want the file name, you need to form a text string that correctly and completely specifies the file identifier. This string concatenates the drive, directory, and file name information. This should be an easy task, except for one problem. The problem involves the backslash (\) character. If you are at the root directory of your drive, the path name ends with a backslash. If you are not at the root directory, there is no backslash at the end of the path name and you have to add one before tacking on the file name.

Example code for concatenating the available information into a proper file name and then loading it into an image box is:

```
Dim YourFile as String
```

```
If Right(filExample.Path,1) = "\" Then
```

```
    YourFile = filExample.Path + filExample.FileName
```

```
Else
```

```
    YourFile = filExample.Path + "\" + filExample.FileName
```

```
End If
```

```
imgExample.Picture = LoadPicture(YourFile)
```

Note we only use properties of the file list box. The drive and directory box properties are only used to create changes in the file list box via code.



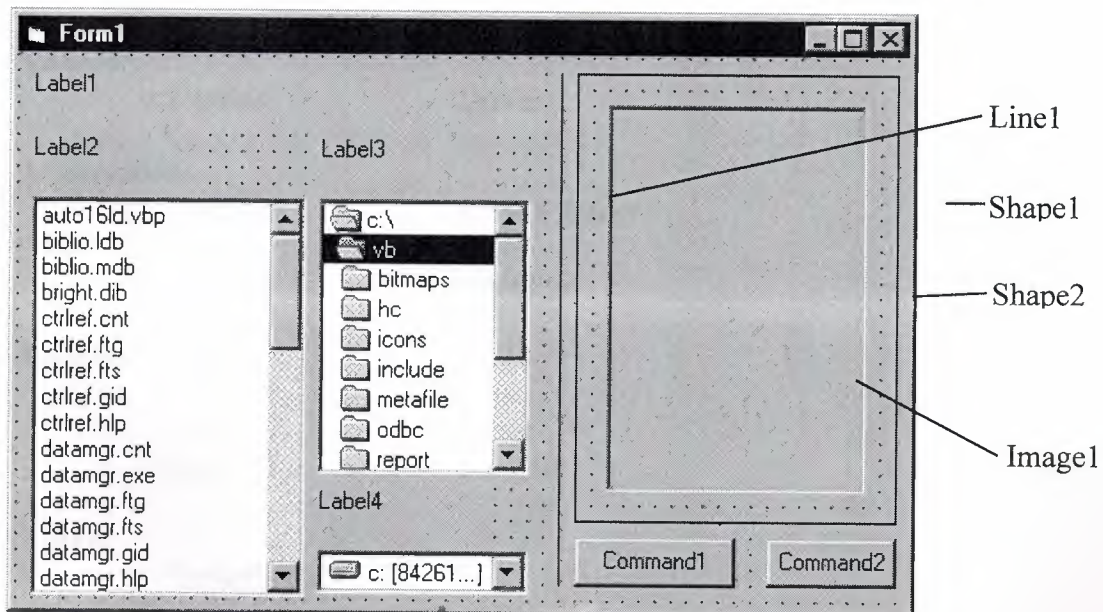
## Example 4-2

### Image Viewer

Start a new project. In this application, we search our computer's file structure for graphics files and display the results of our search in an image box.

#### One possible solution to the Image Viewer Application:

1. Place a drive list box, directory list box, file list box, four label boxes, a line (use the line tool) and a command button on the form. We also want to add an image box, but make it look like it's in some kind of frame. Build this display area in these steps: draw a 'large shape', draw another shape within this first shape that is the size of the image display area, and lastly, draw an image box right on top of this last shape. Since the two shapes and image box are in the same display layer, the image box is on top of the second shape which is on top of the first shape, providing the desired effect of a kind of picture frame. The form should look like this:



Note the second shape is directly beneath the image box.

2. Set properties of the form and each object.

#### Form1:

BorderStyle	1-Fixed Single
Caption	Image Viewer
Name	frmImage

#### Drive1:

Name	drvImage
------	----------

<b>Dir1:</b>	Name	dirImage
<b>File1:</b>	Name	fillImage
	Pattern	*.bmp;*.ico;*.wmf;*.gif;*.jpg [type this line with <u>no</u> spaces]
<b>Label1:</b>	Caption	[Blank]
	BackColor	Yellow
	BorderStyle	1-Fixed Single
	Name	lblImage
<b>Label2:</b>	Caption	Files:
<b>Label3:</b>	Caption	Directories:
<b>Label4:</b>	Caption	Drives:
<b>Command1:</b>	Caption	&Show Image
	Default	True
	Name	cmdShow
<b>Command2:</b>	Cancel	True
	Caption	E&xit
	Name	cmdExit
<b>Line1:</b>	BorderWidth	3
<b>Shape1:</b>	BackColor	Cyan
	BackStyle	1-Opaque
	FillColor	Blue
	FillStyle	4-Upward Diagonal
	Shape	4-Rounded Rectangle
<b>Shape2:</b>	BackColor	White
	BackStyle	1-Opaque



**Image1:**

BorderStyle	1-Fixed Single
Name	imgImage
Stretch	True

3. Attach the following code to the **drvImage\_Change** procedure.

```
Private Sub drvImage_Change()  
'If drive changes, update directory  
dirImage.Path = drvImage.Drive  
End Sub
```

When a new drive is selected, this code forces the directory list box to display directories on that drive.

4. Attach this code to the **dirImage\_Change** procedure.

```
Private Sub dirImage_Change()  
'If directory changes, update file path  
fillImage.Path = dirImage.Path  
End Sub
```

Likewise, when a new directory is chosen, we want to see the files on that directory.

5. Attach this code to the **cmdShow\_Click** event.

```
Private Sub cmdShow_Click()  
'Put image file name together and  
'load image into image box  
Dim ImageName As String  
'Check to see if at root directory  
If Right(fillImage.Path, 1) = "\" Then  
    ImageName = fillImage.Path + fillImage.filename  
Else  
    ImageName = fillImage.Path + "\" + fillImage.filename  
End If  
lblImage.Caption = ImageName  
imgImage.Picture = LoadPicture(ImageName)  
End Sub
```

This code forms the file name (**ImageName**) by concatenating the directory path with the file name. It then displays the complete name and loads the picture into the image box.

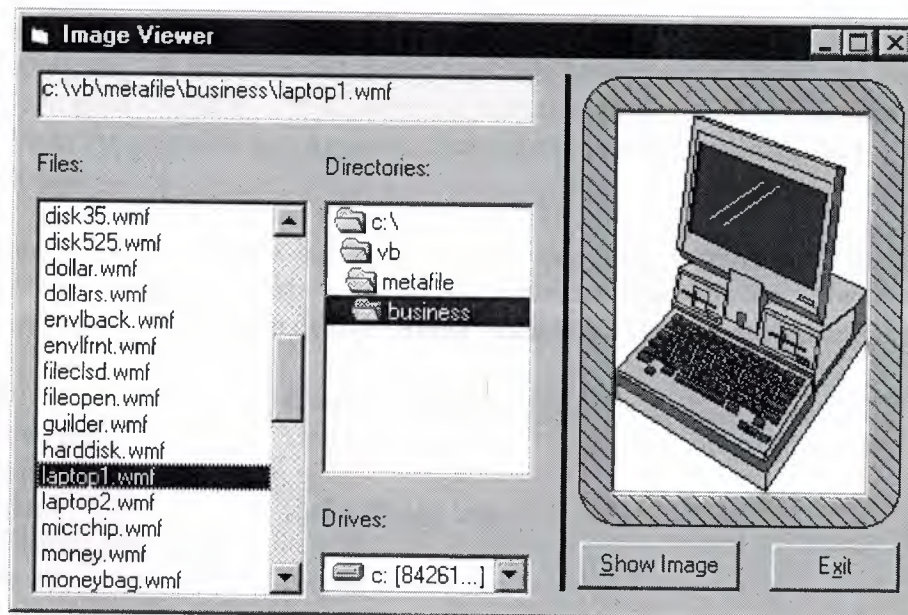


6. Copy the code from the **cmdShow\_Click** procedure and paste it into the **fillImage\_DbClick** procedure. The code is identical because we want to display the image either by double-clicking on the filename or clicking the command button once a file is selected. Those of you who know how to call routines in Visual Basic should note that this duplication of code is unnecessary - we could simply have the **fillImage\_DbClick** procedure call the **cmdShow\_Click** procedure. We'll learn more about this next class.

7. Attach this code to the **cmdExit\_Click** procedure.

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

8. Save your project (saved as **Example4-2** in **LearnVB6/VB Code/Class 4** folder). Run and try the application. Find bitmaps, icons, and metafiles. Notice how the image box Stretch property affects the different graphics file types. Here's how the form should look when displaying one example metafile:





### 3.11.Common Dialog Boxes



The primary use for the drive, directory, and file name list boxes is to develop custom file access routines. For example, you could just use the file list box (not allowing a user to change drive or directory). For general use, two common file access routines in Windows-based applications are the **Open File** and **Save File** operations. These can be used in our Visual Basic applications and, fortunately, you don't have to build these routines.

To give the user a standard interface for common operations in Windows-based applications, Visual Basic provides a set of **common dialog boxes**, two of which are the **Open** and **Save As** dialog boxes. Such boxes are familiar to any Windows user and give your application a professional look. And, some context-sensitive help is available while the box is displayed. Appendix II lists many symbolic constants used with common dialog boxes.

The Common Dialog control is a '**custom control**' which means we have to make sure some other files are present to use it. In normal setup configurations, Visual Basic does this automatically. If the common dialog box does not appear in the Visual Basic toolbox, you need to add it. This is done by selecting **Components** under the **Project** menu. When the selection box appears, click on **Microsoft Common Dialog Control**, then click **OK**.

The common dialog tool, although it appears on your form, is invisible at run-time. You cannot control where the common dialog box appears on your screen. The tool is invoked at run-time using one of five '**Show**' methods. These methods are:

Method	Common Dialog Box
ShowOpen	Open dialog box
ShowSaveSave As dialog box	
ShowColor	Color dialog box
ShowFontFont dialog box	
ShowPrinter	Printer dialog box

The format for establishing a common dialog box named **cdlExample** so that an **Open** box appears is:

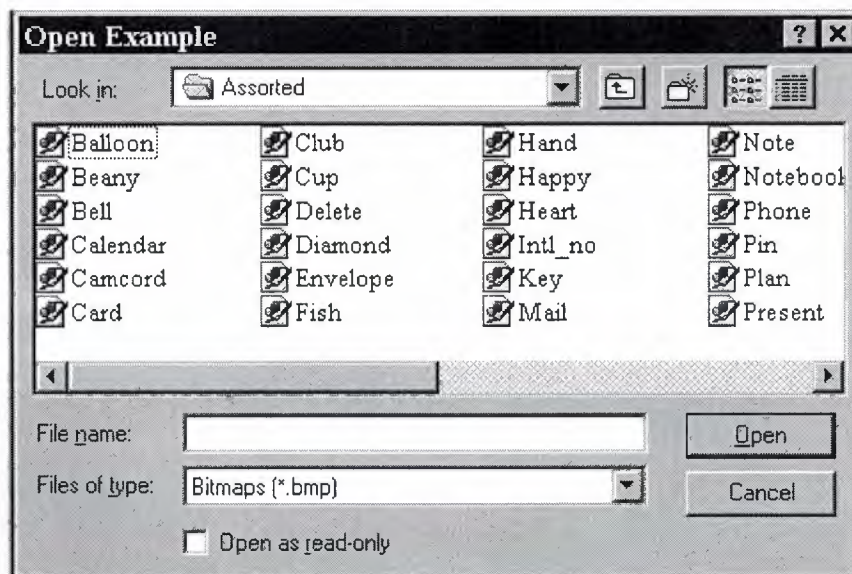
### **cdlExample.ShowOpen**

Control to the program returns to the line immediately following this line, once the dialog box is closed in some manner. Common dialog boxes are system modal.

Learning proper use of all the common dialog boxes would require an extensive amount of time. In this class, we'll limit ourselves to learning the basics of getting file names from the **Open** and **Save As** boxes in their default form.

## **3.12.Open Common Dialog Box**

The **Open** common dialog box provides the user a mechanism for specifying the name of a file to open. We'll worry about how to open a file in Class 6. The box is displayed by using the **ShowOpen** method. Here's an example of an Open common dialog box:



## Open Dialog Box Properties:

**CancelError** If True, generates an error if the Cancel button is clicked. Allows you to use error-handling procedures to recognize that Cancel was clicked.

