Chapter-1- Exploring the Visual Studio IDE, Variables, Operators, and Built-in Functions

1.1. The Visual Studio Development Environment

VB is a user friendly event driven and partly objected oriented programming language for building, testing, debugging, and deploying a variety of applications like:

- database application,
- console application,
- web enabled application, and
- variety of other window based applications

When a program consists of one or more than one event procedures and those procedures accomplish the task to be done by program it is known as event driven programming. Event is any action performed by either the user or by the program itself. For example mouse-click, key press, window resize etc.

To simplify the process of application development, Visual Studio provides an environment that is common to all languages, known as an *integrated development environment (IDE)*. The purpose of the IDE is to enable the developer to do as much as possible with visual tools before writing code. Even as you write code, the IDE will help you in many ways. For example, it underlines errors, it suggests the keywords that may appear at the current place in

your code in a list, and it even provides tools for locating and fixing errors (a process known as *debugging*).

The IDE provides tools for designing, executing, and debugging your applications.

1.2.The Visual Studio Tools

For the purpose of application development, visual studio (VS) provides a number of tools that include the following:

- **The Solution Explorer Window:**

You can use the Solution Explorer tool window to create & manage your solutions and projects and to view & interact with your code.

- **The Form Designer Window:**

The main window of the IDE is the Form Designer, and the gray surface on it is the window of your new application in design mode. Using the Form Designer, you'll be able to design the visible interface of the

application (place various components of the Windows interface on the form and set their properties) and then program the application.

- **The ToolBox Window:**

The toolbox window contains a number of controls that are use to design the user interface. These controls are categorized logically and include buttons, text boxes, radio buttons, lists, and so on.

To place a control on the form, you can double-click the icon for the control.

- **The Properties Window:**

Use this window to view and change the design-time properties and events of selected objects that are located in editors and designers. You can also use the **Properties** window to edit and view file, project, and solution properties. You can find **Properties** Window on the **View** menu. You can also open it by pressing **F4** or by typing **Properties** in the search box.

The **Properties** window displays different types of editing fields, depending on the needs of a particular property. These edit fields include edit boxes, drop-down lists, and links to custom editor dialog boxes. Properties shown in gray are read-only.

1.3. Designing User Interface, UI

User interface is the means of creating communication between the human user and the application. UI design is the process of creating the visual design, layout, and the overall look and feel of an application or website. The goal of UI is creating a simple and efficient means of communication between the user and the application or website.

A UI must be designed to have an easy to use, visually appealing, and efficient interface. And for this some common basic principles must be followed which are listed below.

- Simplicity: this aims to minimize complexity and cognitive load for users. By creating interfaces clean, uncluttered, and easy to understand we can make UIs simple.
- Clarity: it must be explored easily, i.e. easy to use (lengthy texts must be avoided, terms and acronyms must be clear, the text must be readable and understandable, simple layout, limited color palette, simple typography, plenty of white space etc). Hence, clarity focuses on presenting information and content in a clear and easily understandable manner. Use concise and straightforward language, organize content logically, and utilizing appropriate typography and visual cues to enhance readability.
- Consistency: this also supports the user to navigate faster. And for this the interface must have consistent layout, typography (font

formatting), color schemes etc throughout the interface. It ensures that elements and interactions across different screens and sections of the interface are cohesive and predictable. It provides users with a sense of familiarity, making it easier for the users to navigate and understand the interface.

- Feedback: this helps the user to understand what is happening on the interface. It must provide feedback on the user's action whenever important (like by highlighting selected buttons, displaying error messages, providing the format of the input the user should follow (for example date/time and phone number formats) etc).
- Flexibility: this allows the users to customize the UI to suit their needs. This means the interface must be responsive so that it can adapt to different screen sizes and resolutions, and also the users must be able to change color, scheme and layout.
- Usability: for a UI to be usable it must be user-friendly, i.e. it must be easy to understand, navigate, and use. The elements in the UI must be organized in a logical and intuitive way with clear and consistent labeling and navigation.
- Visual hierarchy: this refers to the arrangement of the elements on a screen to prioritize their importance.

1.4. Variables in VB

A variable stores data during a program executes, which is processed with statements. A program is a list of statements that manipulate variables. A variable has a specific name, data type, lifetime, and scope. And variables must be declared before they can be used. But, you can also make use of undeclared variables in a module if you insert 'Option Explicit Off' statement in the General Declaration section of the current module; and this will be applicable in only that module. But if you want to apply to all modules in a VS solution, you can open the solution property window, and make the option explicit OFF. (You can get the solution's property window from the 'Project' Menubar option).

When you use undeclared variables, VS will make an Object variable and you can assign them any type of data.

The following is a general syntax to declare a variable:

```
Dim variable_name As data_type
```

Example: Dim x as Integer, y as double

Dim x, y, sum as Integer

Dim x as integer = 4, y as Double = 7.6, z as char = "g"

1.5.Variable Types

Visual Basic recognizes the following five categories of variables:

- Numeric
- String
- Boolean
- Date
- Object

Numeric Types:

Numeric variables store numbers, and string variables store text. Object variables can store any type of data. For instance, object variables can store integer values; but integer variables are optimized to store integer values.

