Chapter 2

Variables

The named memory Location is called Variable. Variables are used to store values or data In Visual Basic; you use variables to temporarily store values during the execution of an application. Variables have a name and a data type. You can think of a variable as a placeholder in memory for an unknown value.

Declaring Variables

To declare a variable is to tell the program about it in advance. You declare a variable with the Dim statement, supplying a name for the variable:

Syntax: **Dim** *variable name* [**As** *type*]

In this syntax,

- **Dim** is the *keyword* that tells Visual Basic that you want to declare a variable.
- *Variable Name* is the name of the variable.
- As is the keyword that tells Visual Basic that you're defining the data type for the variable.
- *Type* is the data type of the variable.

- Variables declared with the Dim statement within a procedure exist only as long as the procedure is executing. When the procedure finishes, the value of the variable disappears.
- In addition, the value of a variable in a procedure is *local* to that procedure — that is, you can't access a variable in one procedure from another procedure.
- These characteristics allow you to use the same variable names in different procedures without worrying about conflicts or accidental changes.

A variable name:

- Must begin with a letter.
- Can't contain an embedded period or embedded type-declaration character.
- Must not exceed 255 characters.
- Must be unique within the same *scope*, which is the range from which the variable can be referenced a procedure, a form, and so on.

There are various ways of declaring a variable in VB depending upon where the variables are declared as given below.

1) Explicit declaration
2) Implicit Declaration

1) Explicit Declaration

Explicit declaration means that you must use a statement to define a variable. Each of the following statements can be used to explicitly declare a variable's type:

Dim *VarName* [As *VarType*][, *VarName2* [As *VarType2*]]
Private *VarName*[As *VarType*][, *VarName2*[As *VarType2*]]
Static *VarName*[As *VarType*][, *VarName2*[As *VarType2*]]
Public *VarName*[As *VarType*][, *VarName2*[As *VarType2*]]

Dim, Private, Static, and Public are Visual Basic keywords that define how and where the variable can be used. *VarName* and *VarName2* represent the names of the variables that you want to declare.

As indicated in the syntax, you can specify multiple variables in the same statement as long as you separate the variables by commas. *VarType* and *VarType2* represent the type name of the respective variables.

The *type name* is a keyword that tells Visual Basic what kind of information will be stored in the variable.

2) Implicit Declaration

Whenever you use implicit declaration, Visual Basic considers that variable as type Variant. Using implicit declaration isn't recommended, however. Making a variable without a formal declaration is asking for trouble. If you use implicit declaration, then any time you make a spelling mistake or syntax error, Visual Basic will think that you're *implicitly* declaring another variable, which can lead to headaches beyond imagination.

4.5 Constants

To greatly improve the readability of your code — and make it easier to maintain Constants is used.
A *constant* is a meaningful name that takes the place of a number or string that does not change.
Although a constant somewhat be similar to a variable, you can't modify a constant or assign a new value to it as you can to a variable.
There are two sources for constants:
Intrinsic* or *system-defined constants are provided by applications and controls. Visual Basic constants are listed in the Visual Basic (VB) and Visual Basic for applications (VBA) object libraries in the Object Browser. Other applications that provide object libraries, such as Microsoft Excel and Microsoft Project, also provide a list of constants you can use with their objects, methods, and properties.

Symbolic* or *user-defined constants are declared using the Const statement.
The syntax for declaring a constant is:
[**Public**|**Private**] **Const** *constantname*[**As** *type*] **=** *expression*
The argument *constantname* is a valid symbolic name (the rules are the same as those for creating variable names), and *expression* is composed of numeric or string constants and operators; however, you can't use function calls in *expression*.
A Const statement can represent a mathematical or date/time quantity:
Const conPi = 3.14159265358979
Public Const conMaxPlanets As Integer = 9
Const conReleaseDate = #1/1/95#
The Const statement can also be used to define string constants:
Public Const conVersion = "07.10.A"
Const conCodeName = "Enigma"
You can place more than one constant declaration on a single line if you separate them with commas:
Public Const conPi = 3.14, conMaxPlanets = 9,conWorldPop = 6E+09
A Const statement has scope like a variable declaration, and the same rules apply:

- To create a constant that exists only within a procedure, declare it within that procedure.
- To create a constant available to all procedures within a module, but not to any code outside that module, declare it in the Declarations section of the module.
- To create a constant available throughout the application, declare the constant in the Declarations section of a standard module, and place the Public keyword before Const. Public constants cannot be declared in a form or class module.

Operator

An operator is a symbol that performs an operation on one or more code elements that hold values. Value elements include variables, constants, literals, properties, and returns from Function. Visual Basic provides the following types of operators:

1. **Arithmetic Operators**
2. **Comparison Operators**.
3. **Concatenation Operators**
4. **Logical and Bitwise**

Dayanand Science College Latur

Class: B.Sc. Ty **Sub: Visual Basic**

1. Arithmetic Operators:

These operators are used to perform basics arithmetic operations.

Operator	Action
^	exponentiation
−	Subtraction, also unary minus
+	Addition
*	Multiplication
/	Division
Mod	Modulus

2. Comparison Operators:

Compare two expressions and return a Boolean value representing the result of the comparison.

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
Like	a pattern matching operator
Is	an object type comparer

3. Concatenation Operators:

Join multiple strings into a single string.

Operator	Action
&	String concatenation
+	String concatenation

4. Logical and Bitwise Operators:

In Visual Basic combine Boolean or numeric values and return a result of the same data type as the values.

Operator	Action
And	And
Or	Or
Not	Not
Xor	exclusive or

Operator precedence & associativity

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence.

Precedence Rules:

When expressions contain operators from more than one category, they are evaluated according to the following rules:

1. The arithmetic and concatenation operators have the order of precedence described below, and all have higher precedence than the comparison, logical, and bitwise operators.
2. All comparison operators have equal precedence, and all have higher precedence than the logical and bitwise operators, but lower precedence than the arithmetic and concatenation operators.
3. The logical and bitwise operators have the order of precedence described below, and all have lower precedence than the arithmetic, concatenation, and comparison operators.
4. Operators with equal precedence are evaluated left to right in the order in which they appear in the expression.