**DialogTitle** The string appearing in the title bar of the dialog box. Default is Open. In the example, the DialogTitle is Open Example.

**FileName** Sets the initial file name that appears in the File name box. After the dialog box is closed, this property can be read to determine the name of the selected file.

**Filter** Used to restrict the filenames that appear in the file list box. Complete filter specifications for forming a Filter can be found using on-line help. In the example, the Filter was set to allow Bitmap (\*.bmp), Icon (\*.ico), Metafile (\*.wmf), GIF (\*.gif), and JPEG (\*.jpg) types (only the Bitmap choice is seen).

**FilterIndex** Indicates which filter component is default. The example uses a 1 for the FilterIndex (the default value).

**Flags** Values that control special features of the Open dialog box (see Appendix II). The example uses no Flags value.

When the user closes the Open File box, you should check the returned file name to make sure it meets the specifications your application requires before you try to open the file.



### Quick Example: The Open Dialog Box

1. Start a new project. Place a common dialog control, a label box, and a command button on the form. Set the following properties:

#### Form1:

Caption	Common Dialog Examples
Name	frmCommon

#### CommonDialog1:

DialogTitle	Open Example
Filter	Bitmaps (*.bmp) *.bmp  Icons (*.ico) *.ico Metafiles (*.wmf) *.wmf  GIF Files (*.gif) *.gif JPEG Files (*.jpg) *.jpg (all on one line)
Name	cdlExample

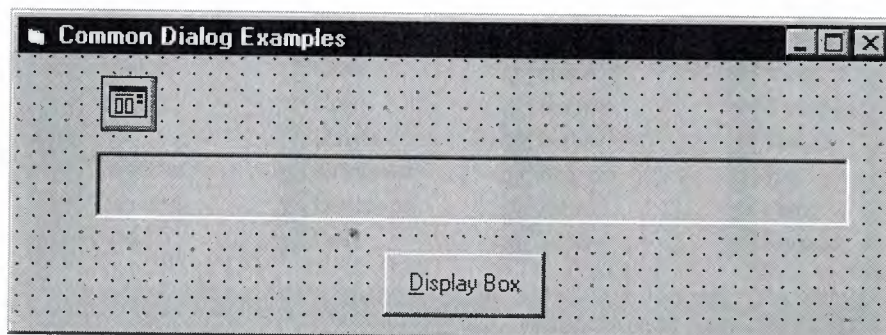
#### Label1:

BorderStyle	1-Fixed Single
Caption	[Blank]
Name	lblExample

#### Command1:

Caption	&Display Box
Name	cmdDisplay

When done, the form should look like this (make sure your label box is very long):





2. Attach this code to the `cmdDisplay_Click` procedure.

```
Private Sub cmdDisplay_Click()  
cdlExample.ShowOpen  
lblExample.Caption = cdlExample.filename  
End Sub
```

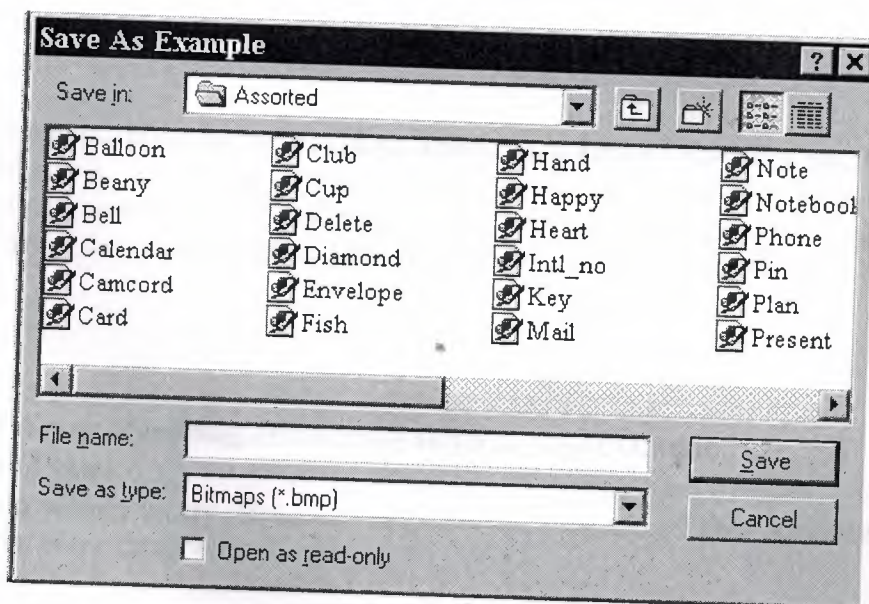
This code brings up the Open dialog box when the button is clicked and shows the file name selected by the user once it is closed.

3. Save the application (saved as **QExampleOpen** in **LearnVB6/VB Code/Class 4** folder). Run it and try selecting file names and typing file names. Notice names can be selected by highlighting and clicking the **OK** button or just by double-clicking the file name. In this example, clicking the **Cancel** button is not trapped, so it has the same effect as clicking **OK**.

4. Notice once you select a file name, the next time you open the dialog box, that selected name appears as default, since the `FileName` property is not affected in code.

### 3.13. Save As Common Dialog Box

The **Save As** common dialog box provides the user a mechanism for specifying the name of a file to save. We'll worry about how to save a file in Class 6. The box is displayed by using the **ShowSave** method.. Here's an example of a Save As common dialog box:



Save As Dialog Box Properties (mostly the same as those for the Open box):

**CancelError** If True, generates an error if the Cancel button is clicked. Allows you to use error-handling procedures to recognize that Cancel was clicked.

**DefaultExt** Sets the default extension of a file name if a file is listed without an extension.

**DialogTitle** The string appearing in the title bar of the dialog box. Default is Save As. In the example, the DialogTitle is Save As Example.

**FileName** Sets the initial file name that appears in the File name box. After the dialog box is closed, this property can be read to determine the name of the selected file.

**Filter** Used to restrict the filenames that appear in the file list box.

**FilterIndex** Indicates which filter component is default.

**Flags** Values that control special features of the dialog box (see Appendix II).

The Save File box is commonly configured in one of two ways. If a file is being saved for the first time, the **Save As** configuration, with some default name in the FileName property, is used. In the **Save** configuration, we assume a file has been previously opened with some name. Hence, when saving the file again, that same name should appear in the FileName property. You've seen both configuration types before.

When the user closes the Save File box, you should check the returned file name to make sure it meets the specifications your application requires before you try to save the file. Be especially aware of whether the user changed the file extension to something your application does not allow.

### Quick Example: The Save As Dialog Box

1. We'll just modify the Open example a bit. Change the **DialogTitle** property of the common dialog control to "**Save As Example**" and set the **DefaultExt** property equal to ".bmp".
2. In the **cmdDisplay\_Click** procedure, change the method to **ShowSave** (opens Save As box).
3. Save the application (saved as **QExampleSave** in **LearnVB6/VB Code/Class 4** folder) and run it. Try typing names without extensions and note how **.bmp** is added to them. Notice you can also select file names by double-clicking them or using the **OK** button. Again, the **Cancel** button is not trapped, so it has the same effect as clicking **OK**.

## Chapter4 Error-Handling, Debugging and File Input/Output

### 4.1.ErrorTypes

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

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

• Syntax errors occur when you mistype a command or leave out an expected phrase or argument. Visual Basic detects these errors as they occur and even provides help in correcting them. You cannot run a Visual Basic program until all syntax errors have



been corrected.

- Run-time errors are usually beyond your program's control. Examples include: when a variable takes on an unexpected value (divide by zero), when a drive door is left open, or when a file is not found. Visual Basic allows you to trap such errors and make attempts to correct them.

- Logic errors are the most difficult to find. With logic errors, the program will usually run, but will produce incorrect or unexpected results. The Visual Basic debugger is an aid in detecting logic errors.

- Some ways to minimize errors:

- Design your application carefully. More design time means less debugging time.
- Use comments where applicable to help you remember what you were trying to do.
- Use consistent and meaningful naming conventions for your variables, objects, and procedures.

- Run-time errors are trappable. That is, Visual Basic recognizes an error has occurred and enables you to trap it and take corrective action. If an error occurs and is not trapped, your program will usually end in a rather unceremonious manner.

- Error trapping is enabled with the On Error statement:

On Error GoTo errlabel

Yes, this uses the dreaded GoTo statement! Any time a run-time error occurs following this line, program control is transferred to the line labeled errlabel. Recall a labeled line is simply a line with the label followed by a colon (:).

- The best way to explain how to use error trapping is to look at an outline of an example procedure with error trapping.

Sub SubExample()

[Declare variables, ...]

On Error GoTo HandleErrors

[Procedure code]

Exit Sub

HandleErrors:

Error handling code]

End Sub



Once you have set up the variable declarations, constant definitions, and any other procedure preliminaries, the On Error statement is executed to enable error trapping. Your normal procedure code follows this statement. The error handling code goes at the end of the procedure, following the HandleErrors statement label. This is the code that is executed if an error is encountered anywhere in the Sub procedure. Note you must exit (with Exit Sub) from the code before reaching the HandleErrors line to avoid inadvertent execution of the error handling code.

- Since the error handling code is in the same procedure where an error occurs, all variables in that procedure are available for possible corrective action. If at some time in your procedure, you want to turn off error trapping, that is done with the following statement:

On Error GoTo 0

- Once a run-time error occurs, we would like to know what the error is and attempt to fix it. This is done in the error handling code.
- Visual Basic offers help in identifying run-time errors. The Err object returns, in its Number property (Err.Number), the number associated with the current error condition. (The Err function has other useful properties that we won't cover here - consult on-line help for further information.) The Error() function takes this error number as its argument and returns a string description of the error. Consult on-line help for Visual Basic run-time error numbers and their descriptions.
- Once an error has been trapped and some action taken, control must be returned to your application. That control is returned via the Resume statement. There are three options:

**Resume** Lets you retry the operation that caused the error. That is, control is returned to the line where the error occurred. This could be dangerous in that, if the error has not been corrected (via code or by the user), an infinite loop between the error handler and the procedure code may result.

**Resume Next** Program control is returned to the line immediately following the line where the error occurred.

**Resume label** Program control is returned to the line labeled label.

- Be careful with the Resume statement. When executing the error handling portion of the code and the end of the procedure is encountered before a Resume, an error occurs. Likewise, if a Resume is encountered outside of the error handling portion of the code, an error occurs.

Development of an adequate error handling procedure is application dependent. You need to know what type of errors you are looking for and what corrective actions must be taken if these errors are encountered. For example, if a 'divide by zero' is found, you need to decide whether to skip the operation or do something to reset the offending denominator.

- What we develop here is a generic framework for an error handling procedure. It

simply informs the user that an error has occurred, provides a description of the error, and allows the user to Abort, Retry, or Ignore. This framework is a good starting point for designing custom error handling for your applications.

- The generic code (begins with label HandleErrors) is:

HandleErrors:

```
Select Case MsgBox(Error(Err.Number), vbCritical + vbAbortRetryIgnore, "Error  
Number" + Str(Err.Number))
```

```
Case vbAbort
```

```
    Resume ExitLine
```

```
Case vbRetry
```

```
    Resume
```

```
Case vbIgnore
```

```
    Resume Next
```

```
End Select
```

```
ExitLine:
```

```
Exit Sub
```

## 4.2.General Error Handling Procedure

Let's look at what goes on here. First, this routine is only executed when an error occurs. A message box is displayed, using the Visual Basic provided error description [Error(Err.Number)] as the message, uses a critical icon along with the Abort, Retry, and Ignore buttons, and uses the error number [Err.Number] as the title. This message box returns a response indicating which button was selected by the user.

If **Abort** is selected, we simply exit the procedure. (This is done using a Resume to the line labeled ExitLine. Recall all error trapping must be terminated with a Resume statement of some kind.)

If **Retry** is selected, the offending program line is retried (in a real application, you or the user would have to change something here to correct the condition causing the error).

If **Ignore** is selected, program operation continues with the line following the error causing line.

- To use this generic code in an existing procedure, you need to do three things:
  1. Copy and paste the error handling code into the end of your procedure.
  2. Place an Exit Sub line immediately preceding the HandleErrors labeled line.
  3. Place the line, On Error GoTo HandleErrors, at the beginning of your procedure.