Numbers can be stored in many formats, depending on the size of the number and its precision.

The following are the number formats that VB supports:

- Integer (there are several Integer data types)
- Decimal
- Single (floating-point numbers with limited precision)
- Double (floating-point numbers with extreme precision)

Decimal, Single, and Double are the three basic data types for storing floating-point numbers (numbers with a fractional part). The Double data

type can represent these numbers more accurately than the Single type and is used almost exclusively in scientific calculations.

The Integer data types store whole numbers. The data type of your variable can make a difference in the results of the calculations. The proper variable types are determined by the nature of the values they represent, and the choice of data type is frequently a trade-off between precision and speed of execution (less-precise data types are manipulated faster).

There are three types of variables for storing integers, and they differ only in the range of numbers each can represent. The more bytes a type takes, the larger values it can hold.

Single- and Double-Precision Numbers:

The Single and Double data type names come from single-precision and double-precision numbers. Double-precision numbers are stored internally with greater accuracy than single-precision numbers.

The Single and Double data types are approximate; you can't represent any numeric value accurately and precisely with these two data types. The problem stems from the fact that computers must store values in a fixed number of bytes, so some accuracy will be lost.

The following table summarizes some numeric types:

Data type:	Size in memory:	Range of values:
Byte	1 byte	0-255
Signed byte (SByte)	1 Byte	-128 - 127
Short (Int16)	2 Bytes	-32,768 – 32,767
Integer (Int32)	4 Bytes	-2,147,483,648 – 2, 147,483, 647
Long (Int64)	8 Byte	-10^{18} - -0^{18}

Boolean Variable:

The Boolean data type stores True/False values. Boolean variables are, in essence, integers that take the value –1 (for True) and 0 (for False). Actually, any nonzero value is considered True. Boolean variables are declared as:

```
Dim x As Boolean
```

And it is initialized to False. Boolean variables are also combined with the logical operators And, Or, Not, and Xor. The Not operator toggles the value of a Boolean variable.

Date Variables:

Date variables store date values that may include a time part (or not), and they are declared with the Date data type and the assignment statements are all valid:

```
Dim regDate As Date
```

```
regDate = #4/23/2000#

regDate = #4/23/2000 12:45:24 PM#

regDate = "March 25, 1999"
```

1.6.Scope of Variables

In addition to a type, a variable has a scope. The scope (or visibility) of a variable is the section of the application that can see and manipulate the variable.

The scope of a variable can be *procedure level, module level, block level,* or *Global level.*

If a variable is declared within a procedure, only the code in the specific procedure has access to that variable; the variable doesn't exist for the rest of the application. When the variable's scope is limited to a procedure, it's called *local variable.*

Suppose that you are coding the handler for the Click event of a button to calculate the sum of all even numbers in the range 0 to 100. One possible implementation is shown below:

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As _
System.EventArguments) Handles Button1.Click

      Dim i As Integer
      Dim Sum As Integer = 0
      For i = 0 to 100 Step 2

            Sum = Sum + i
      Next
      MsgBox "The sum is" & Sum.ToString


End Sub
```

The variables *i* and ***Sum*** are local to the Button1_Click () procedure and hence they are called *procedure level variables*. If you attempt to set the value of the Sum variable from within another procedure, Visual Basic will complain that the variable hasn't been declared.

Sometimes, however, you'll need to use a variable with a broader scope: a variable that's available to all procedures within the same file. This variable, which must be declared outside any procedure, is said to have a module-level scope. In principle, you could declare all variables outside the procedures that use them, but this would lead to problems. Every procedure

in the file would have access to any variable, and you would need to be extremely careful not to change the value of a variable without good reason. Variables that are needed by a single procedure (such as loop counters) should be declared in that procedure.

Another type of scope is the block-level scope. Variables introduced in a block of code, such as an If statement or a loop, are local to the block but invisible outside the block.

The final type of scope is *Global scope*. In some situations, the entire application must access a certain variable. In this case, the variable must be declared as *Public*. *Public* variables have a global scope; they are visible from any part of the application. To declare a public variable, use a *Public* statement in place of a *Dim* statement. Moreover, you can't declare public variables in a procedure. If you have multiple forms in your application and you want the code in one form to see a certain variable in another form, you can use the *Public modifier*.

1.7. Lifetime of Variables

In addition to type and scope, variables have a lifetime, which is the period for which they retain their value. Variables declared as Public exist for the lifetime of the application. Local variables, declared within procedures with the *Dim* or *Private* statement, live as long as the procedure. When the procedure finishes, the local variables cease to exist, and the allocated

memory is returned to the system. Of course, the same procedure can be called again, and then the local variables are re-created and initialized again. If a procedure calls another, its local variables retain their values while the called procedure is running.

Variables declared in a form module outside any procedure take effect when the form is loaded and cease to exist when the form is unloaded. If the form is loaded again, its variables are initialized as if it's being loaded for the first time.

Variables are initialized when they're declared, according to their type. Numeric variables are initialized to zero, string variables are initialized to a blank string, and object variables are initialized to *Nothing*.

1.8.Constants

Some variables (normally they are called *constant identifiers*) don't change value during the execution of a program. These variables are constants that appear many times in your code.

The manner in which you declare constants is similar to the manner in which you declare variables except that you use the Const keyword, and in addition to supplying the constant's name, you must also supply a value, as follows:

```
Const x As Integer = 5
```

Constants also have a scope and can be *Public* or *Private* and the declaration statement is put in the form's declaration section.

1.9. User Defined Data Types

You can create custom data types that are made up of multiple values using *Structures*. A Structure allows you to combine multiple values of the basic data types and handle them as a whole.

To define a structure of, for example, student, you use the *structure* keyword as follows:

```
Structure student

      Dim name As String
      Dim department As String
      Dim age as Integer
      Dim dateOfBirth As Date
End Structure
```

In a structure definition, the Dim and Public keywords are equivalent; but if you replace the *Dim* with *Private*, the member variables will have a limited scope.

A structure declaration must appear outside any procedure; you can't declare a Structure in a subroutine or function and the new structure will become a new data type to the application.

Above, once a new structure is defined which is **student**, you can declare and use its variables as follows:

```
Dim st1 As Student
st1.name = "Abebe Kebede"
st1.age = 24
```

1.10. 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 a number or index. By default, the first element in an array has an index of 0, and the second element 1and so on.

Arrays must be declared with the Dim (or Public) statement followed by the name of the array and the index of the last element in the array in parentheses, as in this example:

```
Dim numList(9) As Integer, studList(100) as student
```

Above, the array *numList* is declared to contain 10 integers and *studList* is declared to contain 101 students.

To assign a value to the first element in the array, you can use the following:

```
numList(0) = 5
studList(0).name = "Abebe Kebede"
```

All elements in an array have the same data type. Of course, when the data type is Object, the individual elements can contain different kinds of data (objects, strings, numbers, and so on).

You can initialize arrays while declaring them as in the following code:

```
Dim numList() = {5, 4, 6}
```

Above, the array variable numList() will have a fixed size of three. But, later on the program you can resize the array by using the **Redim** declaration statement as follows:

```
Redim numList(9)
```

Now, the `numList` array has a size of ten. But the problem of resizing arrays with the Redim statement is that the previous content are deleted.

As in other programming languages, you can declare and use two dimensional arrays as follows:

```
Dim scoreTwo (3, 5) as Integer ' two dimensional array
```

```
Dim scoreThree (3, 5, 9) as Integer ' Three dimensional array
```

The following statement declares and initializes a two dimensional array of 2 by 3: (2 rows and 3 columns)

```
Dim numList( , ) as Integer = {{5,6,7},{1,8,4} }
numList(0, 1) = 6
numList(1, 0) = 1
```

Array class provides methods for sorting arrays, searching for an element, and more.

1.11. Collections

Arrays are the primary structures for storing sets of data. They were the primary storage mechanism for in-memory data manipulation but now they are being replaced with other more flexible and powerful structures which are called *collections.*

A collection is a dynamic data storage structure. You don't have to declare the size of a collection ahead of time. Moreover, the position of the items in a collection is not nearly as important as the position of the items in an array. New items are appended to a collection with the **`Add`** method, while existing items are removed with the **`Remove`** method.

Collections can be categorized as those that store only data (like List and ArrayList collections) and those that store pairs of keys and data values (like Dictionary and HashTable collections).

Lists and ArrayLists:

The ArrayList and List collections allow you to maintain multiple elements, similar to an array; however, Lists and ArrayLists allow the insertion of elements anywhere in the collection, as well as the removal of any element. In other words, they're dynamic structures that can also grow automatically as you add or remove elements. Like an array, the collection's elements can be sorted and searched. You can also remove elements by value, not only by index.

Lists are special types of ArrayList collections in that the items in a List collection must be of the same data type, i.e. they are typed collections; but in case of ArrayList collection, the items can be of any type.

You can declare collections as follows:

```
Dim names As New List(Of String)

Dim myCol As New ArrayList
```

The list item data types are objects of strings, integers, or any other type.

You use the Add method of collections to add/append list items in a collection as follows:

```
names.Add("Abebe") …… correct
names.Add(90)   …… is wrong because 90 is not string!
myCol.Add("Abebe") …… correct
myCol.Add(90)  …… Correct
```

Using Structure with Lists:

A list can contain collections of items which are objects of a structure. Assume that you have defined the following structure:

```
   Structure user
        Dim userName as string
        Dim password as string
        Dim userType as string
   End structure
```

The code below demonstrates how to create, assign value, and add to a list structure objects:

```
Dim userList as New List(Of user)… 'Declares the user list
Dim user1 as New user … 'creates a new user
user1.userName = "Abebe"
user1.password = "Abebe2022"
user1.userType = "student"
userList.Add(user1)
```

You can add the same user multiple times in the list. But if you want to prevent duplication, you can use the **`Contains`** method as follows:

```
If (userList.Contains(user1) then
   MsgBox ("user1 already exists!")
Else
   userList.Add(user1)
```

```
    MsgBox ("The user is inserted successfully")
End If
```

By default, items are appended to the collection. To insert an item at a specific location, use the **Insert** method, which accepts as an argument the location at which the new item will be inserted and an object to insert in the list:

```
userList.Insert(0, user1)
```

You can also add multiple items via a single call to the **AddRange** method. This method appends to the collection of a set of items, which could come from an array or from another list.

The following code creates two lists (userList1 and userList2) and adds userList1 to userList2.