Precedence Order:

Operators are evaluated in the following order of precedence:

Arithmetic Operators	Comparison Operators	Logical and Bitwise	Concatenation
^	=	Not	&, +
− negation	<>	And	
*, /	<	Or	
Mod	>	Xor	

+, -	<=
	>=
	Like, Is

Associativity

When operators of equal precedence appear together in an expression, for example multiplication and division, the compiler evaluates each operation as it encounters it from left to right. The following example illustrates this.

Dim n1 As Integer = 96 / 8 / 4

The first expression evaluates the division 96 / 8 (resulting in 12) and then the division 12 / 4, resulting in 3. Because the compiler evaluates the operations for n1 from left to right

Dim n2 As Integer = (96 / 8) / 4

The evaluation is exactly the same when that order is explicitly indicated for n2. Both n1 and n2 have a result of

Dim n3 As Integer = 96 / (8 / 4)

By contrast, n3 has a result of 48, because the parentheses force the compiler to evaluate 8 / 4 first.
Because of this behavior, operators are said to be left associative in Visual Basic.

Data Types

Visual basic Provides the following data types.

Data type	**Description**	**Range**
Byte	1-byte binary data	0 to 255
Integer	2-byte integer	– 32,768 to 32,767
Long	4-byte integer	– 2,147,483,648 to 2,147,483,647
Single	4-byte floating-point number	– 3.402823E38 to – 1.401298E – 45 (negative values)
		1.401298E – 45 to 3.402823E38 (positive values)
Double	8-byte floating-point number	– 1.79769313486231E308 to – 4.94065645841247E – 324 (negative values)
		4.94065645841247E – 324 to 1.79769313486231E308 (positive values)
Currency	8-byte number with fixed decimal point	– 922,337,203,685,477.5808 to 922,337,203,685,477.5807
String	String of characters	Zero to approximately two billion characters
Variant	Date/time, floating-point number, integer, string, or object. 16 bytes, plus 1 byte for each character if a string value.	Date values: January 1, 100 to December 31, 9999 Numeric values: same range as **Double** String values: same range as **String** Can also contain **Error** or **Null** values
Boolean	2 bytes	**True** or **False**
Date	8-byte date/time value	January 1, 100 to December 31, 9999
Object	4 bytes	Any **Object** reference

Boolean

A Boolean variable is one whose value can be only either True or False. To declare such a variable, use the Boolean keyword. Here is an example:

```
Private Sub Form_Load()
   Dim IsMarried As Boolean
End Sub
```

After declaring a Boolean variable, you can initialize by assigning it either True or False. Here is an example:

```
Private Sub Form_Load()
   Dim IsMarried As Boolean
   IsMarried = False
End Sub
```

Like any other variable, after initializing the variable, it keeps its value until you change its value again.

Byte

A byte is a small natural positive number that ranges from 0 to 255. A variable of byte type can be used to hold small values such as a person's age, the number of fingers on an animal, etc.

To declare a variable for a small number, use the Byte keyword. Here is an example:

```
Private Sub Form_Load()
   Dim StudentAge As Byte
End Sub
```

Currency

Currency variables are stored as 64-bit (8-byte) numbers in an integer format, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This representation provides a range of -922,337,203,685,477.5808 to 922,337,203,685,477.5807. The type-declaration character for **Currency** is the at sign (**@**).The **Currency** data type is useful for calculations involving money and for fixed-point calculations in which accuracy is particularly important.

```
Private Sub Form_Load()
   Dim Sal As currency
End Sub
```

Date

Date variables are stored as 64-bit (8-byte) floating-point numbers that represent dates ranging from 1 January 100 to 31 December 9999 and times from 0:00:00 to 23:59:59. Any recognizable literal date values can be assigned to **Date** variables. Date literals must be enclosed within number signs (**#**), for example, #January 1, 1993# or #1 Jan 93#. **Date** variables display dates according to the short date format recognized by your computer. Times display according to the time format (either 12-hour or 24-hour) recognized by your computer.

Double

Double (double-precision floating-point) variables are stored as 64-bit (8-byte) floating-point numbers ranging in value from -1.79769313486232E308 to -4.94065645841247E-324 for negative values and from 4.94065645841247E-324 to 1.79769313486232E308 for positive values.

Integer

Integer variables are stored as 16-bit (2-byte) numbers ranging in value from -32,768 to 32,767. The type-declaration character for **Integer** is the percent sign (**%**)

Long

Long (long integer) variables are stored as signed 32-bit (4-byte) numbers ranging in value from -2,147,483,648 to 2,147,483,647. The type-declaration character for **Long** is the ampersand (**&**).

Object

Object variables are stored as 32-bit (4-byte) addresses that refer to objects. Using the **Set** statement, a variable declared as an **Object** can have any object reference assigned to it.

Single

Single (single-precision floating-point) variables are stored as IEEE 32-bit (4-byte) floating-point numbers, ranging in value from -3.402823E38 to -1.401298E-45 for negative values and from 1.401298E-45 to 3.402823E38 for positive values. The type-declaration character for **Single** is the exclamation point (**!**).

String

There are two kinds of strings: variable-length and fixed-length strings. A variable-length string can contain up to approximately 2 billion (2^31) characters. A fixed-length string can contain 1 to approximately 64K (2^16) characters.

Variant

The **Variant** data type is the data type for all variables that are not explicitly declared as some other type (using statements such as **Dim**, **Private**, **Public**, or **Static**). The **Variant** data type has no type-declaration character A **Variant** is a special data type that can contain any kind of data.