For example, if your procedure is the SubExample seen earlier, the modified code will look like this:

```

Sub SubExample()
    . [Declare variables, ...]
    .
    On Error GoTo HandleErrors
    . [Procedure code]
    .
Exit Sub
HandleErrors:
Select Case MsgBox(Error(Err.Number), vbCritical + vbAbortRetryIgnore, "Error
Number" + Str(Err.Number))
Case vbAbort
    Resume ExitLine
Case vbRetry
    Resume
Case vbIgnore
    Resume Next

End Select
ExitLine:
Exit Sub
End Sub

```

Again, this is a very basic error-handling routine. You must determine its utility in your applications and make any modifications necessary. Specifically, you need code to clear error conditions before using the Retry option.

- One last thing. Once you've written an error handling routine, you need to test it to make sure it works properly. But, creating run-time errors is sometimes difficult and perhaps dangerous. Visual Basic comes to the rescue! The Visual Basic Err object has a method (Raise) associated with it that simulates the occurrence of a run-time error. To cause an error with value Number, use:

```
Err.Raise Number
```

- We can use this function to completely test the operation of any error handler we write. Don't forget to remove the Raise statement once testing is completed, though! And, to really get fancy, you can also use Raise to generate your own 'application-defined' errors. There are errors specific to your application that you want to trap.

- To clear an error condition (any error, not just ones generated with the Raise method), use the method Clear:

```
Err.Clear
```

### Example-Simple Error Trapping



1. Start a new project. Add a text box and a command button.
2. Set the properties of the form and each control:

**Form1:**

BorderStyle - 1-Fixed Single  
Caption - Error Generator  
Name frmError

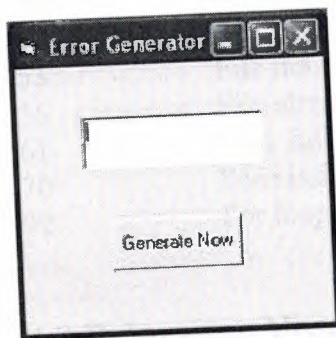
**Command1:**

Caption - Generate Error  
Default - True  
Name - cmdGenError

**Text1:**

Name - txtError  
Text - [Blank]

The form should look something like this:



3. Attach this code to the cmdGenError\_Click event.

```
Private Sub cmdGenError_Click()
    On Error GoTo HandleErrors
    Err.Raise Val(txtError.Text)
    Err.Clear
    Exit Sub
```

```
HandleErrors:
    Select Case MsgBox(Error(Err.Number), vbCritical + vbAbortRetryIgnore, "Error
    Number" + Str(Err.Number))
```

```
    Case vbAbort
        Resume ExitLine
    Case vbRetry
        Resume
    Case vbIgnore
        Resume Next
    End Select
```

```
ExitLine:
```

Exit Sub  
End Sub

In this code, we simply generate an error using the number input in the text box. The generic error handler then displays a message box which you can respond to in one of three ways.

4. Save your application. Try it out using some of these typical error numbers (or use numbers found with on-line help). Notice how program control changes depending on which button is clicked.

#### **Error Number    Error Description**

6	Overflow
9	Subscript out of range
11	Division by zero
13	Type mismatch
16	Expression too complex
20	Resume without error
52	Bad file name or number
53	File not found
55	File already open
61	Disk full
70	Permission denied
92	For loop not initialized

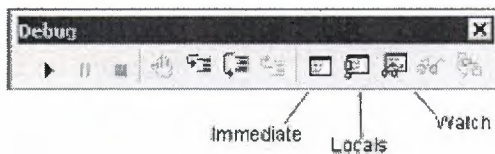
### **4.3. Debugging Visual Basic programs**

- We now consider the search for, and elimination of, logic errors. These are errors that don't prevent an application from running, but cause incorrect or unexpected results. Visual Basic provides an excellent set of debugging tools to aid in this search.

- Debugging a code is an art, not a science. There are no prescribed processes that you can follow to eliminate all logic errors in your program. The usual approach is to eliminate them as they are discovered.

- What we'll do here is present the debugging tools available in the Visual Basic environment (several of which appear as buttons on the toolbar) and describe their use with an example. You, as the program designer, should select the debugging approach and tools you feel most comfortable with.

- The interface between your application and the debugging tools is via three different debug windows: the Immediate Window, the Locals Window, and the Watch Window. These windows can be accessed from the View menu (the Immediate Window can be accessed by pressing Ctrl+G). Or, they can be selected from the Debug Toolbar (accessed using the Toolbars option under the View menu):



- All debugging using the debug windows is done when your application is in break mode. You can enter break mode by setting breakpoints, pressing Ctrl+Break, or the program will go into break mode if it encounters an untrapped error or a Stop statement.

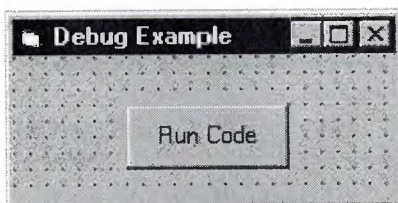
- Once in break mode, the debug windows and other tools can be used to:

1. Determine values of variables
2. Set breakpoints
3. Set watch variables and expressions
4. Manually control the application
5. Determine which procedures have been called
6. Change the values of variables and properties

### Example - Debugging

1. Unlike other examples, we'll do this one as a group. It will be used to demonstrate use of the debugging tools.

2. The example simply has a form with a single command button. The button is used to execute some code. We won't be real careful about proper naming conventions and such in this example.



3. The code attached to this button's Click event is a simple loop that evaluates a function at several values.

```
Private Sub Command1_Click()
Dim X As Integer, Y As Integer
X = 0
Do
Y = Fcn(X)
X = X + 1
Loop While X <= 20
End Sub
```

This code begins with an X value of 0 and computes the Y value using the general integer function Fcn. It then increments X by 1 and repeats the Loop. It continues looping While X is less than or equal to 20. The function Fcn is computed using:



```
Function Fcn(X As Integer) As Integer
Fcn = CInt(0.1 * X ^ 2)
End Function
```

Admittedly, this code doesn't do much, especially without any output, but it makes a good example for looking at debugger use. Set up the application and get ready to try debugging

#### 4.4.Using the Debugging Tools

- There are several debugging tools available for use in Visual Basic. Access to these tools is provided with both menu options and buttons on the Debug toolbar. These tools include breakpoints, watch points, calls, step into, step over, and step out.
- The simplest tool is the use of direct prints to the immediate window.
- **Printing to the Immediate Window:**



You can print directly to the immediate window while an application is running. Sometimes, this is all the debugging you may need. A few carefully placed print statements can sometimes clear up all logic errors, especially in small applications.

To print to the immediate window, use the Print method:

Debug.Print [List of variables separated by commas or semi-colons]

- Debug.Print Example:

1. Place the following statement in the Command1\_Click procedure after the line calling the general procedure Fcn:

```
Debug.Print X; Y
```

and run the application.

2. Examine the immediate window. Note how, at each iteration of the loop, the program prints the value of X and Y. You could use this information to make sure X is incrementing correctly and that Y values look acceptable.
3. Remove the Debug.Print statement.

- **Breakpoints:**



In the above examples, the program ran to completion before we could look at the debug window. In many applications, we want to stop the application while it is running, examine variables and then continue running. This can be done with breakpoints.

A breakpoint is a line in the code where you want to stop (temporarily) the execution of the program, that is force the program into break mode. To set a breakpoint, put the cursor in the line of code you want to break on. Then, press <F9> or click the Breakpoint button on the toolbar or select Toggle Breakpoint from the Debug menu. The line will be highlighted.

When you run your program, Visual Basic will stop when it reaches lines with breakpoints and allow you to use the immediate window to check variables and expressions. To continue program operation after a breakpoint, press <F5>, click the Run button on the toolbar, or choose Start from the Run menu.

You can also change variable values using the immediate window. Simply type a valid Basic expression. This can sometimes be dangerous, though, as it may change program operation completely.

- Breakpoint Example:

1. Set a breakpoint on the  $X = X + 1$  line in the sample program. Run the program.
2. When the program stops, display the immediate window and type the following line:

Print X;Y

3. The values of these two variables will appear in the debug window. You can use a question mark (?) as shorthand for the command Print, if you'd like. Restart the application. Print the new variable values.
4. Try other breakpoints if you have time. Once done, all breakpoints can be cleared by Ctrl+Shift+<F9> or by choosing Clear All Breakpoints from the Debug menu. Individual breakpoints can be toggled using <F9> or the Breakpoint button on the toolbar.

- Viewing Variables in the Locals Window:



The locals window shows the value of any variables within the scope of the current procedure. As execution switches from procedure to procedure, the contents of this window changes to reflect only the variables applicable to the current procedure. Repeat the above example and notice the values of X and Y also appear in the locals window.

- Watch Expressions:



The Add Watch option on the Debug menu allows you to establish watch expressions for your application. Watch expressions can be variable values or logical expressions you want to view or test. Values of watch expressions are displayed in the watch window.

In break mode, you can use the Quick Watch button on the toolbar to add watch expressions you need. Simply put the cursor on the variable or expression you want to add to the watch list and click the Quick Watch button.

Watch expressions can be edited using the Edit Watch option on the Debug menu.

- **Watch Expression Example:**

1. Set a breakpoint at the  $X = X + 1$  line in the example.
2. Set a watch expression for the variable X. Run the application. Notice X appears in the watch window. Every time you re-start the application, the value of X changes.
3. At some point in the debug procedure, add a quick watch on Y. Notice it is now in the watch window.
4. Clear the breakpoint. Add a watch on the expression:  $X = Y$ . Set Watch Type to 'Break When Value Is True.' Run the application. Notice it goes into break mode and displays the watch window whenever  $X = Y$ . Delete this last watch expression.

- **Call Stack:**



Selecting the Call Stack button from the toolbar (or pressing Ctrl+L or selecting Call Stack from the View menu) will display all active procedures, that is those that have not been exited.

Call Stack helps you unravel situations with nested procedure calls to give you some idea of where you are in the application.

- **Call Stack Example:**

1. Set a breakpoint on the  $Fcn = Cint()$  line in the general function procedure. Run the application. It will break at this line.
2. Press the Call Stack button. It will indicate you are currently in the Fcn procedure which was called from the Command1\_Click procedure. Clear the breakpoint.

- **Single Stepping (Step Into):**



While at a breakpoint, you may execute your program one line at a time by pressing <F8>, choosing the Step Into option in the Debug menu, or by clicking the Step Into button on the toolbar.

This process is single stepping. It allows you to watch how variables change (in the locals window) or how your form changes, one step at a time.

You may step through several lines at a time by using Run To Cursor option. With this option, click on a line below your current point of execution. Then press Ctrl+<F8> (or



choose Run To Cursor in the Debug menu). the program will run through every line up to the cursor location, then stop.

- Step Into Example:

1. Set a breakpoint on the Do line in the example. Run the application.
2. When the program breaks, use the Step Into button to single step through the program.
3. At some point, put the cursor on the Loop While line. Try the Run To Cursor option (press Ctrl+<F8>).

- Procedure Stepping (Step Over):



While single stepping your program, if you come to a procedure call you know functions properly, you can perform procedure stepping. This simply executes the entire procedure at once, rather than one step at a time.

To move through a procedure in this manner, press Shift+<F8>, choose Step Over from the Debug menu, or press the Step Over button on the toolbar.

- Step Over Example:

1. Run the previous example. Single step through it a couple of times.
2. One time through, when you are at the line calling the Fcn function, press the Step Over button. Notice how the program did not single step through the function as it did previously.

- Function Exit (Step Out):



While stepping through your program, if you wish to complete the execution of a function you are in, without stepping through it line-by-line, choose the Step Out option. The function will be completed and you will be returned to the procedure accessing that function.

To perform this step out, press Ctrl+Shift+<F8>, choose Step Out from the Debug menu, or press the Step Out button on the toolbar. Try this on the previous example.

## 4.5. Debugging Strategies

- We've looked at each debugging tool briefly. Be aware this is a cursory introduction. Use the on-line help to delve into the details of each tool described. Only through lots of use and practice can you become a proficient debugger. There are some guidelines to doing a good job, though.

- My first suggestion is: keep it simple. Many times, you only have one or two bad lines of code. And you, knowing your code best, can usually quickly narrow down the areas

with bad lines. Don't set up some elaborate debugging procedure if you haven't tried a simple approach to find your error(s) first. Many times, just a few intelligently-placed `Debug.Print` statements or a few examinations of the immediate and locals windows can solve your problem.