```
Dim userList1 as New List(Of user)
Dim userList2 as New List(Of user)
Dim user1 as New user
user1.userName = "Abebe"
user1.password = "Abebe2022"
user1.userType = "student"
userList1.Add(user1)
        .
        .
        .
userList2.addRange(userList1)
```

To insert a range of items anywhere in the collection, you use the **InsertRange** method instead of the addRange method. Its syntax is the following, where *index* is the index of the collections where the new elements will be inserted, and *objects* is a collection of the elements to be inserted:

```
userList2.InsertRange(index, objects)
```

The GetRange Method of List and ArrayList Class:

Normally, you use index to access each element of a list being O, zero, the index of the first element and **collectionName.Count – 1** the index of the last item. You can also extract a range of items from the list by using the **GetRange** method. Bellow is the syntax of this method:

```
newLst = sourceLst.GetRange(index, count)
```

Above, the method extracts a number of consecutive elements from the **sourceLst** collection and stores them to a new collection, the **newLst** collection, where *index* is the index of the first item to copy, and *count* is the number of items to be copied.

The Dictionary Collection:

The List and ArrayList classes address most of the problems of the Array class, while they support all the convenient array features. Yet, the two

lists, like the Array, have a major drawback: You must access their items by an index value. Another collection, the Dictionary collection, is similar to the List and ArrayList collections, but it allows you to access the items by a meaningful key.

Each item in a Dictionary has a value and a key. The value is the same value you'd store in an array, but the key is a meaningful entity for accessing the items in the collection, and each element's key must be unique. Both the values stored in a Dictionary and their keys can be objects. Typically, the keys are short strings (such as student IDs, ISBN values, product IDs, and so on) or integers.

To create a Dictionary in your code, declare it with the **New** keyword, and supply the type of the keys and values you plan to store in the collection with a statement like the following:

```
Dim Students As New Dictionary(Of Integer, Student)
```

The first argument is the type of the keys, and the second argument is the type of the values you plan to store in the collection. Dictionaries are typed collections, unless you declare them with the Object type. To add an item to the Dictionary, use the **Add** method with the following syntax:

```
Students.Add(key, value)
```

Above, value is the item you want to add (it can be an object of the type specified in the collection's declaration), and key is an identifier you supply, which represents the item. After populating the collection you will use the key to access specific elements of the collection.

```
Dim users As New Dictionary(Of String, String)
users.Add("userName1", "Password1")
users.Add("userName2", "Password2")
users.Add("userName3", "Password3")
MsgBox("The password of "&" userName1 "&" is "&users("userName1"))
```

When working with Dictionaries, you use the **ContainsKey** and **ContainsValue** methods of the dictionary class to check the existence of a key and a value, respectively. If the key or value is present in the dictionary, they return True.

The following code demonstrates this based on the users dictionary above.

```
users.ContainsKey("userName1") … Returns True
users.ContainsValue("password1") … Returns True
users.ContainsKey("userName4") … Returns False
```

The following line of code checks if the given user name, exists in the Dictionary and if so it will update the value; but if the key is absent, it will insert it as a new item.

```
users("userName1") = "password7"
```

The **Values** and **Keys** properties of Dictionary class are used to iterate the values and keys of the dictionary.

Example:

```
'Iterates the user names
Dim item As Object
For Each item In users.Keys
  MsgBox("The current user name is "&item.ToString)
Next

'And the following iterates the passwords
Dim item As Object
For Each item In users.Values
  MsgBox("The current password is "&item.ToString)
Next
```

The HashTable Collection

Another collection, very similar to the Dictionary collection, is the HashTable collection. The HashTable is an untyped Dictionary.

The code bellow creates a Hashtable collection and populates it.

```
Dim users As New HashTable
users("userName1")=57
users("userName2") = "abc"
users(56) = 12
users(45) = "def"
```

Notice in the above code neither the key nor the values are typed!

If an element with the same key you're trying to add exists already, then no new item will be added to the list. Instead, the existing item's value will be changed.

You can always use the **Add** method to add elements to a HashTable collection, and the syntax of this method is identical to the **Add** method of the Dictionary collection. Like the Dictionary collection, it exposes the **Add** method that accepts a key and a value as arguments, a **Remove** method that accepts the key of the element to be removed, and the *Keys* and *Values*
properties that return all the keys and values in the collection, respectively. The major difference between the Dictionary and HashTable collections is that the Dictionary class is a *strongly typed* one, while the HashTable can accept arbitrary objects as values and is not as fast as the Dictionary class. Another very important difference is that the Dictionary collection class is not
serializable. Practically, this means that a Dictionary collection can't be persisted to a disk file with a single statement, which is true for the HashTable collection.

1.12. VB Operators and Expressions

Operators are symbols (characters or keywords) that specify operations to be performed on one or two operands (or arguments). Operators that

take one operand are called unary operators. Operators that take two operands are called binary operators.

Example of unary operators is negation (-) and the logical `NOT` operators.

Arithmetic Operators:

The arithmetic operators perform the standard arithmetic operations on numeric values. The arithmetic operators supported by Visual Basic .NET are: Multiplication (*), regular division (/), integer division (\), Modulo (MOD), Exponentiation (^), Addition (+), Subtraction (-) etc.

Relational Operators:

The relational operators all perform some comparison between two operands and return a Boolean value indicating whether the operands satisfy the comparison. The relational operators supported by Visual Basic .NET are: equality (=), inequality (<>), Less than (<), Greater than (>), Less than or equal to (<=), Greater than or equal to (>=) etc.

String Concatenation Operators:

String concatenation is defined for operands of type String only. The result is a string that consists of the characters from the first operand followed by the characters from the second operand. The & (ampersand) and + (plus) characters signify string concatenation.

Logical Operators:

Logical operators are operators that require Boolean operands. They are: AND, OR, XOR, NOT etc.

Chapter-2- **Control Structures, Modules, and Procedures in VB**

2. 1. **Using Control Structures**

Program flow control structures/statements control the normal program execution steps, top-down execution. This is done either by executing some block of statements based of some condition or by letting the block of statements to execute repeatedly. These control structures are of two categories: *Selection structures* and *Iteration structures.*

Selection Structures:

Selection structures, also called decision structures, can be `If … then` or `Select … Case` decision structures.

Syntax of *If … then* Structure:

```
IF condition Then statement
```

or,

```
IF condition1 Then
   Block_of_statements_1
ELSEIF condition2 Then
   Block_of_statements_2
   ……….
ELSE
   Default_Block_of_statements
END IF
```

An alternative to the efficient but difficult-to-read code of the multiple **ElseIf** structure is the **Select Case** structure, which compares the same expression to different values. The advantage of the **Select Case** statement over multiple **If…Then…ElseIf** statements is that it makes the code easier to read and maintain.

The Select Case structure evaluates a single expression at the top of the structure. The result of the expression is then compared with several values; if it matches one of them, the corresponding block of statements is executed.

Syntax of Select …Case Structure:

```
Select Case expression
Case value1
  statementblock1
Case value2
  statementblock2
. . .
Case Else
  Default_statementblock
End Select
```

Sometimes, when you want to test a variable is larger/smaller than a value, you use the `Is` keyword as follows:

```
Select Case mark
     Case Is > 90
         grade="A"
     Case Is > 80
         grade="B"
     Case Is > 50
         grade="C"
     Case Else
         grade="D"
End Select
```

Iteration Structures:

These are also called loop statements. These statements allow you to execute one or more lines of code repetitively. Many tasks consist of

operations that must be repeated over and over again, and loop statements are an important part of any programming language. Visual Basic supports the following loop statements: *For…Next, Do…Loop, and While…End While*

The *For . . . Next* Loop:

Unlike the other two loops, the For…Next loop requires that you know the number of times that the statements in the loop will be executed.

Syntax:

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

The variable ***start*** is the counter's initial value; ***end*** is the stopping value; ***increment*** specifies by how much the counter increases for the next iteration.

The other variation of this For loop is the ***For Each … Next*** loop and it's used to iterate through the items of a collection or array like in the following:

```
Dim numList() As Integer = {12, 45, 3, 7}
Dim sum as Integer
For Each number As Integer In numList
      sum = sum + number
Next
```

Do While and Do Until Loops:

The Do…Loop statement executes a block of statements for as long as a condition is True or until a condition becomes True. Visual Basic evaluates an expression (the loop's condition), and if it's True, the statements in the loop body are executed. The expression is evaluated either at the beginning of the loop (before any statements are executed) or at the end of the loop (after
the block statements are executed at least once). If the expression is False, the program's execution continues with the statement following the loop. These two variations use the keywords `While` and `Until` to specify how long the statements will be 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
```

Another variations of the **Do** statement is the **Do…Loop** statement, allows you to always evaluate the condition at the end of the loop and the following two syntaxes can be used:

```
Do
    ' statement-block
Loop While condition
```

and,

```
Do
    ' statement-block
Loop Until condition
```

While Loop:

The While…End While loop executes a block of statements as long as a condition is True. The loop has the following syntax:

```
While condition

' statement-block
End While
```

2. 2. The Exit and Continue Statements

The Exit statement allows you to prematurely exit from a block of statements in a control structure, from a loop, or even from a procedure.

For example, following is a code that prints the quotient of two numbers (x and y) which the user inputs from the keyboard and repeats this ten times, and the program exits if the user inputs 0 for y:

```
For i = 0 To 9 Step 1
     x = Convert.ToInt16(InputBox("Input a number"))
     y = Convert.ToInt16(InputBox("Input a number"))
     If (y = 0) Then
         Exit For
     End If
     Label1.Text = (x / y).ToString
 Next i
```

There are similar Exit statements for the Do loop (**Exit Do**), the While loop (**Exit While**), the Select statement (**Exit Select**), and functions and subroutines (**Exit Function** and **Exit Sub**). If the previous loop was part of a function, you might want to display an error and exit not only the loop, but also the function itself by using the **Exit Function** statement.

Sometimes you may need to continue with the following iteration instead of exiting the loop (in other words, skip the body of the loop and continue with the next iteration). In these cases, you can use the **Continue** statement (**Continue For** for `For … Next loops`, **Continue While** for `While loops`, and so on).