I/O Statements

Visual Basic provides two basic I/O statements for displaying (or requesting) information to the user: MsgBox() and InputBox(). Windows applications should communicate with the user via nicely designed Forms, but the MsgBox() and InputBox() functions are still around and quite useful.

The Message Box (MSG Box)

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

MsgBox Message, Type, Title

where

Message Text message to be displayed

Type Type of message box

Title Text in title bar of message box

InputBox

One **excellent use** for a variable is to hold **user input** information. Often, you can use an object such as a file list box or a text box to retrieve this information. At times, though, you'll want to deal directly with the user and save the input in a variable rather than in a property. One way to do this is to use the **InputBox** function to display a dialog box on the screen and then store in a variable the text that the user types.

Like a message box, an input box is a (relatively) small form (in reality, it is a dialog box) that displays a message to the user. Unlike a message box, an input box presents a small text box that expects the user to enter a value. After using it, the user can either send the form with the new value or dismiss it without any change.

To create an input box, you can use the **InputBox** function procedure prompts the user to enter some information in a message box, and the function will return the content of that box.

InputBox syntax looks like this:

VariableName = InputBox (Prompt)

where

VariableName is a variable used to hold the input.

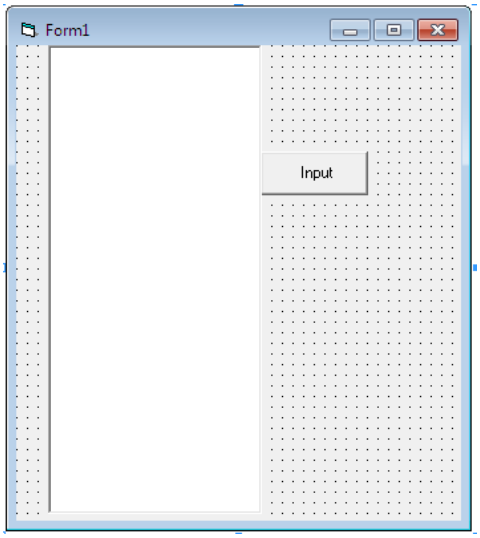
Prompt is a prompt that appears in the dialog box.

The dialog box created with an **InputBox** function typically contains these features:

- A prompt for directing the user.
- A text box for receiving typed input.
- Two command buttons, **OK** and **Cancel**.

Example (For Practical):

1. Start a new project.
2. Place the one Textbox & 1 command button on the form as shown in following form.



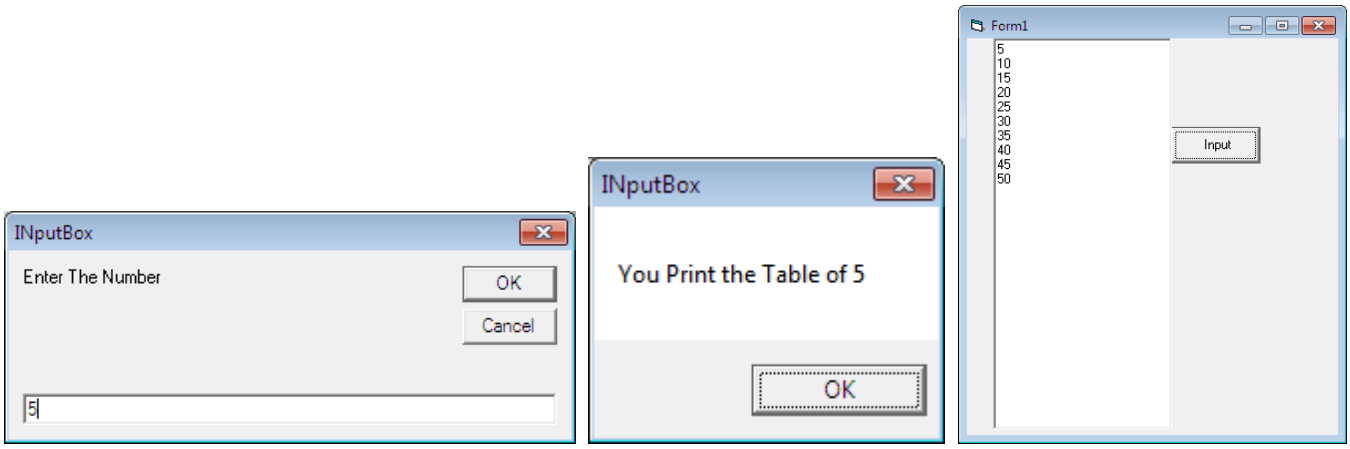
3. Set the properties of the form and each object.

Form1:	**Text1:**	**Command1**
Caption: Input	Multiline: True.	Caption: Input

4. Attach the following code to given event.

```
Private Sub Command1_Click()
Dim n, i, sum As Integer
Dim ms As String
sum = 0
n = InputBox("Enter The Number")
For i = 1 To 10
sum = 0
sum = n * i
ms = ms & sum & vbNewLine
 Next i
Text1.Text = ms
MsgBox "You Print the Table of‖ & n
End Sub
```

Output:



Control Flow Statements

An application needs a built-in capability to test condition and take a different course of action depending on the test. Visual Basic Provides three Control flow or Decision structures.

1. If….. Then
2. If……. Then …..Else
3. Select Case

If….. Then

The If…. Then structure tests the conditions specified, and if it's true, executes the statement(s) that follow. The If structure can have a single line or a multiple line syntax. To Execute one statement conditionally. Use the Single-Line syntax as follows

Syntax

IF
(Condition)
Then
(Statement)
Eg:
If Month(Date)=1 then
Year=year+1
You can also execute the multiple statements by separating them with a colon:
Syntex
IF
(Condition)
Then
(Statement):(Statement):(Statement)

If….Then….Else

A variation of the If...Then statement is the *If... Then... Else* statement, which executes one block of statements if the condition is True and another if the condition is False. The syntax of the If...Then...Else statement is as follows:

If condition Then
statementblock-1
Else
statementblock-2
End If

Visual Basic evaluates the *condition*, and if it's True, it executes the first block of statements and then jumps to the statement following the End If statement. If the condition is False, Visual Basic ignores the first block of statements and executes the block following the *Else* keyword.