- A tried and true approach to debugging can be called Divide and Conquer. If you're not sure where your error is, guess somewhere in the middle of your application code. Set a breakpoint there. If the error hasn't shown up by then, you know it's in the second half of your code. If it has shown up, it's in the first half. Repeat this division process until you've narrowed your search.
- And, of course, the best debugging strategy is to be careful when you first design and write your application to minimize searching for errors later.

## **4.6.Sequential Files**

- In many applications, it is helpful to have the capability to read and write information to a disk file. This information could be some computed data or perhaps information loaded into a Visual Basic object.
- Visual Basic supports two primary file formats: sequential and random access. We first look at sequential files.
- A sequential file is a line-by-line list of data. You can view a sequential file with any text editor. When using sequential files, you must know the order in which information was written to the file to allow proper reading of the file.
- Sequential files can handle both text data and variable values. Sequential access is best when dealing with files that have lines with mixed information of different lengths. I use them to transfer data between applications.

## **4.7.Sequential File Output(Variables)**

- We first look at writing values of variables to sequential files. The first step is to Open a file to write information to. The syntax for opening a sequential file for output is:

`Open SeqFileName For Output As #N`

where `SeqFileName` is the name of the file to open and `N` is an integer file number. The filename must be a complete path to the file.

- When done writing to the file, Close it using:

`Close N`

Once a file is closed, it is saved on the disk under the path and filename used to open the file.

- Information is written to a sequential file one line at a time. Each line of output requires a separate Basic statement.

- There are two ways to write variables to a sequential file. The first uses the Write statement:

Write #N, [variable list]

where the variable list has variable names delimited by commas. (If the variable list is omitted, a blank line is printed to the file.) This statement will write one line of information to the file, that line containing the variables specified in the variable list. The variables will be delimited by commas and any string variables will be enclosed in quotes. This is a good format for exporting files to other applications like Excel.

### Example

```
Dim A As Integer, B As String, C As Single, D As Integer
```

```
.
```

```
Open TestOut For Output As #1
```

```
Write #1, A, B, C
```

```
Write #1, D
```

```
Close 1
```

After this code runs, the file TestOut will have two lines. The first will have the variables A, B, and C, delimited by commas, with B (a string variable) in quotes. The second line will simply have the value of the variable D.

- The second way to write variables to a sequential file is with the Print statement:

Print #N, [variable list]

This statement will write one line of information to the file, that line containing the variables specified in the variable list. (If the variable list is omitted, a blank line will be printed.) If the variables in the list are separated with semicolons (;), they are printed with a single space between them in the file. If separated by commas (,), they are spaced in wide columns. Be careful using the Print statement with string variables. The Print statement does not enclose string variables in quotes, hence, when you read such a variable back in, Visual Basic may have trouble knowing where a string ends and begins. It's good practice to 'tack on' quotes to string variables when using Print.

### Example

```
Dim A As Integer, B As String, C As Single, D As Integer
```

```
.
```

```
Open TestOut For Output As #1
```

```
Print #1, A; Chr(34) + B + Chr(34), C
```

```
Print #1, D
```

```
Close 1
```



After this code runs, the file TestOut will have two lines. The first will have the variables A, B, and C, delimited by spaces. B will be enclosed by quotes [Chr(34)]. The second line will simply have the value of the variable D.

### Quick Example: Writing Variables to Sequential Files

1. Start a new project.
2. Attach the following code to the Form\_Load procedure. This code simply writes a few variables to sequential files.

```
Private Sub Form_Load()  
Dim A As Integer, B As String, C As Single, D As Integer  
A = 5  
B = "Visual Basic"  
C = 2.15  
D = -20  
Open "Test1.Txt" For Output As #1  
Open "Test2.Txt" For Output As #2  
Write #1, A, B, C  
Write #1, D  
Print #2, A, B, C  
Print #2, D  
Close 1  
Close 2  
End Sub
```

3. Run the program. Use a text editor (try the Windows 95 Notepad) to examine the contents of the two files, Test1.Txt and Test2.Txt. They are probably in the Visual Basic main directory. Note the difference in the two files, especially how the variables are delimited and the fact that the string variable is not enclosed in quotes in Test2.Txt. Save the application, if you want to

### 4.8.Sequential File Input(Variables)

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

```
Open SeqFileName For Input As #N
```

where N is an integer file number and SeqFileName is a complete file path. The file is closed using:

```
Close N
```

- The Input statement is used to read in variables from a sequential file. The format is:

```
Input #N, [variable list]
```

The variable names in the list are separated by commas. If no variables are listed, the current line in the file N is skipped.

- Note variables must be read in exactly the same manner as they were written. So,

using our previous example with the variables A, B, C, and D, the appropriate statements are:

```
Input #1, A, B, C
Input #1, D
```

These two lines read the variables A, B, and C from the first line in the file and D from the second line. It doesn't matter whether the data was originally written to the file using Write or Print (i.e. commas are ignored).

### **Quick Example: Reading Variables from Sequential Files**

1. Start a new project or simply modify the previous quick example.
2. Attach the following code to the Form\_Load procedure. This code reads in files created in the last quick example.

```
Private Sub Form_Load()
Dim A As Integer, B As String, C As Single, D As Integer
Open "Test1.Txt" For Input As #1
Input #1, A, B, C
Debug.Print "A="; A
Debug.Print "B="; B
Debug.Print "C="; C
Input #1, D
Debug.Print "D="; D
Close #1
End Sub
```

Note the Debug.Print statements and how you can add some identifiers (in quotes) for printed information.

3. Run the program. Look in the debug window and note the variable values. Save the application, if you want to.
4. Rerun the program using Test2.Txt as in the input file. What differences do you see? Do you see the problem with using Print and string variables? Because of this problem, I almost always use Write (instead of Print) for saving variable information to files. Edit the Test2.Txt file (in Notepad), putting quotes around the words Visual Basic. Rerun the program using this file as input - it should work fine now

### **4.10. Writing and Reading Text Using Sequential Files**

- In many applications, we would like to be able to save text information and retrieve it for later reference. This information could be a text file created by an application or the contents of a Visual Basic text box.

#### **• Writing Text Files:**

To write a sequential text file, we follow the simple procedure: open the file, write the file, close the file. If the file is a line-by-line text file, each line of the file is written to disk using a single Print statement:

```
Print #N, Line
```

where Line is the current line (a text string). This statement should be in a loop that encompasses all lines of the file. You must know the number of lines in your file, beforehand.

If we want to write the contents of the Text property of a text box named txtExample to a file, we use:

```
Print #N, txtExample.Text
```

### **Example**

We have a text box named txtExample. We want to save the contents of the Text property of that box in a file named MyText.ned on the c: drive in the \MyFiles directory. The code to do this is:

```
Open "c:\MyFiles\MyText.ned" For Output As #1
Print #1, txtExample.Text
Close 1
```

The text is now saved in the file for later retrieval.

### **• Reading Text Files:**

To read the contents of a previously-saved text file, we follow similar steps to the writing process: open the file, read the file, close the file. If the file is a text file, we read each individual line with the Line Input command:

```
Line Input #1, Line
```

This line is usually placed in a Do/Loop structure that is repeated until all lines of the file are read in. The EOF() function can be used to detect an end-of-file condition, if you don't know, a priori, how many lines are in the file.

To place the contents of a file opened with number N into the Text property of a text box named txtExample we use the Input function:

```
txtExample.Text = Input(LOF(N), N)
```

This Input function has two arguments: LOF(N), the length of the file opened as N and N, the file number.

### **Example**

We have a file named MyText.ned stored on the c: drive in the \MyFiles directory. We want to read that text file into the text property of a text box named txtExample. The code to do this is:

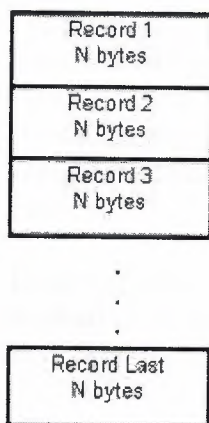
```
Open "c:\MyFiles\MyText.ned" For Input As #1
txtExample.Text = Input(LOF(1), 1)
Close 1
```



The text in the file will now be displayed in the text box.

#### 4.11.Random Access Files

- Note that to access a particular data item in a sequential file, you need to read in all items in the file prior to the item of interest. This works acceptably well for small data files of unstructured data, but for large, structured files, this process is time-consuming and wasteful. Sometimes, we need to access data in nonsequential ways. Files which allow nonsequential access are random access files.
- To allow nonsequential access to information, a random access file has a very definite structure. A random access file is made up of a number of records, each record having the same length (measured in bytes). Hence, by knowing the length of each record, we can easily determine (or the computer can) where each record begins. The first record in a random access file is Record 1, not 0 as used in Visual Basic arrays. Each record is usually a set of variables, of different types, describing some item. The structure of a random access file is:



- A good analogy to illustrate the differences between sequential files and random access files are cassette music tapes and compact discs. To hear a song on a tape (a sequential device), you must go past all songs prior to your selection. To hear a song on a CD (a random access device), you simply go directly to the desired selection. One difference here though is we require all of our random access records to be the same length - not a good choice on CD's!
- To write and read random access files, we must know the record length in bytes. Some variable types and their length in bytes are:

Type	Length (Bytes)
Integer	2
Long	4
Single	4
Double	8
String	1 byte per character

So, for every variable that is in a file's record, we need to add up the individual variable length's to obtain the total record length. To ease this task, we introduce the idea of user-defined variables.

#### 4.12. User-Defined Variables

- Data used with random access files is most often stored in user-defined variables. These data types group variables of different types into one assembly with a single, user-defined type associated with the group. Such types significantly simplify the use of random access files.
- The Visual Basic keyword `Type` signals the beginning of a user-defined type declaration and the words `End Type` signal the end. An example best illustrates establishing a user-defined variable. Say we want to use a variable that describes people by their name, their city, their height, and their weight. We would define a variable of `Type Person` as follows:

```
Type Person
Name As String
City As String
Height As Integer
Weight As Integer
End Type
```

These variable declarations go in the same code areas as normal variable declarations, depending on desired scope. At this point, we have not reserved any storage for the data. We have simply described to Visual Basic the layout of the data.

- To create variables with this newly defined type, we employ the usual `Dim` statement. For our `Person` example, we would use:

```
Dim Lou As Person
Dim John As Person
Dim Mary As Person
```

And now, we have three variables, each containing all the components of the variable type `Person`. To refer to a single component within a user-defined type, we use the dot-notation:

`VarName.Component`

As an example, to obtain Lou's Age, we use:

```
Dim AgeValue as Integer
```

```
AgeValue = Lou.Age
```

Note the similarity to dot-notation we've been using to set properties of various Visual Basic tools.

### 4.13. Writing and Reading Random Access Files

- We look at writing and reading random access files using a user-defined variable. For other variable types, refer to Visual Basic on-line help. To open a random access file named `RanFileName`, use:

```
Open RanFileName For Random As #N Len = RecordLength
```

where `N` is an available file number and `RecordLength` is the length of each record. Note you don't have to specify an input or output mode. With random access files, as long as they're open, you can write or read to them.

- To close a random access file, use:

```
Close N
```

- As mentioned previously, the record length is the sum of the lengths of all variables that make up a record. A problem arises with String type variables. You don't know their lengths ahead of time. To solve this problem, Visual Basic lets you declare fixed lengths for strings. This allows you to determine record length. If we have a string variable named `StrExample` we want to limit to 14 characters, we use the declaration:

```
Dim StrExample As String * 14
```

Recall each character in a string uses 1 byte, so the length of such a variable is 14 bytes.

- Recall our example user-defined variable type, `Person`. Let's revisit it, now with restricted string lengths:

```
Type Person  
Name As String * 40  
City As String * 35  
Height As Integer  
Weight As Integer  
End Type
```

The record length for this variable type is 79 bytes ( $40 + 35 + 2 + 2$ ). To open a file named `PersonData` as File #1, with such records, we would use the statement:

```
Open PersonData For Random As #1 Len = 79
```

- The `Get` and `Put` statements are used to read from and write to random access files, respectively. These statements read or write one record at a time. The syntax for these statements is simple:

```
Get #N, [RecordNumber], variable
```



Put #N, [RecordNumber], variable

The Get statement reads from the file and stores data in the variable, whereas the Put statement writes the contents of the specified variable to the file. In each case, you can optionally specify the record number. If you do not specify a record number, the next sequential position is used.