The above piece of code is rewritten so as it continues to the next iteration instead of exiting the loop:

```
For i = 0 To 9 Step 1
     x = Convert.ToInt16(InputBox("Input a number"))
     y = Convert.ToInt16(InputBox("Input a number"))
     If (y = 0) Then
         Continue For
     End If
     Label1.Text = (x / y).ToString
 Next i
```

2. 3. **Procedures**

A procedure is a series of statements that tell the computer how to carry out a specific task.

The idea of breaking a large application into smaller, more manageable sections is not new to computing. Using event handlers is just one example of breaking a large application into smaller tasks.

The two types of procedures supported by Visual Basic are *subroutines* and *functions*.

Subroutines perform actions and they don't return any result. Functions, on the other hand, perform some calculations and return a value. This is the only difference between subroutines and functions. Both subroutines and functions can accept arguments (values you pass to the procedure when you call it). Usually, the arguments are the values on which the procedure's code acts.

A subroutine is a block of statements that carries out a well-defined task. The block of statements is placed within a set of **Sub … End Sub** statements and can be invoked by name.

The following subroutine displays the current date in a message box:

```
Sub ShowDate()
    MsgBox("Today's date is " & Now().ToShortDateString)
End Sub
```

To use it in your code, you can just enter the name of the function in a line of its own:

```
ShowDate()
```

It's possible to exit a subroutine prematurely by using the **Exit Sub** statement.

Use subroutines to break your code into smaller, more manageable units and certainly if you're coding tasks that may be used in multiple parts of the application. Note that the **ShowDate()** subroutine above can be called from any event handler in the current form. All variables declared within a subroutine are local to that subroutine. When the subroutine exits, all variables declared in it cease to exist.

A function is similar to a subroutine, but a function returns a result. Because they return values, functions have types. The value you pass back to the calling program from a function is called the return value, and its type determines the type of the function. Functions can accept arguments, just like subroutines. The statements that make up a function are placed in a set of **Function … End Function** statements, as shown below:

```
Function NextDay() As Date
    Dim theNextDay As Date
    theNextDay = Now.AddDays(1)
    Return theNextDay
End Function
```

Functions are called by name; but their return value is usually assigned to a variable. To call the **NextDay()** function above, use a statement like this:

```
Dim tomorrow As Date = NextDay()
```

Sometimes, procedures accept and act upon arguments. Arguments are data items that a caller and a called procedure communicate. In VB, arguments are passed either by value (**ByVal**) or by reference (**ByRef**).

When you pass an argument by value, the procedure sees only a copy of the argument. Even if the procedure changes this copy, the changes aren't

reflected in the original variable passed to the procedure. The benefit of passing arguments by value is that the argument values are isolated from the procedure and only the code segment in which they are declared can change
their values.

If you want to write a function that returns more than a single result, you will most likely pass additional arguments by reference and set their values from within the function's code.

2. 4. **Built-in Functions**

VB provides many functions that implement common or complicated tasks.

There are functions for the common math operations, functions to perform calculations with dates (these are truly complicated operations), financial functions, and many more. When you use the built-in functions, you don't have to know how they work internally — just how to call them and how to retrieve the return value.

Examples of built-in functions:

Name of function:	What it does:	Syntax/ Example:
Ucase(stringObject)	Converts the uppercase equivalent 'stringObject'	Ucase("abc") returns ABC
Lcase(stringObject)	Converts the lowercase equivalent 'stringObject'	Lcase("ABC") returns abc
Rtrim(stringObject)	Removes trailing blank spaces from a string.	Rtrim(" Abc ") returns " Abc"
Ltrim(stringObject)	Removes leading blank spaces from a string.	Ltrim(" Abc ") returns "Abc "
Trim(stringObject)	Removes both trailing and leading blank spaces from a string.	Trim(" Abc ") returns "Abc"
Chr(integerValue)	Returns the character for the specified character code	Chr(65) returns "A"
Asc(characterValue)	Returns the character code for the given character	Asc("A") returns 65

Chapter-3- **Working with Forms and Classes**

3. 1. **Form Basics**

In Visual Basic, the form is the container for all the controls that make up the user interface. When a Visual Basic application is executing, each window it displays on the Desktop is a form. The terms form and window describe the same entity. A window is what the user sees on the Desktop when the application is running. A form is the same entity at design time. The proper term is Windows form, as opposed to web form.

The term windows form includes both typical Windows forms and dialog boxes, which are simple forms you use for very specific actions, such as to prompt the user for a particular piece of data or to display critical information. A dialog box is a form with a small number of controls, no menus, and usually an OK and a Cancel button to close it. Forms have a built-in functionality that is always available without any programming effort on your part. You can move a form around, resize it, and even cover it with other forms. You do so with the mouse or with the keyboard through the Control menu.

3. 2. Common Form Object Properties

You can change the look of any window or dialog box through the following properties of the Form object.

AcceptButton and CancelButton

These two properties let you specify the default Accept and Cancel buttons of the form. The Accept button is the one that's automatically activated when you press Enter, no matter which control has the focus at the time; it is usually the button with the OK caption. Likewise, the Cancel button is the one that's automatically activated when you hit the Esc key; it is usually the button with the Cancel caption. To specify the Accept and Cancel buttons on a form, locate the *AcceptButton* and *CancelButton* properties of the form and select the corresponding controls from a drop-down list, which contains the names of all the buttons on the form.