Another variation of the If...Then...Else statement uses several conditions, with the *ElseIf* keyword:

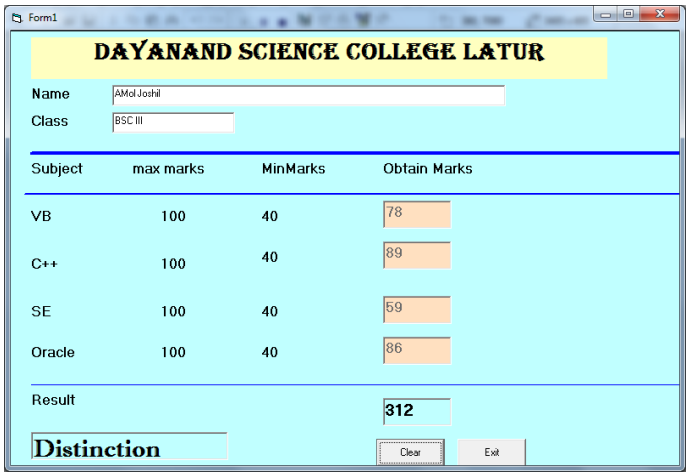
If condition1 Then
statementblock-1
ElseIf condition2 Then
statementblock-2
ElseIf condition3 Then
statementblock-3
Else
statementblock-4
End If

You can have any number of ElseIf clauses. The conditions are evaluated from the top, and if one of them is True, the corresponding block of statements is executed. The Else clause will be executed if none of the previous expressions are True. Here's an example of an If statement with ElseIf clauses:

score = InputBox("Enter score")
If score < 50 Then
Result = "Failed"
ElseIf score < 75 Then
Result = "Pass"
ElseIf score < 90 Then
Result = "Very Good"
Else
Result = "Excellent"
End If
MsgBox Result

Example (For Practical):

1. Start a new project.
2. Place the labels, Textboxes & lines on the form as shown in following form.



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

Form1:
BorderStyle 1-Fixed Single
Caption MarkMemo
Name frmMM

Label1:	**Label2:**	**Label3:**
Caption: Dayanand Science College, Latur Alignment 2-Center	Caption: Name	Caption: Class
Label4:	**Label5:**	**Label6:**
Caption Subject	Caption Max Mark	Caption Min Mark
Label7:	**Label8:**	**Label9:**
Caption Obtain Mark	Caption VB	Caption C++
Label10:	**Label11:**	**Label12:**
Caption SE	Caption Oracle	Caption 100
Label13:	**Label14:**	**Label15:**
Caption 100	Caption 100	Caption 100
Label16:	**Label17:**	**Label18:**
Caption 40	Caption 40	Caption 40
Label19:	**Label20:**	**Label21:**
Caption 40	Caption Result	Caption [Blank]
Label22:		
Caption [Blank]		

4. Attach the following code to given event.

```
Dim total As Integer
Dim per As Double

Private Sub Text3_LostFocus()
Text3.Enabled = False
End Sub

Private Sub Text4_LostFocus()
Text4.Enabled = False
```

```
End Sub

Private Sub Text5_LostFocus()
Text5.Enabled = False
End Sub

Private Sub Text6_LostFocus()
Text6.Enabled = False
total = Val(Text3.Text) + Val(Text4.Text) + Val(Text5.Text) + Val(Text6.Text)
Label15.Caption = total
If Val(Text3.Text) >= 40 And Val(Text4.Text) >= 40 And Val(Text5.Text) >= 40 And Val(Text6.Text) >= 40 Then
per = total / 4
If per >= 70 Then
Label16.Caption = "Des"
ElseIf per >= 60 Then
Label16.Caption = "First Class"
ElseIf per >= 50 Then
Label16.Caption = "Second Class"
ElseIf per >= 40 Then
Label16.Caption = "Third Class"
Else
Label16.Caption = "Fail"
End If
Else
Label16.Caption = "Fail"
End If
End Sub
Private Sub Command1_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
Text6.Text = ""
Text3.Enabled = True
Text4.Enabled = True
Text5.Enabled = True
Text6.Enabled = True
Text1.SetFocus
End Sub
Private Sub Command2_Click()
End
End Sub
```

Select Case statements

The *Select Case* structure compares one expression to different values. The advantage of the Select Case statement over multiple If...Then...Else a statement is that it makes the code easier to read and maintain.
The Select Case structure tests a single expression, which is evaluated once at the top of the structure. The result of the test is then compared with several values, and if it matches one of them, the corresponding block of statements is executed. Here's the syntax of the Select Case statement:

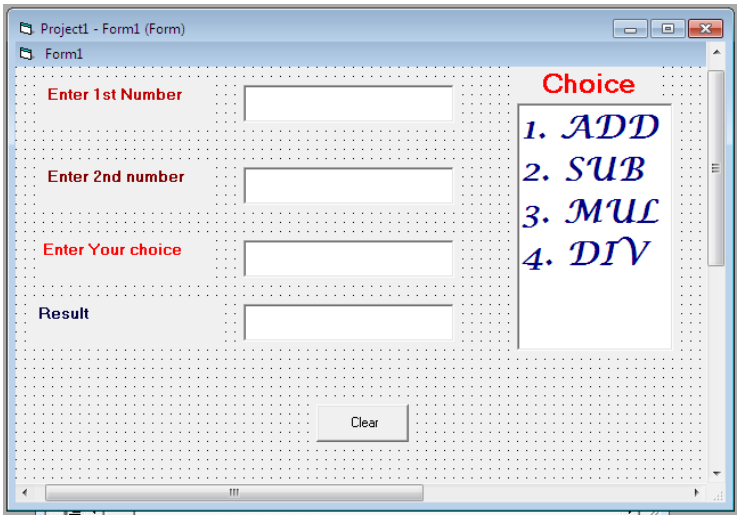
Select Case expression

Case value1
statementblock-1
Case value2
statementblock-2
.
.
.
Case Else
statementblock
End Select

The *expression* variable is evaluated at the beginning of the statement. The value of the expression is then compared with the values that follow each *Case* keyword. If they match, the block of statements up to the next *Case* keyword is executed, and the program skips to the statement following the End Select statement. The block of the Case Else statement is optional and is executed if none of the previous Case values match the expression.