- The variable argument in the Get and Put statements is usually a single user-defined variable. Once read in, you obtain the component parts of this variable using dot-notation. Prior to writing a user-defined variable to a random access file, you 'load' the component parts using the same dot-notation.
- There's a lot more to using random access files; we've only looked at the basics. Refer to your Visual Basic documentation and on-line help for further information. In particular, you need to do a little cute programming when deleting records from a random access file or when 'resorting' records.

#### **4.14.Using the Open and Save Common Dialog Boxes**

- Note to both write and read sequential and random access files, we need a file name for the Open statement. To ensure accuracy and completeness, it is suggested that common dialog boxes be used to get this file name information from the user. I'll provide you with a couple of code segments that do just that. Both segments assume you have a common dialog box on your form named cdlFiles, with the CancelError property set equal to True. With this property True, an error is generated by Visual Basic when the user presses the Cancel button in the dialog box. By trapping this error, it allows an elegant exit from the dialog box when canceling the operation is desired.
- The code segment to obtain a file name (MyFileName with default extension Ext) for opening a file to read is:

```
Dim MyFileName As String, Ext As String
.
.
cdlFiles.Filter = "Files (*. " + Ext + ")|*." + Ext
cdlFiles.DefaultExt = Ext
cdlFiles.DialogTitle = "Open File"
cdlFiles.Flags = cdlOFNFileMustExist + cdlOFNPathMustExist
On Error GoTo No_Open
cdlFiles.ShowOpen
MyFileName = cdlFiles.filename
.
.
Exit Sub
No_Open:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

A few words on what's going on here. First, some properties are set such that only files with Ext (a three letter string variable) extensions are displayed (Filter property), the default extension is Ext (DefaultExt property), the title bar is set (DialogTitle property), and some Flags are set to insure the file and path exist (see Appendix II for more common dialog flags).

Error trapping is enabled to trap the Cancel button. Finally, the common dialog box is displayed and the filename property returns with the desired name. That name is put in the string variable MyFileName. What you do after obtaining the file name depends on what type of file you are dealing with. For sequential files, you would open the file, read in the information, and close the file. For random access files, we just open the file here. Reading and writing to/from the file would be handled elsewhere in your coding.

- The code segment to retrieve a file name (MyFileName) for writing a file is:

```
Dim MyFileName As String, Ext As String
.
.
cdlFiles.Filter = "Files (*. " + Ext + ")*.*" + Ext
cdlFiles.DefaultExt = Ext
cdlFiles.DialogTitle = "Save File"
cdlFiles.Flags = cdlOFNOverwritePrompt + cdlOFNPathMustExist
On Error GoTo No_Save
cdlFiles.ShowSave
MyFileName = cdlFiles.filename
.
.
Exit Sub
No_Save:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

Note this code is essentially the same used for an Open file name. The Flags property differs slightly. The user is prompted if a previously saved file is selected for overwrite. After obtaining a valid file name for a sequential file, we would open the file for output, write the file, and close it. For a random access file, things are trickier.

If we want to save the file with the same name we opened it with, we simply close the file. If the name is different, we must open a file (using a different number) with the new name, write the complete random access file, then close it. Like I said, it's trickier.

- We use both of these code segments in the final example where we write and read sequential files.

### **Example - Note Editor - Reading and Saving Text Files**

We now add the capability to read in and save the contents of the text box in the Note Editor application from last class. Load that application. Add a common dialog box to your form. Name it cdlFiles and set the CancelError property to True.

Modify the File menu (use the Menu Editor and the Insert button) in your application, such that Open and Save options are included. The File menu should now read:

- File
- New
- Open
- Save
- Exit

Properties for these new menu items should be:

Caption	Name	Shortcut
---------	------	----------

&Open	mnuFileOpen	[None]
-------	-------------	--------

&Save	mnuFileSave	[None]
-------	-------------	--------

The two new menu options need code. Attach this code to the mnuFileOpen\_Click event. This uses a modified version of the code segment seen previously. We assign the extension ned to our note editor files.

```
Private Sub mnuFileOpen_Click()  
cdlFiles.Filter = "Files (*.ned)|*.ned"  
cdlFiles.DefaultExt = "ned"  
cdlFiles.DialogTitle = "Open File"  
cdlFiles.Flags = cdIOFNFileMustExist + cdIOFNPathMustExist  
On Error GoTo No_Open  
cdlFiles.ShowOpen  
Open cdlFiles.filename For Input As #1  
txtEdit.Text = Input(LOF(1), 1)  
Close 1  
Exit Sub  
No_Open:  
Resume ExitLine  
ExitLine:  
  
Exit Sub  
End Sub
```