AutoScroll

The AutoScroll property is a True/False value that indicates whether scroll bars will be automatically attached to the form if the form is resized to a point that not all its controls are visible.

FormBorderStyle

The FormBorderStyle property determines the style of the form's border. You can make the form's title bar disappear altogether by setting the

form's FormBorderStyle property to FixedToolWindow, the ControlBox property to False, and the Text property (the form's caption) to an empty string.

ControlBox

This property is also True by default. Set it to False to hide the control box icon and disable the Control menu. Although the Control menu is rarely used, Windows applications don't disable it. When the ControlBox property is False, the three buttons on the title bar are also disabled. If you set the Text property to an empty string, the title bar disappears altogether.

MinimizeBox, MaximizeBox

These two properties, which specify whether the Minimize and Maximize buttons will appear on the form's title bar, are True by default. Set them to False to hide the corresponding buttons on a form's title bar.

MinimumSize, MaximumSize

These two properties read or set the minimum and maximum size of a form. When users resize the form at runtime, the form won't become any smaller than the dimensions specified by the MinimumSize property or any larger than the dimensions specified by the MaximumSize property. The MinimumSize property is a Size object, and you can set it with a statement like the following:

```
Me.MinimumSize = New Size(400, 300)
```

Or you can set the width and height separately:

```
Me.MinimumSize.Width = 400
Me.MinimumSize.Height = 300
```

The default value of both properties is (0, 0), which means that no minimum or maximum size is imposed on the form and the user can resize it as desired.

TopMost

This property is a True/False setting that lets you specify whether the form will remain on top of all other forms in your application. Its default value is False, and you should change it only on rare occasions. Some dialog boxes, such as the Find & Replace dialog box of any text-processing application, are always visible, even when they don't have the focus.

3. 3. Anchoring and Docking

A common issue in form design is the design of forms that can be properly resized. For instance, you might design a nice form for a given size, but when it's resized at runtime, the controls are all clustered in the upper-left corner.

Visual Studio provides several techniques for designing forms that scale nicely. The two most important of them are the Anchor and Dock properties.

The Anchor property lets you attach one or more edges of the control to corresponding edges of the form. The anchored edges of the control maintain the same distance from the corresponding edges of the form.

In addition to the Anchor property, most controls provide a Dock property, which determines how a control will dock on the form. The default value of this property is None.

3. 4. **Form Events**

The Form object triggers several events. The most important are Activated, Deactivate, FormClosing, Resize, and Paint.

The Activated and Deactivate Events:

When more than one form is displayed, the user can switch from one to the other by using the mouse or by pressing **`Alt + Tab`**. Each time a form is activated, the Activated event takes place. Likewise, when a form is activated, the previously active form receives the Deactivate event.

The FormClosing and FormClosed Events:

The *FormClosing* event is fired when the user closes the form by clicking its *Close* button. If the application must terminate because Windows is shutting down, the same event will be fired. Users don't always quit applications in an orderly manner, and a professional application should behave gracefully under all circumstances. The same code you execute in the application *Exit* command must also be executed from within the *FormClosing* event. For example, you might display a warning if the user has unsaved data, you might have to update a database, and so on. Place the code that performs these tasks in a subroutine and call it from within your menu's *Exit* command as well as from within the *FormClosing* event's handler.

3. 5. Loading and Showing Forms

Most practical applications are made up of multiple forms and dialog boxes. One of the operations you'll have to perform with multiform applications is to load and manipulate forms from within other forms' code. For example, you might want to display a second form to prompt the user for data specific to an application. You must explicitly load the second form and read the information entered by the user when the auxiliary form is closed. Or you might want to maintain two forms open at once and let the user switch between them.

The `show` method is used to open/show a form and the following is its syntax:

```
formName.show()
```

Forms can be opened in two manners: modeless or modal. The `show` method opens a form in a modeless manner. When a form is opened in a modeless manner the user can switch between the two forms; But, when a form is opened in a modal manner, the newly opened form will be on top of the old one and the user can't access the old one unless the new one is closed. The `ShowDialog()` method is used to open a modal form by the following syntax:

```
formName.showDialog()
```

A dialog box is an example of a modal form. And when a dialog box is opened, it is very customary to make it NOT resizable by setting its border style property to FixedDialog as in the following code:

```
formName. FormBorderStyle = Windows.Forms.FormBorderStyle.FixedDialog
```

When you're finished with the second form, you can either close it by calling its `Close` method or hide it by calling its `Hide` method. The `Close` method closes the form, and its resources are returned to the system. The

`Hide` method sets the form's `Visible` property to `False`; you can still access a hidden form's controls from within your code, but the user can't interact with it.

3. 6. Sharing Variables Between Forms

The preferred method for two forms to communicate with each other is through public variables. These variables are declared in the form's declarations section, outside any procedure, with the keyword `Public`.