Example (For Practical):

4. Start a new project.
5. Place the 5 labels, 5 Textboxes & 1 command button on the form as shown in following form.



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

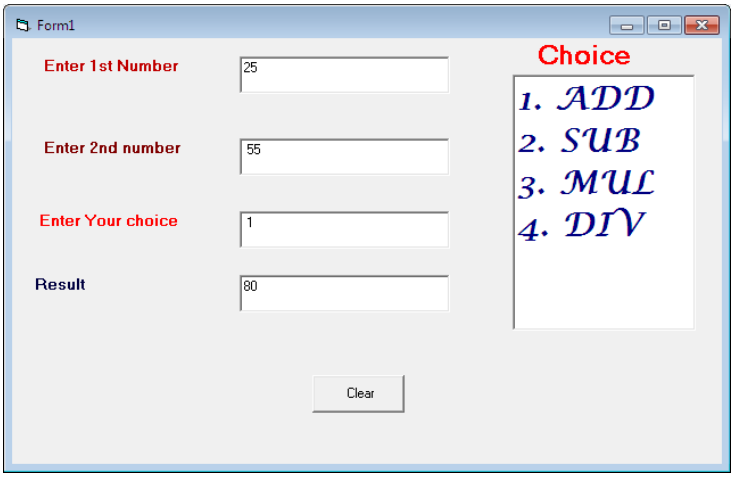
Form1:	**Command1**
Caption Select_Case	Caption Clear
Name frmSC	
Label1:	**Label2:**
Caption Enter 1st Number	Caption Enter 2nd Number
Alignment 2- Center	Alignment 2- Center
Label3:	**Label4:**
Caption Enter Your Choice	Caption Result
Alignment 2- Center	Alignment 0- Left
Label5:	**Text1:**
Caption Choice Multiline True.	

Alignment 2- Center	Text 1. ADD
Multiline True	2. SUB
Forecolor red	3. MUL
Font	4. DIV

5. Attach the following code to given event.

```
Private Sub Command1_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text5.Text = ""
Text1.SetFocus
End Sub

Private Sub Text3_LostFocus()
Dim n1, n2, res, ch As Integer
n1 = Val(Text1.Text)
n2 = Val(Text2.Text)
ch = Val(Text3.Text)
Select Case ch
Case 1
Text5.Text = n1 + n2
Case 2
Text5.Text = n1 - n2
Case 3
Text5.Text = n1 * n2
Case 4
Text5.Text = n1 / n2
Case 5
End
Case Else
Text5.Text = "Your choice is wrong"
End Select
End Sub
```

Output:



Loop Statements

Loop statements allow you to execute one or more lines of code repetitively. Many tasks consist of trivial operations that must be repeated over and over again, and looping structures are an important part of any programming language. Visual Basic supports the following loop statements:

- Do...Loop

- For...Next

Do...Loop

The Do...Loop executes a block of statements for as long as a condition is True. Visual Basic evaluates an expression, and if it's True, the statements are executed. If the expression is False, the program continues and the statement following the loop is executed.

There are two variations of the Do...Loop statement and both use the same basic model. A loop can be executed either while the condition is True or until the condition becomes True. These two variations use the keywords *While* and *Until* to specify how long the statements are executed. To execute a block of statements while a condition is True, use the following syntax:

Do While condition
statement-block
Loop

To execute a block of statements until the condition becomes True, use the following syntax:

Do Until condition
statement-block
Loop

When Visual Basic executes the previous loops, it first evaluates *condition*. If *condition* is False, the Do...While or Do...Until loop is skipped (the statements aren't even executed once). When the Loop statement is reached, Visual Basic evaluates the expression again and repeats the statement block of the Do...While loop if the expression is True, or repeats the statements of the Do...Until loop if the expression is False.

The Do...Loop can execute any number of times as long as *condition* is True. Moreover, the number of iterations need not be known before the loops starts. If *condition* is initially False, the statements may never execute.

Another variation of the Do loop executes the statements first and evaluates the *condition* after each execution. This Do loop has the following syntax:

Do
statements
Loop While condition
or
Do
statements
Loop Until condition

The statements in this type of loop execute at least once, since the condition is examined at the end of the loop.

For...Next

The *For...Next* loop is one of the oldest loop structures in programming languages. Unlike the Do loop, the For...Next loop requires that you know how many times the statements in the loop will be executed. The For...Next loop uses a variable (it's called the loop's *counter*) that increases or decreases in value during each repetition of the loop. The For...Next loop has the following syntax:

For counter = start To end [Step increment]
statements
Next [counter]

The keywords in the square brackets are optional. The arguments *counter*, *start*, *end*, and *increment* are all numeric. The loop is executed as many times as required for the *counter* to reach (or exceed) the *end* value.