And for the mnuFileSave\_Click procedure, use this code. Much of this can be copied from the previous procedure.

```
Private Sub mnuFileSave_Click()  
cdlFiles.Filter = "Files (*.ned)|*.ned"  
cdlFiles.DefaultExt = "ned"  
cdlFiles.DialogTitle = "Save File"  
cdlFiles.Flags = cdIOFNOverwritePrompt + cdIOFNPathMustExist  
On Error GoTo No_Save  
cdlFiles.ShowSave  
Open cdlFiles.filename For Output As #1  
Print #1, txtEdit.Text  
Close 1  
Exit Sub
```



```
No_Save:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

Each of these procedures is similar. The dialog box is opened and, if a filename is returned, the file is read/written. If Cancel is pressed, no action is taken. These routines can be used as templates for file operations in other applications.

Save your application. Run it and test the Open and Save functions. Note you have to save a file before you can open one. Check for proper operation of the Cancel button in the common dialog box.

If you have the time, there is one major improvement that should be made to this application. Notice that, as written, only the text information is saved, not the formatting (bold, italic, underline, size). Whenever a file is opened, the text is displayed based on current settings. It would be nice to save formatting information along with the text. This can be done, but it involves a fair amount of reprogramming. Suggested steps:

A. Add lines to the **mnuFileSave\_Click** routine that write the text box properties FontBold, FontItalic, FontUnderline, and FontSize to a separate sequential file. If your text file is named TxtFile.ned, I would suggest naming the formatting file TxtFile.fmt. Use string functions to put this name together. That is, chop the ned extension off the text file name and tack on the fmt extension. You'll need the Len() and Left() functions.

B. Add lines to the **mnuFileOpen\_Click** routine that read the text box properties FontBold, FontItalic, FontUnderline, and FontSize from your format sequential file. You'll need to define some intermediate variables here because Visual Basic won't allow you to read properties directly from a file. You'll also need logic to set/reset any check marks in the menu structure to correspond to these input properties.

C. Add lines to the **mnuFileNew\_Click** procedure that, when the user wants a new file, reset the text box properties FontBold, FontItalic, FontUnderline, and FontSize to their default values and set/reset the corresponding menu check marks.

D. Try out the modified application. Make sure every new option works as it should.

Actually, there are 'custom' tools (we'll look at custom tools in Class 10) that do what we are trying to do with this modification, that is save text box contents with formatting information. Such files are called 'rich text files' or rtf files. You may have seen these before when transferring files from one word processor to another.

Another thing you could try: Modify the message box that appears when you try to Exit. Make it ask if you wish to save your file before exiting - provide Yes, No, Cancel buttons. Program the code corresponding to each possible response. Use calls to existing procedures, if possible.

## **Chapter5 Database Access Management**

### **5.1.Database Structure and Terminology**

- In simplest terms, a database is a collection of information. This collection is stored in well-defined tables, or matrices.

- The rows in a database table are used to describe similar items. The rows are referred to as database records. In general, no two rows in a database table will be alike.
- The columns in a database table provide characteristics of the records. These characteristics are called database fields. Each field contains one specific piece of information. In defining a database field, you specify the data type, assign a length, and describe other attributes.
- Here is a simple database example:

Field				
ID No	Name	Date of Birth	Height	Weight
1	Bob Jones	01/04/58	72	170
2	Mary Rodgers	11/22/61	65	125
3	Sue Williams	08/11/57	68	130
Record				
Table				

In this database table, each record represents a single individual. The fields (descriptors of the individuals) include an identification number (ID No), Name, Date of Birth, Height, and Weight.

- Most databases use indexes to allow faster access to the information in the database. Indexes are sorted lists that point to a particular row in a table. In the example just seen, the ID No field could be used as an index.
- A database using a single table is called a flat database. Most databases are made up of many tables. When using multiple tables within a database, these tables must have some common fields to allow cross-referencing of the tables. The referral of one table to another via a common field is called a relation. Such groupings of tables are called relational databases.
- In our first example, we will use a sample database that comes with Visual Basic. This database (BIBLIO.MDB) is found in the main Visual Basic directory (try c:\Program Files\Microsoft Visual Studio\VB98). It is a database of books about computers. Let's look at its relational structure. The BIBLIO.MDB database is made up of four tables:

**Authors Table (6246 Records, 3 Fields)**

Au_ID	Author	Year Born

**Publishers Table** (727 Records, 10 Fields)

PubID	Name	Company	Fax	Comments

**Title Author Table** (16056 Records, 2 Fields)

ISBN	Au_ID

**Titles Table** (8569 Records, 8 Fields)

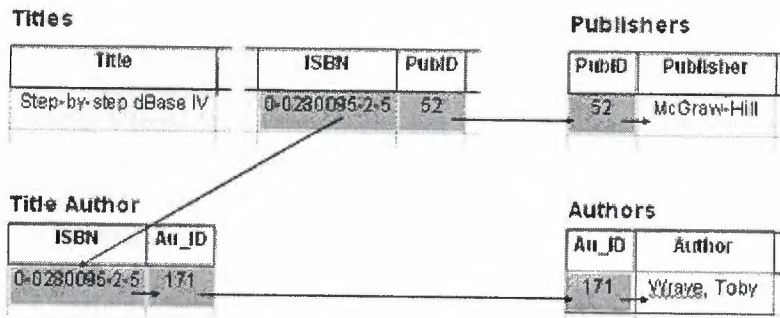
Title	Year Pub	ISBN	PubID	Comments

The **Authors** table consists of author identification numbers, the author's name, and the year born. The **Publishers** table has information regarding book publishers. Some of the fields include an identification number, the publisher name, and pertinent phone numbers. The **Title Author** table correlates a book's ISBN (a universal number assigned to books) with an author's identification number. And, the **Titles** table has several fields describing each individual book, including title, ISBN, and publisher identification

Note each table has two types of information: **source** data and **relational** data. Source data is actual information, such as titles and author names. Relational data are references to data in other tables, such as Au\_ID and PubID. In the Authors, Publishers and Title Author tables, the first column is used as the table **index**. In the Titles table, the ISBN value is the **index**.

- Using the relational data in the four tables, we should be able to obtain a complete description of any book title in the database. Let's look at one example:





Here, the book in the **Titles** table, entitled "Step-by-step dBase IV," has an ISBN of 0-0280095-2-5 and a PubID of 52. Taking the PubID into the **Publishers** table, determines the book is published by McGraw-Hill and also allows us to access all other information concerning the publisher. Using the ISBN in the **Title Author** table provides us with the author identification (Au\_ID) of 171, which, when used in the **Authors** table, tells us the book's author is Toby Wraye.

- We can form alternate tables from a database's inherent tables. Such **virtual tables**, or **logical views**, are made using queries of the database. A **query** is simply a request for information from the database tables. As an example with the BIBLIO.MDB database, using pre-defined query languages, we could 'ask' the database to form a table of all authors and books published after 1992, or provide all author names starting with B. We'll look briefly at queries.

- Keeping track of all the information in a database is handled by a **database management system (DBMS)**. They are used to create and maintain databases. Examples of commercial DBMS programs are Microsoft Access, Microsoft FoxPro, Borland Paradox, Borland dBase, and Claris FileMaker. We can also use Visual Basic to develop a DBMS. Visual Basic shares the same 'engine' used by Microsoft Access, known as the **Jet engine**. In this class, we will see how to use Visual Basic to access data, display data, and perform some elementary management operations.

## 5.2.ADO(ActiveX Data Object) data control

- The ADO (ActiveX Data Object) data control is the primary interface between a Visual Basic application and a database. It can be used without writing any code at all! Or, it can be a central part of a complex database management system. This icon may not appear in your Visual Basic toolbox. If it doesn't, select Project from the main menu, then click Components. The Components window will appear. Select Microsoft ADO Data Control, then click OK. The control will be added to your toolbox.

- As mentioned in Review and Preview, previous versions of Visual Basic used another data control. That control is still included with Visual Basic 6.0 (for backward compatibility) and has as its icon:



Make sure you are not using this data control for the work in this class. This control is suitable for small databases. You might like to study it on your own.

- The data control (or tool) can access databases created by several other programs besides Visual Basic (or Microsoft Access). Some other formats supported include Btrieve, dBase, FoxPro, and Paradox databases.
- The data control can be used to perform the following tasks:
  1. Connect to a database.
  2. Open a specified database table.
  3. Create a virtual table based on a database query.
  4. Pass database fields to other Visual Basic tools, for display or editing. Such tools are bound tools (controls), or data aware.
  5. Add new records or update a database.
  6. Trap any errors that may occur while accessing data.
  7. Close the database.
- Data Control Properties:

**Align** Determines where data control is displayed.

**Caption** Phrase displayed on the data control.

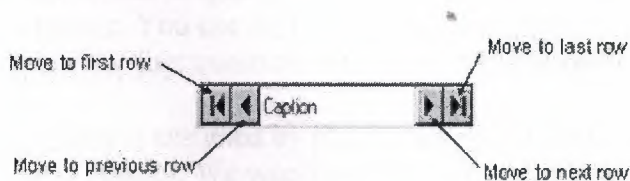
**ConnectionString** Contains the information used to establish a connection to a database.

**LockType** Indicates the type of locks placed on records during editing (default setting makes databases read-only).

**Recordset** A set of records defined by a data control's ConnectionString and RecordSource properties. Run-time only.

**RecordSource** Determines the table (or virtual table) the data control is attached to.

- As a rule, you need one data control for every database table, or virtual table, you need access to. One row of a table is accessible to each data control at any one time. This is referred to as the current record.
- When a data control is placed on a form, it appears with the assigned caption and four arrow buttons:



The arrows are used to navigate through the table rows (records). As indicated, the buttons can be used to move to the beginning of the table, the end of the table, or from record to record.

### 5.3.Data Links

- After placing a data control on a form, you set the **ConnectionString** property. The ADO data control can connect to a variety of database types. There are three ways to connect to a database: using a data link, using an ODBC data source, or using a connection string. In this lesson, we will look only at connection to a Microsoft Access



database using a **data link**. A data link is a file with a **UDL** extension that contains information on database type.

- If your database does not have a data link, you need to create one. This process is best illustrated by example. We will be using the **BIBLIO.MDB** database in our first example, so these steps show you how to create its data link:

1. Open Windows **Explorer**.
2. Open the folder where you will store your data link file.
3. Right-click the right side of Explorer and choose **New**. From the list of files, select **Microsoft Data Link**.
4. Rename the newly created file **BIBLIO.UDL**.
5. Right-click this new **UDL** file and click **Properties**.
6. Choose the **Provider** tab and select **Microsoft Jet 3.51 OLE DB Provider** (an Access database).
7. Click the **Next** button to go to the **Connection** tab.
8. Click the **ellipsis** and use the **Select Access Database** dialog box to choose the **BIBLIO.MDB** file which is in the Visual Basic main folder. Click **Open**.
9. Click **Test Connection**. Then, click **OK** (assuming it passed). The **UDL** file is now created and can be assigned to **ConnectionString**, using the steps below.

- If a data link has been created and exists for your database, click the **ellipsis** that appears next to the **ConnectionString** property. Choose **Use Data Link File**. Then, click **Browse** and find the file. Click **Open**. The data link is now assigned to the property. Click **OK**.

#### 5.4. Assigning Tables

- Once the ADO data control is connected to a database, we need to assign a table to that control. Recall each data control is attached to a single table, whether it is a table inherent to the database or the virtual table we discussed. Assigning a table is done via the **RecordSource** property.

- Tables are assigned by making queries of the database. The language used to make a query is **SQL** (pronounced 'sequel,' meaning structured query language). **SQL** is an English-like language that has evolved into the most widely used database query language. You use **SQL** to formulate a question to ask of the database. The data base 'answers' that question with a new table of records and fields that match your criteria.

- A table is assigned by placing a valid **SQL** statement in the **RecordSource** property of a data control. We won't be learning any **SQL** here. There are many texts on the subject - in fact, many of them are in the **BIBLIO.MDB** database we've been using. Here we simply show you how to use **SQL** to have the data control 'point' to an inherent database table.

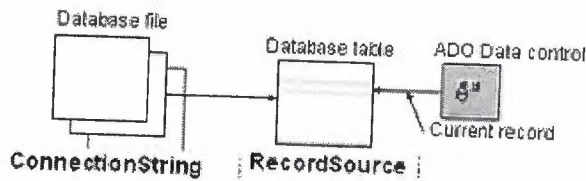
- Click on the **ellipsis** next to **RecordSource** in the **property** box. A **Property Pages** dialog box will appear. In the box marked **Command Text (SQL)**, type this line:

**SELECT \* FROM TableName**



This will select all fields (the \* is a wildcard) from a table named **TableName** in the database. Click **OK**.

- Setting the **RecordSource** property also establishes the **Recordset** property, which we will see later is a very important property.
- In summary, the relationship between the **data control** and its two primary properties (**ConnectionString** and **RecordSource**) is:



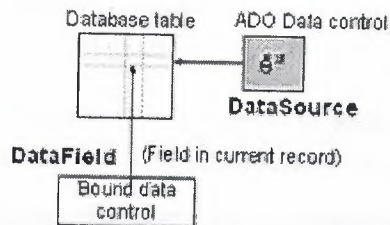
### 5.5.Bound Data Tools

- Most of the Visual Basic tools we've studied can be used as bound, or data-aware, tools (or controls). That means, certain tool properties can be tied to a particular database field. To use a bound control, one or more data controls must be on the form.
  - Some bound data tools are:
    - Label** - Can be used to provide display-only access to a specified text data field.
    - Text Box** - Can be used to provide read/write access to a specified text data field. Probably, the most widely used data bound tool.
    - Check Box** - Used to provide read/write access to a Boolean field.
    - Combo Box** - Can be used to provide read/write access to a text data field.
    - List Box** - Can be used to provide read/write access to a text data field.
    - Picture Box** - Used to display a graphical image from a bitmap, icon, or metafile on your form. Provides read/write access to a image/binary data field.
    - Image Box** - Used to display a graphical image from a bitmap, icon, or metafile on your form (uses fewer resources than a picture box). Provides read/write access to a image/binary data field.
  - There are also three 'custom' data aware tools, the **DataCombo** (better than using the bound combo box), **DataList** (better than the bound list box), and **DataGrid** tools, we will look at later.
  - Bound Tool Properties:
    - DataChanged** - Indicates whether a value displayed in a bound control has changed.
    - DataField** - Specifies the name of a field in the table pointed to by the respective data control.
    - DataSource** - Specifies which data control the control is bound to.
- If the data in a data-aware control is changed and then the user changes focus to another control or tool, the database will automatically be updated with the new data (assuming LockType is set to allow an update).
- To make using bound controls easy, follow these steps (in order listed) in placing the controls on a form:
1. Draw the bound control on the same form as the data control to which it will be bound.
  2. Set the **DataSource** property. Click on the drop-down arrow to list the data controls on your form. Choose one.

3. Set the **DataField** property. Click on the drop-down arrow to list the fields associated with the selected data control records. Make your choice.
4. Set all other properties, as required.

By following these steps in order, we avoid potential data access errors.

- The relationships between the bound data control and the data control are:



### Example - Accessing the Books Database

1. Start a new application. We'll develop a form where we can skim through the books database, examining titles and ISBN values. Place an ADO data control, two label boxes, and two text boxes on the form.
2. If you haven't done so, create a data link for the BIBLIO.MDB database following the steps given under Data Links in these notes.
3. Set the following properties for each control. For the data control and the two text boxes, make sure you set the properties in the order given.

#### Form1:

BorderStyle - 1-Fixed Single  
Caption - Books Database  
Name - frmBooks

#### Adodc1:

Caption - Book Titles  
ConnectionString - BIBLIO.udl (in whatever folder you saved it in - select, don't type)  
RecordSource - SELECT \* FROM Titles  
Name - dtaTitles

#### Label1:

Caption - Title

#### Label2:

Caption - ISBN

#### Text1:

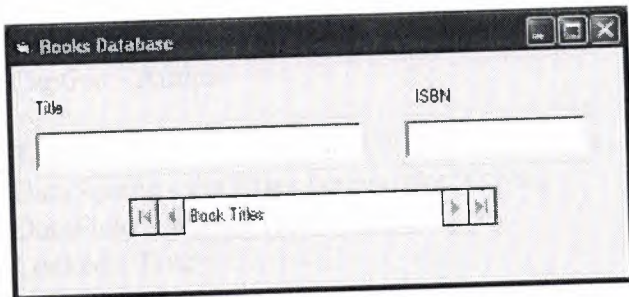
DataSource - dtaTitles (select, don't type)  
DataField - Title (select, don't type)  
Locked - True

MultiLine - True  
Name - txtTitle  
Text - [Blank]

#### Text2:

DataSource - dtaTitles (select, don't type)  
DataField - ISBN (select, don't type)  
Locked - True  
Name - txtISBN  
Text - [Blank]

When done, the form will look something like this (try to space your controls as shown; we'll use all the blank space as we continue with this example):



4. Save the application. Run the application. Cycle through the various book titles using the data control. Did you notice something? You didn't have to write one line of Visual Basic code! This indicates the power behind the data tool and bound tools.

### 5.6. Creating a Virtual Table

- Many times, a database table has more information than we want to display. Or, perhaps a table does not have all the information we want to display. For instance, in Example 8-1, seeing the Title and ISBN of a book is not real informative - we would also like to see the Author, but that information is not provided by the Titles table. In these cases, we can build our own virtual table, displaying only the information we want the user to see.

- We need to form a different SQL statement in the RecordSource property. Again, we won't be learning SQL here. We will just give you the proper statement.

#### Quick Example: Forming a Virtual Table

1. We'll use the results of Example 8-1 to add the Author name to the form. Replace the RecordSource property of the dtaTitles control with the following SQL statement:

```
SELECT Author, Titles.ISBN, Title FROM Authors, [Title Author], Titles WHERE  
Authors.Au_ID=[Title Author].Au_ID AND Titles.ISBN=[Title Author].ISBN  
ORDER BY Author
```

This must be typed as a single line in the Command Text (SQL) area that appears when you click the ellipsis by the RecordSource property. Make sure it is typed in exactly as



shown. Make sure there are spaces after 'SELECT', after 'Author, Titles.ISBN, Title', after 'FROM', after 'Authors, [Title Author], Titles', after 'WHERE', after 'Authors.Au\_ID=[Title Author].Au\_ID', after 'AND', after 'Titles.ISBN=[Title Author].ISBN', and separating the final three words 'ORDER BY Author'. The program will tell you if you have a syntax error in the SQL statement, but will give you little or no help in telling you what's wrong.

Here's what this statement does: It selects the Author, Titles.ISBN, and Title fields from the Authors, Title Author, and Titles tables, where the respective Au\_ID and ISBN fields match. It then orders the resulting virtual table, using authors as an index.

2. Add a label box and text box to the form, for displaying the author name. Set the control properties.

### Label3:

Caption - Author

### Text1:

DataSource - dtaTitles (select, don't type)

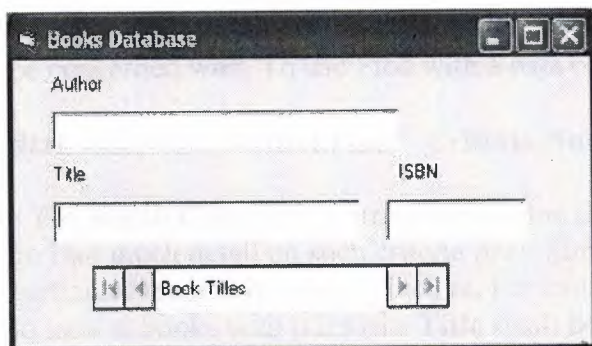
DataField - Author (select, don't type)

Locked - True

Name - txtAuthor

Text - [Blank]

When done, the form should resemble this:



3. Save, then rerun the application. The author's names will now appear with the book titles and ISBN values. Did you notice you still haven't written any code?

I know you had to type out that long SQL statement, but that's not code, technically speaking. Notice how the books are now ordered based on an alphabetical listing of authors' last names

## 5.7.Finding Specific Records

- In addition to using the data control to move through database records, we can write Visual Basic code to accomplish the same, and other, tasks. This is referred to as **programmatic control**. In fact, many times the data control **Visible** property is set to **False** and all data manipulations are performed in code. We can also use programmatic control to find certain records.

- There are four methods used for moving in a database. These methods replicate the capabilities of the four arrow buttons on the data control:

**MoveFirst** - Move to the first record in the table.

**MoveLast** - Move to the last record in the table.

**MoveNext** - Move to the next record (with respect to the current record) in the table.

**MovePrevious** - Move to the previous record (with respect to the current record) in the table.

- When moving about the database programmatically, we need to test the **BOF (beginning of file)** and **EOF (end of file)** properties. The BOF property is True when the current record is positioned before any data. The EOF property is True when the current record has been positioned past the end of the data. If either property is True, the current record is invalid. If both properties are True, then there is no data in the database table at all.

- These properties, and the programmatic control methods, operate on the **Recordset** property of the data control. Hence, to move to the first record in a table attached to a data control named **dtaExample**, the syntax is:

**dtaExample.Recordset.MoveFirst**

- There is a method used for searching a database:

**Find** - Find a record that meets the specified search criteria.

This method also operates on the Recordset property and has three arguments we will be concerned with. To use Find with a data control named dtaExample:

**dtaExample.Recordset.Find Criteria,NumberSkipped,SearchDirection**

- The search **Criteria** is a string expression like a **WHERE** clause in SQL. We won't go into much detail on such criteria here. Simply put, the criteria describes what particular records it wants to look at. For example, using our book database, if we want to look at books with titles (the **Title** field) beginning with S, we would use:

**Criteria = "Title >= 'S'"**

Note the use of single quotes around the search letter. Single quotes are used to enclose strings in Criteria statements. Three logical operators can be used: equals (=), greater than (>), and less than (<).

- The **NumberSkipped** argument tells how many records to skip before beginning the Find. This can be used to exclude the current record by setting NumberSkipped to 1.

- The **SearchDirection** argument has two possible values: **adSearchForward** or **adSearchBackward**. Note, in conjunction with the four Move methods, the **SearchDirection** argument can be used to provide a variety of search types (search from the top, search from the bottom, etc.)

- If a search fails to find a record that matches the criteria, the Recordset's **EOF** or **BOF** property is set to **True** (depending on search direction). Another property used in searches is the **Bookmark** property. This allows you to save the current record pointer in case you want to return to that position later. The **example** illustrates its use.

## 6.1 Basics of the SELECT Statement

In a relational database, data is stored in tables.

EmplAddressTable					
SSN	FirstName	LastName	Address	City	State
123456789	Hal	Glass	45 A Street	Indy	Indiana
565656565	Joe	Wilder	12 B Ave.	Burlington	Vermont
345678654	Mary	Smith	1234 C St.	Otto	Iowa
456743244	Rosa	Cobb	123 D St	Portland	Maine

If you want to see the address of each employee. Use the SELECT statement.

```
SELECT FirstName, LastName, Address, City, State
FROM EmplAddressTable;
```

The following is the results of your *query* of the database:

First Name	Last Name	Address	City	State
Hal	Glass	45 A Street	Indy	Indiana
Joe	Wilder	12 B Ave.	Burlington	Vermont
Mary	Smith	1234 C St..	Otto	Iowa
Rosa	Cobb	123 D St	Portland	Maine

The Statement asked for the all of data in the EmployeeAddressTable, from the *columns* called FirstName, LastName, Address, City, and State.

To get all columns from table do the following:

```
SELECT * FROM TableName;
```

## 6.2. Conditional Selection

To further discuss the SELECT statement, let's look at a new example table (for hypothetical purposes only):

EmployeeStatisticsTable			
EmployeeIDNo	Salary	Benefits	Position
010	75000	15000	Manager
105	65000	15000	Manager
152	60000	15000	Manager



215	60000	12500	Manager
244	50000	12000	Staff
300	45000	10000	Staff
335	40000	10000	Staff
400	32000	7500	Entry-Level
441	28000	7500	Entry-Level

### Relational Operators

There are six Relational Operators in SQL, and after introducing them, we'll see how they're used:

=	Equal
<> or != (see manual)	Not Equal
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To

The *WHERE* clause is used to specify that only certain rows of the table are displayed, based on the criteria described in that *WHERE clause*. It is most easily understood by looking at a couple of examples.

If you wanted to see the *EMPLOYEEIDNO*'s of those making at or over \$50,000, use the following:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY >= 50000;
```

Notice that the >= (greater than or equal to) sign is used, as we wanted to see those who made greater than \$50,000, or equal to \$50,000, listed together. This displays:

```
EMPLOYEEIDNO
-----
```

```
010
105
152
215
244
```

The *WHERE* description, *SALARY >= 50000*, is known as a condition (an operation which evaluates to True or False). The same can be done for text columns:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager';
```

This displays the ID Numbers of all Managers. Generally, with text columns, stick to equal to or not equal to, and make sure that any text that appears in the statement is surrounded by single quotes ('). Note: Position is now an illegal identifier because it is now an unused, but reserved, keyword in the SQL-92 standard.

### More Complex Conditions: Compound Conditions / Logical Operators

The AND operator joins two or more conditions, and displays a row only if that row's data satisfies ALL conditions listed (i.e. all conditions hold true). For example, to display all staff making over \$40,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY > 40000 AND POSITION = 'Staff';
```

The OR operator joins two or more conditions, but returns a row if ANY of the conditions listed hold true. To see all those who make less than \$40,000 or have less than \$10,000 in benefits, listed together, use the following query:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY < 40000 OR BENEFITS < 10000;
```

AND & OR can be combined, for example:

```
SELECT EMPLOYEEIDNO
```

## 6.3.Joins

In this section, we will only discuss *inner* joins, and *equijoins*, as in general, they are the most useful. For more information, try the SQL links at the bottom of the page.

Good database design suggests that each table lists data only about a single *entity*, and detailed information can be obtained in a relational database, by using additional tables, and by using a *join*.

First, take a look at these example tables:

### AntiqueOwners

OwnerID	OwnerLastName	OwnerFirstName
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

### Orders

OwnerID	ItemDesired
02	Table
02	Desk

21	Chair
15	Mirror

### Antiques

SellerID	BuyerID	Item
01	50	Bed



02	15	Table
15	02	Chair
21	50	Mirror
50	01	Desk
01	21	Cabinet
02	21	Coffee Table
15	50	Chair
01	15	Jewelry Box
02	21	Pottery
21	02	Bookcase
50	01	Plant Stand

## Keys

First, let's discuss the concept of *keys*. A *primary key* is a column or set of columns that uniquely identifies the rest of the data in any given row. For example, in the AntiqueOwners table, the OwnerID column uniquely identifies that row. This means two things: no two rows can have the same OwnerID, and, even if two owners have the same first and last names, the OwnerID column ensures that the two owners will not be confused with each other, because the unique OwnerID column will be used throughout the database to track the owners, rather than the names.

A *foreign key* is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key. In DBMS-speak, this correspondence is known as *referential integrity*. For example, in the Antiques table, both the BuyerID and SellerID are foreign keys to the primary key of the AntiqueOwners table (OwnerID; for purposes of argument, one has to be an Antique Owner before one can buy or sell any items), as, in both tables, the ID rows are used to identify the owners or buyers and sellers, and that the OwnerID is the primary key of the AntiqueOwners table. In other words, all of this "ID" data is used to refer to the owners, buyers, or sellers of antiques, themselves, without having to use the actual names.

## Performing a Join

The purpose of these *keys* is so that data can be related across tables, without having to repeat data in every table--this is the power of relational databases. For example, you can find the names of those who bought a chair without having to list the full name of the buyer in the Antiques table...you can get the name by relating those who bought a chair with the names in the AntiqueOwners table through the use of the OwnerID, which *relates* the data in the two tables. To find the names of those who bought a chair, use the following query:

```
SELECT OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE BUYERID = OWNERID AND ITEM = 'Chair';
```

Note the following about this query...notice that both tables involved in the relation are listed in the FROM clause of the statement. In the WHERE clause, first notice that the ITEM = 'Chair' part restricts the listing to those who have bought (and in this example, thereby own) a chair. Secondly, notice how the ID columns are related from one table to the next by use of the BUYERID = OWNERID clause. Only where ID's match across tables and the item purchased is a chair (because of the AND), will the names from the



AntiqueOwners table be listed. Because the joining condition used an equal sign, this join is called an *equijoin*. The result of this query is two names: Smith, Bob & Fowler, Sam.

*Dot notation* refers to prefixing the table names to column names, to avoid ambiguity, as follows:

```
SELECT ANTIQUEOWNERS.OWNERLASTNAME, ANTIQUEOWNERS.OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE ANTIQUES.BUYERID = ANTIQUEOWNERS.OWNERID AND ANTIQUES.ITEM =
'Chair';
```

As the column names are different in each table, however, this wasn't necessary.

### ***DISTINCT* and Eliminating Duplicates**

Let's say that you want to list the ID and names of **only** those people who have sold an antique. Obviously, you want a list where each seller is only listed once--you don't want to know how many antiques a person sold, just the fact that this person sold one (for counts, see the Aggregate Function section below). This means that you will need to tell SQL to eliminate duplicate sales rows, and just list each person only once. To do this, use the *DISTINCT* keyword.

First, we will need an equijoin to the AntiqueOwners table to get the detail data of the person's LastName and FirstName. However, keep in mind that since the SellerID

column in the Antiques table is a foreign key to the AntiqueOwners table, a seller will only be listed if there is a row in the AntiqueOwners table listing the ID and names. We also want to eliminate multiple occurrences of the SellerID in our listing, so we use *DISTINCT* on the column where the repeats may occur (however, it is generally not necessary to strictly put the Distinct in front of the column name).

To throw in one more twist, we will also want the list alphabetized by LastName, then by FirstName (on a LastName tie). Thus, we will use the *ORDER BY* clause:

```
SELECT DISTINCT SELLERID, OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUES, ANTIQUEOWNERS
WHERE SELLERID = OWNERID
ORDER BY OWNERLASTNAME, OWNERFIRSTNAME;
```

In this example, since everyone has sold an item, we will get a listing of all of the owners, in alphabetical order by last name. For future reference (and in case anyone asks), this type of join is considered to be in the category of *inner joins*.

### ***Aliases & In/Subqueries***

In this section, we will talk about *Aliases*, *In* and the use of subqueries, and how these can be used in a 3-table example. First, look at this query which prints the last name of those owners who have placed an order and what the order is, only listing those orders which can be filled (that is, there is a buyer who owns that ordered item):

```
SELECT OWN.OWNERLASTNAME Last Name, ORD.ITEMDESIRED Item Ordered
FROM ORDERS ORD, ANTIQUEOWNERS OWN
WHERE ORD.OWNERID = OWN.OWNERID
AND ORD.ITEMDESIRED IN
(SELECT ITEM
FROM ANTIQUES);
```

This gives:

Last Name	Item Ordered
Smith	Table
Smith	Desk

Akins      Chair  
Lawson     Mirror

There are several things to note about this query:

First, the "Last Name" and "Item Ordered" in the Select lines gives the headers on the report.

The OWN & ORD are aliases; these are new names for the two tables listed in the FROM clause that are used as prefixes for all dot notations of column names in the

#### 6.4. Aggregate Functions

I will discuss five important *aggregate functions*: SUM, AVG, MAX, MIN, and COUNT. They are called aggregate functions because they summarize the results of a query, rather than listing all of the rows.

SUM () gives the total of all the rows, satisfying any conditions, of the given column, where the given column is numeric.

AVG () gives the average of the given column.

MAX () gives the largest figure in the given column.

MIN () gives the smallest figure in the given column.

COUNT(\*) gives the number of rows satisfying the conditions.

Looking at the tables at the top of the document, let's look at three examples:

```
SELECT SUM(SALARY), AVG(SALARY)
FROM EMPLOYEESTATISTICSTABLE;
```

This query shows the total of all salaries in the table, and the average salary of all of the entries in the table.

```
SELECT MIN(BENEFITS)
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager';
```

This query gives the smallest figure of the Benefits column, of the employees who are Managers, which is 12500.