In executing a For...Next loop, Visual Basic completes the following steps:

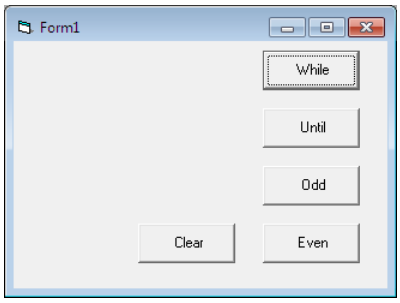
1. Sets *counter* equal to *start*

2. Tests to see if *counter* is greater than *end*. If so, it exits the loop. If *increment* is negative, Visual Basic tests

to see if *counter* is less than *end*. If it is, it exits the loop.
3. Executes the statements in the block
4. Increments *counter* by the amount specified with the *increment* argument. If the *increment* argument isn't specified, *counter* is incremented by 1.
5. Repeats the statements

Example (For Practical):

1. Start a new project.
2. Place five command buttons on the form as shown in following form.



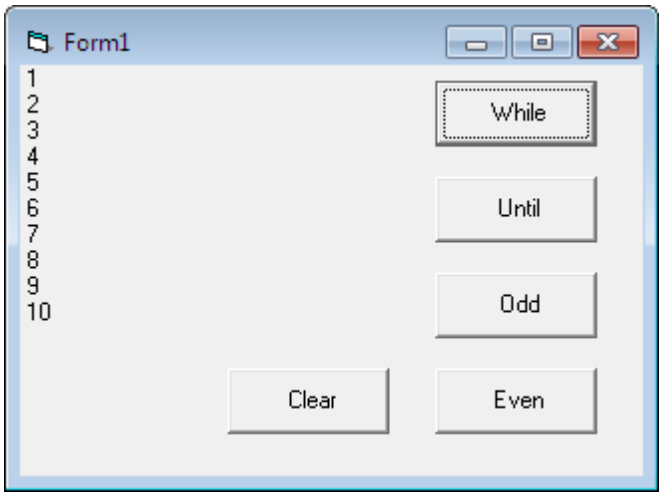
3. Set the properties of the command button.

Command1:
Caption While
Command2:
Caption Until
Command3:
Caption Odd
Command4:
Caption Even
Command5:
Caption Clear

4. Attach the following code to given event.

```
Private Sub Command1_Click()
Dim i As Integer
i = 1
Do While i <= 10
Form1.Print i
i = i + 1
Loop
End Sub

Private Sub Command2_Click()
Dim i As Integer
i = 1
Do Until i > 10
Form1.Print i
i = i + 1
Loop
End Sub

Private Sub Command3_Click()
Form1.Cls
End Sub

Private Sub Command4_Click()
Dim i As Integer
For i = 1 To 10 Step 2
Form1.Print i
Next
End Sub

Private Sub Command5_Click()
Dim i As Integer
For i = 2 To 10 Step 2
Form1.Print i
Next
End Sub
```

Output :



Arrays

A standard structure for storing data in any programming language is the array. Whereas individual variables can hold single entities, such as one number, one date, or one string, *arrays* can hold sets of data of the same type (a set of numbers, a series of dates, and so on). An array has a name, as does a variable, and the values stored in it can be accessed by an index.

An array is similar to a variable: it has a name and multiple values. Each value is identified by an index (an integer value) that follows the array's name in parentheses. Each different value is an *element* of the array.

For eg.

Dim salaries (15) As Integer

If the array *Salaries* holds the salaries of 16 employees, the element Salaries(0) holds the salary of the first employee, the element Salaries(1) holds the salary of the second employee, and so on up the element Salaries(15).

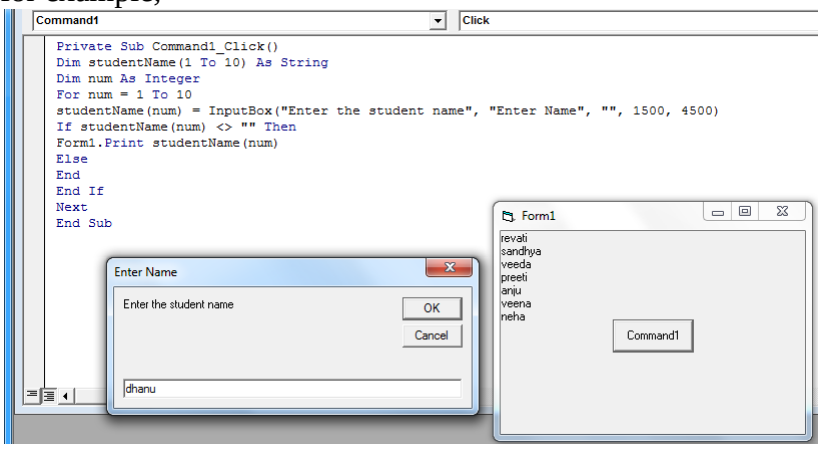
The indexing of arrays in VB.NET starts at zero.

Declaring Arrays

Arrays must be declared with the Dim statement followed by the name of the array and the index of the last element in the array in parentheses.

Syntax : ***Dim Array name( ) As Data type***

for example,



Initializing Arrays

Just as you can initialize variables in the same line where you declare them, you can initialize arrays, too, with the following syntax:

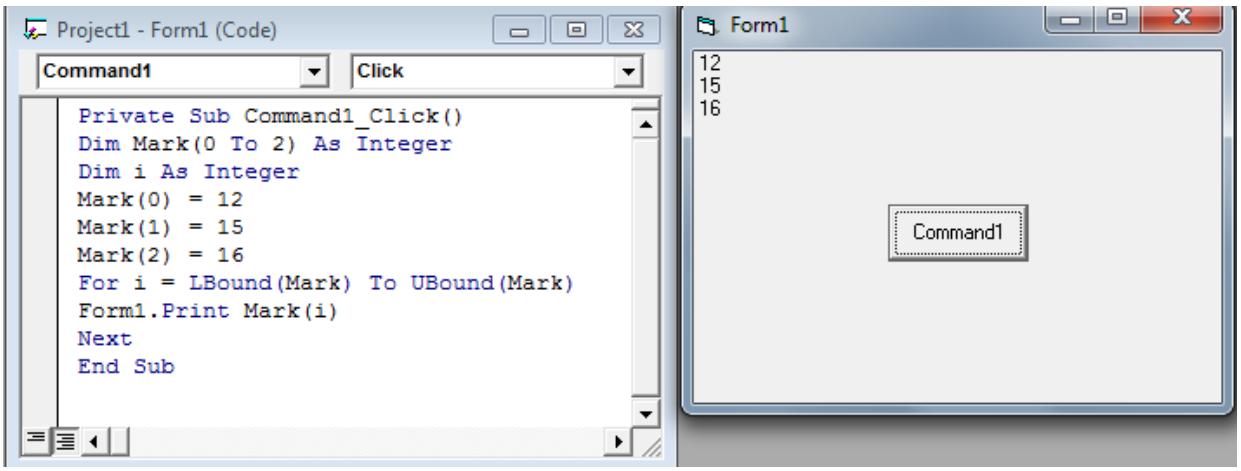
Dim arrayname() As type = {entry0, entry1, … entryN}
example that initializes an array of strings:
Dim names() As String = {"Joe Doe", "Peter Smack"}
This statement is equivalent to the following statements, which declare an array with two elements and then set their values:
Dim names(1) As String
names(0) = "Joe Doe"
names(1) = "Peter Smack"
The number of elements in the curly brackets following the array's declaration determines the dimensions of the array, and you can't add new elements to the array without resizing it.

Array Limits

The first element of an array has index 0. The number that appears in parentheses in the Dim statement is one less than the array's total capacity and is the array's upper limit (or upper bound).
The index of the last element of an array (its upper bound) is given by the function UBound(), which accepts as argument the array's name. For the array
Dim myArray(19) As Integer
its upper bound is 19, and its capacity is 20 elements.

```
Private Sub Command1_Click()
Dim a(5), i, max As Integer
    For i = 0 To 4
      a(i) = InputBox("enter a number")
    Next
    max = a(0)
    For i = 0 To 4
      If a(i + 1) > max Then
        max = a(i + 1)
      End If
    Next
    MsgBox ("Largest number is " & max)
End Sub
```

Eg.



Multidimensional Arrays

One-dimensional arrays, such as those presented so far, are good for storing long sequences of one dimensional

data (such as names or temperatures). But how would you store a list of cities *and* their average temperatures in an array? Or names and scores, years and profits.
A two-dimensional array has two indices. The first identifies the row, and the second identifies the column.

Declaring Multidimensional Arrays

Multidimensional arrays must be declared with the Dim statement followed by the name of the array and the index of the row & column in the array in parentheses.
Syntax : ***Dim Array name(row, column) As Data type***
for example,
Dim Marks(3,3) As Integer

Initializing Arrays

The following statements initialize a two-dimensional array.
Syntax :
Dim Array name( ) As Data type = {{element of 1st row}, { element of 2nd row }, { element of 3rd row },{ element of nth row }}
Dim a(,) As Integer = {{10, 20, 30}, {11, 21, 31}, {12, 22, 32}}

```
Dim i, j As Int16
  For i = 0 To 2
    For j = 0 To 3
      a(i, j) = (i * 4) + j + 1
    Next
  Next
  Label1.Text = a(0, 0)
  Label2.Text = a(0, 1)
  Label3.Text = a(0, 2)
  Label4.Text = a(0, 3)
  Label5.Text = a(1, 0)
  Label6.Text = a(1, 1)
  Label7.Text = a(1, 2)
  Label8.Text = a(1, 3)
  Label9.Text = a(2, 0)
  Label10.Text = a(2, 1)
  Label11.Text = a(2, 2)
  Label12.Text = a(2, 3)
```

Dynamic Arrays

Sometimes you may not know how large to make an array. Instead of making it large enough to hold the maximum number of data, you can declare a *dynamic array*. The size of a dynamic array can vary during the execution of the program.
With a dynamic array, you can discard the data and return the resources it occupied to the system.
To create a dynamic array, declare it as usual with the Dim statement (or Public or Private) but don't specify its dimensions:
Each time you execute the ReDim statement, all the values currently stored in the array are lost.
You can, however, change the size of the array without losing its data. The ReDim statement recognizes the Preserve keyword, which forces it to resize the array without discarding the existing data.
For example, you can enlarge an array by one element without losing the values of the existing elements by using the UBound() function as follows:

ReDim Preserve DynamicArray(UBound(DynArray) + 1)
If the array *DynamicArray* held 12 elements, this statement would add one element to the array, the element DynamicArray(12). The values of the elements with indices 0 through 11 wouldn't change.

```
Private Sub Command2_Click()
Dim a(), i, max, count, n As Integer
      count = InputBox("Enter array limit")
      ReDim a(count)
      For i = 0 To count
         a(i) = InputBox("enter a number")
      Next
           max = a(0)
      For i = 0 To count - 1
         If a(i + 1) > max Then
            max = a(i + 1)
         End If
      Next
      MsgBox ("Largest number is " & max)
End Sub
```

Collections

Collection class provides an array-like container more flexible than an array in some ways and less flexible in other ways. More flexible in some ways than Collection is Dictionary class.

Creating a new collection:

```
  Dim Cats As Collection
  Set Cats = New Collection
```

Dimensioning and creating a new collection in one line:

```
  Dim Cats As New Collection
```

Adding an item:

```
   Cats.Add "Item"
   Cats.Add "Item", "Key"
```

Accessing an item at an index, in a read-only way:

```
  Cats (3) 'The third member of the collection
  Cats.Item(3) 'An alternative
  Cats.Item("Key 3") 'Works if an item has a key associated
```

Overwriting a item at an index:

```
   NewValue = MyCollection(i) + 1
   MyCollection.Add NewValue, Before:=i
   MyCollection.Remove Index:=i + 1
```

Removing an item:

```
  Collection.Remove Index
  Collection.Remove HashKey
```

The size of a collection:

```
  Cats.Count
```

Iterating over a collection, read-only:

```
  For Each Cat In Cats
    Rem Do something on Cat
  Next
```