```
SELECT COUNT(*)
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Staff';
```

This query tells you how many employees have Staff status (3).

#### 6.5. Deleting Data

Let's delete this new row back out of the database:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman';
```

But if there is another row that contains 'Ottoman', that row will be deleted also. Let's delete all rows (one, in this case) that contain the specific data we added before:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman' AND BUYERID = 01 AND SELLERID = 21;
```

---

#### 6.6. Updating Data

Let's update a Price into a row that doesn't have a price listed yet:

```
UPDATE ANTIQUES SET PRICE = 500.00 WHERE ITEM = 'Chair';
```

This sets all Chair's Prices to 500.00. As shown above, more WHERE conditionals, using AND, must be used to limit the updating to more specific rows. Also, additional columns may be set by separating equal statements with commas.

## **chapter.7 Car Service Garrage Program with visual basic**

### **7.1.About Project**

At car service garage program including 4 parts first part is customer information form. Second part is invoice display form, third part is invoice record form, and finally database access part . Customer information form and invoice record form connect to database access table .Enty the knowledge customer information form and invoice information form records in the database access table.



## **7.2.Advantages of this program**

This program customer and car information save in the database access table and this information make a add ,update ,search and delete.This program also contain invoice program part invoice information save in the access table make a add , update,search and delete and display the invoice screen .

- 1.fast record in the database access
- 2.fast search
- 3.fast delete records in the table
- 4.update the records

### 7.3.step by step program

At project, created three form

Form1 is "CUSTOMER INFORMATION FORM"

At form1 used the 9 label 7 text 5 command

For form1 (code)

```
Dim Db As Database
Dim Tb As Recordset
```

```
Private Sub Command1_Click()
Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("customertable")
Tb.AddNew
Tb.Fields("Customer_ID") = Text1.Text
Tb.Fields("Customer_Name") = Text2.Text
Tb.Fields("Customer_tel") = Text3.Text
Tb.Fields("Customer_Address") = Text4.Text
Tb.Fields("Car_Plate") = Text5.Text
Tb.Fields("Car_Mark") = Text6.Text
Tb.Fields("Car_Model") = Text7.Text
Tb.Update
Tb.Close
Db.Close
End Sub
```

```
Private Sub Command2_Click()
```

```

Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("customertable")
Tb.Index = "primarykey"
Tb.Seek "=", Text1.Text
If Tb.NoMatch = 0 Then
ans = MsgBox("Do you want update", 4, "update")
If ans = 6 Then
Tb.Edit
Tb.Fields("Customer_Name") = Text2.Text
Tb.Fields("Customer_tel") = Text3.Text
Tb.Fields("Customer_Address") = Text4.Text
Tb.Fields("Car_Plate") = Text5.Text
Tb.Fields("Car_Mark") = Text6.Text
Tb.Fields("Car_Model") = Text7.Text
Tb.Update
Tb.Close
Db.Close
End If
Else
Exit Sub
End If
End Sub

```

```

Private Sub Command4_Click()
Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("customertable")
Tb.Index = "primarykey"
Tb.Seek "=", Text1.Text
If Tb.NoMatch = 0 Then
Text2.Text = Tb.Fields("Customer_Name")
Text3.Text = Tb.Fields("Customer_Tel")
Text4.Text = Tb.Fields("Customer_Address")
Text5.Text = Tb.Fields("Car_Plate")
Text6.Text = Tb.Fields("Car_mark")
Text7.Text = Tb.Fields("Car_Model")
End If
Tb.Close
Db.Close

```

```

End Sub

```

```

Private Sub Command5_Click()
Form2.Show
End Sub

```

```

Private Sub Text2_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Text3.SetFocus

```



```
End If
End Sub
```

```
Private Sub Text3_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Text4.SetFocus
End If
End Sub
```

```
Private Sub Text4_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Text5.SetFocus
End If
End Sub
Private Sub Text5_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Text6.SetFocus
End If
End Sub
```

```
Private Sub Text6_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Text7.SetFocus
End If
End Sub
Private Sub Text7_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Command1.SetFocus
End If
End Sub
```

```
Private Sub Form_Load()
Label1.Caption = "CUSTOMER INFORMATION FORM"
Label2.Caption = "Customer ID"
Label3.Caption = "Customer Name"
Label4.Caption = "CustomerTel "
Label5.Caption = "Customer Address"
Label6.Caption = "Plate No of Car"
Label7.Caption = "Car Mark"
Label8.Caption = "Car Model"
Label9.Caption = "Car information Part"
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
Text6.Text = ""
Text7.Text = ""
```

Command1.Caption = "Add"  
 Command2.Caption = "Update"  
 Command3.Caption = "Delete"  
 Command4.Caption = "Search"  
 Command5.Caption = "invoice"  
 End Sub

```

Private Sub Command3_Click()
Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("customertable")
Tb.Index = "primarykey"
Tb.Seek "=", Text1.Text
If Tb.NoMatch = 0 Then
Tb.Delete
MsgBox "Record deleted"
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
Text6.Text = ""
Text7.Text = ""
Else
MsgBox "Record not found"
End If
Tb.Close
Db.Close
End Sub
  
```

**Form2** is "INVOICE"

At form2(INVOICE) used the 5 label 1 combo 1 3 text 1 msflexgrid 2 button

### For form2(codes)

```
Dim Db As Database
Dim Tb As Recordset
```

```
Private Sub Combo1_click()
Set Db = OpenDatabase("customer.mdb")
SQL = "select * from car where part='" & Combo1.Text & "'"
Set Tb = Db.OpenRecordset(SQL)
While Not Tb.EOF
MSFlexGrid1.AddItem Tb.Fields("part") & Chr(9) & Str(Tb.Fields("price")) & Chr(9)
& Str(Tb.Fields("labor")) & Chr(9) & Str(Tb.Fields("total"))
Tb.MoveNext
Wend
Tb.Close
Db.Close

End Sub
```

```
Private Sub Command1_Click()
Set Db = OpenDatabase("customer.mdb")
SQL = "select * from car where ID=" & Val(Text1.Text) & ""
Set Tb = Db.OpenRecordset(SQL)
```



```

tot = 0
While Not Tb.EOF
tot = tot + Tb.Fields("price") + Tb.Fields("labor")
MSFlexGrid1.AddItem Tb.Fields("part") & Chr(9) & Str(Tb.Fields("price")) & Chr(9)
& Str(Tb.Fields("labor")) & Chr(9) & Str(tot)
Tb.MoveNext
Wend
Tb.Close
Db.Close

```

End Sub

```

Private Sub Form_Load()
Label1.Caption = "INVOICE"
Label2.Caption = "Customer id"
Label3.Caption = "Customer Name"
Label4.Caption = "Customer Tel"
Combo1.AddItem "Polen Filter"
Combo1.AddItem "Air Filter"
Combo1.AddItem "spark Plug"
Combo1.AddItem "Battery"
Combo1.AddItem "hydroulic oil"
Combo1.AddItem "Hydroulic"
Combo1.AddItem "Brake oil"
Combo1.AddItem "Rubber"
Combo1.AddItem "Engine Oil"
Combo1.AddItem "Engine Water"
Combo1.AddItem "Medicated Water"
Combo1.AddItem "Other"
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Command1.Caption = "calculate"
Command2.Caption = "invoice record"
MSFlexGrid1.Cols = 4
MSFlexGrid1.Rows = 1
MSFlexGrid1.Clear
MSFlexGrid1.Col = 0
MSFlexGrid1.Row = 0
MSFlexGrid1.Text = "part"
MSFlexGrid1.Col = 1
MSFlexGrid1.Row = 0
MSFlexGrid1.Text = "price"
MSFlexGrid1.Col = 2
MSFlexGrid1.Row = 0
MSFlexGrid1.Text = "labor"
MSFlexGrid1.Col = 3
MSFlexGrid1.Row = 0
MSFlexGrid1.Text = "total"

```

```

MSFlexGrid1.ColWidth(0) = 1200
MSFlexGrid1.ColWidth(1) = 2000
MSFlexGrid1.ColWidth(2) = 2000
MSFlexGrid1.ColWidth(3) = 3000
End Sub
Private Sub Command2_Click()
Form3.Show
End Sub

```

Form3 is "Invoice record FORM"  
 At form1 used the 5 label 4 text 4command

form3(code)

```

Dim Db As Database
Dim Tb As Recordset

```

```

Private Sub Command1_Click()
Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("car")
Tb.AddNew
Tb.Fields("Part") = Text1.Text
Tb.Fields("Price") = Val(Text2.Text)
Tb.Fields("Labor") = Val(Text3.Text)
Tb.Fields("Total") = Val(Text4.Text)

```

```
Tb.Update
Tb.Close
Db.Close
End Sub
```

```
Private Sub Command2_Click()
Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("car")
Tb.Index = "primarykey"
Tb.Seek "=", Val(Text1.Text)
If Tb.NoMatch = 0 Then
ans = MsgBox("Do you want update", 4, "update")
If ans = 6 Then
Tb.Edit
Tb.Fields("Part") = Text2.Text
Tb.Fields("Price") = Text3.Text
Tb.Fields("Labor") = Text4.Text
Tb.Fields("Total") = Text5.Text
Tb.Update
Tb.Close
Db.Close
End If
Else
Exit Sub
End If
End Sub
```

```
Private Sub Command4_Click()
Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("car")
Tb.Index = "primarykey"
Tb.Seek "=", Val(Text1.Text)
If Tb.NoMatch = 0 Then
Text2.Text = Tb.Fields("Part")
Text3.Text = Tb.Fields("Price")
Text4.Text = Tb.Fields("Labor")
Text5.Text = Tb.Fields("Total")
End If
Tb.Close
Db.Close

End Sub
```

```
Private Sub Text2_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Text3.SetFocus
End If
```



End Sub

```
Private Sub Text3_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Text4.SetFocus
End If
End Sub
```

```
Private Sub Text4_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then
Command1.SetFocus
End If
End Sub
```

```
Private Sub Form_Load()
Label1.Caption = "INVOICE INFORMATION RECORD"
Label2.Caption = "Part"
Label3.Caption = "Price"
Label4.Caption = "Labor "
Label5.Caption = "Total"
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Command1.Caption = "Add"
Command2.Caption = "Update"
Command3.Caption = "Delete"
Command4.Caption = "Search"
End Sub
```

```
Private Sub Command3_Click()
Set Db = OpenDatabase("customer.mdb")
Set Tb = Db.OpenRecordset("car")
Tb.Index = "primarykey"
Tb.Seek "=", Val(Text1.Text)
If Tb.NoMatch = 0 Then
Tb.Delete
MsgBox "Record deleted"
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Else
MsgBox "Record not found"
End If
Tb.Close
Db.Close
End Sub
```

End the program

## 7.4.Conclusion

This program included three part form1,form2,form3. A t program when the clicked on a add button at form1 than saved the customer and car information in the database access(customertable),when clicked on a search button at form1 find the I,nformation by id number,when clicked delete button at form1 delete record by id number in the database access(customertable)and when clicked on a update button at form1 change the customer and car information.form2 display the invoice model when clicked on a add button at form3 than saved the invoice information in the database acces(invoice) when cliced on a search button at form3 find the record by id number, when clicked on a delete button at form3 delete record by id number in the database access(invoicetable).



## CONCLUSION

At graduation projects introduced 6 important chapter of visual basic6,0. Chapter one described(The visual basic language) and parts ,chapter two described(Exploring the visual basic tollbok) and parts, chapter three described(More Exploration of the Visual Basic Toolbox) and parts ,chapter four described(Error-Handling, Debugging and File Input/Output) and parts

Chapter5 described(Database Access Management) and parts chapter6 described(sql) and parts.

At chapter seven maked the car service garage program with visual basic.

## REFERENCES

References to electronic sources-online sources from web:

1. [www.freeprogrammingresources.com/visual-basic-books.html](http://www.freeprogrammingresources.com/visual-basic-books.html)
2. <http://www.freetutes.com/VisualBasic/lesson21.html>
3. <http://staffwww.fullcollc.edu/dcraig/vbasic>
4. [msdn.microsoft.com/en-us/vbasic/default.aspx](http://msdn.microsoft.com/en-us/vbasic/default.aspx)
5. [www.vbtutor.net/vbtutor.html](http://www.vbtutor.net/vbtutor.html)
6. [www.murach.com/books/vb60/index.htm](http://www.murach.com/books/vb60/index.htm)