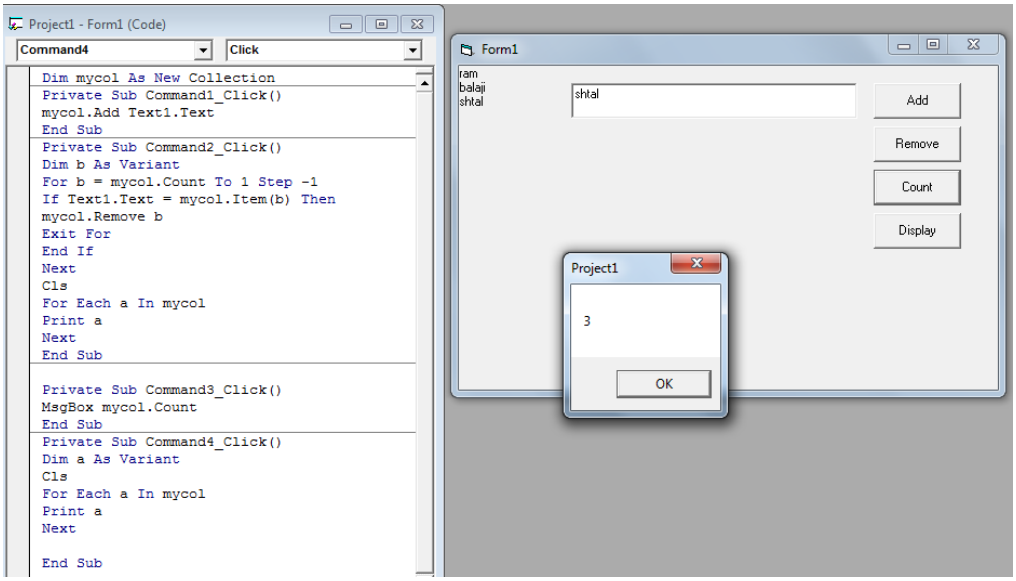
Iterating over a collection, read-write:

```
'Fill the collection
Set MyCollection = New Collection
For i = 1 To 10
  MyCollection.Add i
Next
'Increment each item by 1
For i = 1 To MyCollection.Count
  NewValue = MyCollection(i) + 1
  MyCollection.Add NewValue, Before:=i
  MyCollection.Remove Index:=i + 1
Next
```

Testing the emptiness of a collection:

```
If Cats.Count=0 Then
  '...
End If
```

Example of Collection is:



Procedures

Visual Basic offers different types of procedures to execute small sections of coding in applications. The various procedures are elucidated in details in this section. Visual Basic programs can be broken into smaller logical components called Procedures. Procedures are useful for condensing repeated operations such as the frequently used calculations, text and control manipulation etc. The benefits of using procedures in programming are:
It is easier to debug a program a program with procedures, which breaks a program into discrete logical limits.
Procedures used in one program can act as building blocks for other programs with slight modifications.

Following are the types of procedures

1. Sub Procedures
2. Event Procedures
3. Function Procedures
4. Property procedures

Sub Procedures

A sub procedure can be placed in standard, class and form modules. Each time the procedure is called, the statements between Sub and End Sub are executed. The syntax for a sub procedure is as follows:

```
[Private | Public] [Static] Sub Procedurename [( arglist)]
[ statements]
End Sub
```

arglist is a list of argument names separated by commas. Each argument acts like a variable in the procedure. There are two types of Sub Procedures namely general procedures and event procedures.

Event Procedures

An event procedure is a procedure block that contains the control's actual name, an underscore(_), and the event name. The following syntax represents the event procedure for a Form_Load event.

```
Private Sub Form_Load()
....statement block..
End Sub
```

Event Procedures acquire the declarations as Private by default.

General Procedures

A general procedure is declared when several event procedures perform the same actions. It is a good programming practice to write common statements in a separate procedure (general procedure) and then call them in the event procedure.

In order to add General procedure:

- The Code window is opened for the module to which the procedure is to be added.
- The *Add Procedure* option is chosen from the Tools menu, which opens an Add Procedure dialog box as shown in the figure given below.
- The name of the procedure is typed in the Name textbox
- Under *Type*, *Sub* is selected to create a Sub procedure, *Function* to create a Function procedure or *Property* to create a Property procedure.
- Under *Scope*, *Public* is selected to create a procedure that can be invoked outside the module, or Private to create a procedure that can be invoked only from within the module.

Function Procedures

Functions are like sub procedures, except they return a value to the calling procedure. They are especially useful for taking one or more pieces of data, called *arguments* and performing some tasks with them. Then the functions returns a value that indicates the results of the tasks complete within the function.

The following function procedure calculates the third side or hypotenuse of a right triangle, where A and B are the other two sides. It takes two arguments A and B (of data type Double) and finally returns the results.

```
Function Hypotenuse (A As Double, B As Double) As Double
Hypotenuse = sqr (A^2 + B^2)
End Function
```

The above function procedure is written in the general declarations section of the Code window. A function can also be written by selecting the *Add Procedure* dialog box from the Tools menu and by choosing the required scope and type.

Property Procedures

A property procedure is used to create and manipulate custom properties. It is used to create read only properties for Forms, Standard modules and Class modules.Visual Basic provides three kind of property procedures-Property Let procedure that sets the value of a property, Property Get procedure that returns the value of a property, and Property Set procedure that sets the references to an object.

Questions for Assignment

Q1. Explain the Variables used in Visual Basic.

Dayanand Science College Latur

Class: B.Sc. Ty **Sub: Visual Basic**

Q2. Explain the Operators used in Visual Basic.
Q3. Explain the Data types used in Visual Basic.
Q4. Explain the Control flow statement with suitable example.
Q5. Explain the Looping statement with suitable example.
Q6. Explain the Select Case statement with suitable example.
Q7.Explain the types of Arrays in visual basic.
Q8.Explain the collections in Visual Basic.
Q9. Explain the Procedures in Visual Basic.