For example, the integer variable x bellow is declared in form1's declaration section and it can be accessed from another from code as follows: Note that x is access by prefixing it with the name of the form it is declared in, i.e. form1.

```
Public x as integer
```

In another form's code,

```
form1.x = 5
```

3. 7. Classes and Objects in Visual Basic

Classes are practically synonymous with objects and they're at the very heart of programming with Visual Basic. The controls you use to build the visible interface of your application are objects.

You manipulate all objects by setting their properties and calling their methods.

When you create a variable of any type, you're creating an instance of a class. The variable lets you access the functionality of the class through its properties and methods. Even base data types are implemented as classes (the `System.Integer` class, `System.Double`, and so on). An integer value, such as 3, is an instance of the `System.Integer` class, and you can call the properties and methods of this class by using its instance.

A class can be thought of as a program that doesn't run on its own; it's a collection of properties and methods that must be used by another application. We exploit the functionality of the class by creating a variable of the same type as the class and then calling the class's properties and methods through this variable. The methods and properties of the class, as well as its events, constitute the class's interface.

3. 8. Working with Classes and Objects

You can create a class following the step below:

1. Right click on the solution name in the solution explorer window
2. Click the Add menu
3. Click the Class option
4. Then from the dialog box, select 'Class' and give it a name, for example, 'user'

5. Then click Add button

The above step creates an empty class with the following header and footer:

```
Public Class user


End Class
```

The data parts that you will use to describe the class are called member variables, or more properly properties or fields of the class. You can define, for example, the two class properties (username and password) as:

```
Public Class user
      Public userName as String
      Public password as String
End Class
```

Also, you can define an object of the above class and access its properties as follows:

```
Dim user1 as New user
user1.userName = "Abebe"
user1.password = "abebe@org"
MsgBox("User Name is:" & user1.userName & "Password is:" & user1.password)
```

Fields and Properties:

Fields and properties are the same things with some exceptions. A field is simply the data part that is declared with the Public modifier and is then directly accessible from everywhere in your code; whereas, a property is implemented with a **`Property Procedure`**.

Property procedures are a special type of procedure that contains a Get and a Set section (frequently referred to as the property's getter and setter, respectively). The Set section of the procedure is invoked when the application attempts to set the property's value; the Get section is invoked when the application requests the property's value. The value passed to the property is usually validated in the Set section and, if valid, is stored to a local variable. The same local variable's value is returned to the application when it requests the property's value, from the property's Get section.

Like fields, Properties are also accessible from everywhere in your code, but the difference is that when a value is assigned to a field, you can't validate that value from within the class code; but in case of properties, as far as they are implemented with property procedures, they can be validated from within the class code.

The above user Class can be redefined so that it checks if the assigned password length is less than five, and if so, it displays a message box:

```
Public Class user
      Public userName as String
      Private password as String
      Property MypassWord() As String
         Get
             MypassWord = passWord
         End Get
         Set(ByVal value As String)
             If value.Length < 5 Then
                 MsgBox("Too short password length!")
             Else
                 passWord = value
             End If
         End Set
      End Property
End Class
```

Above, `password` is the local variable and `MypassWord` is the property that is used to access the password variable as in the following code:

```
Dim user1 as New user
user1.userName = "Abebe"            'Correct syntax
user1.password = "abebe@org"        'Wrong syntax
user1.Mypassword = "abebe@org"      'Correct syntax & the setter is invoked
MsgBox("User Name is" & user1.userName & "Password is" & user1.Mypassword)
```

In the above example, it is the class itself (not your code) that displays the error message box. And this means, the programmer that uses the class has no right to change the message displayed in the message dialog box if the class is to be used from different applications. To let the programmer change the message from his own code as per his application's context, you have to implement the class in such a way that it throws an `InvalidArgument` exception, as in the following code:

```
Public Class user
      Public userName as String
      Private password as String
      Property MypassWord() As String
         Get
             MypassWord = passWord
         End Get
         Set(ByVal value As String)
             If value.Length < 5 Then
                 Dim argExcep As New ArgumentException()
                  Throw argExcep
             Else
                 passWord = value
             End If
         End Set
      End Property
End Class
```

And your code that accesses the class must be modified as follows:

```
Dim user1 as New user
Try
      user1.password = "abebe@org"
Catch ex As ArgumentException
      MsgBox ("You can type your own error message here")
End Try
```

Chapter-4- **Working with Database**

4. 1. **General Overview**

A database is a container for storing relational, structured information. The same is true for a file or even for the file system on your hard disk. What makes a database unique is that it is designed to preserve relationships and make data easily retrievable.

It is possible to create a database for storing products and invoices and add new invoices every day. In addition to just storing information, you should also be able to retrieve invoices by period, retrieve invoices by customer, or retrieve invoices that include specific products. Unless the database is designed properly, you won't be able to retrieve the desired information efficiently.

Databases are maintained by special programs, such as Microsoft Office Access and SQL Server. These programs are called database management systems (DBMSs), and they're among the most complicated applications. A fundamental characteristic of a DBMS is that it isolates much of the complexity of the database from the developer. Regardless of how each DBMS

stores data on disk, you see your data organized in tables with relationships between tables. To access or update the data stored in the database, you use a special language, the Structured Query Language (SQL). Unlike other areas of programming, SQL is a truly universal language, and all major DBMSs support it.

One popular and common type of database system is relational database. It is called relational because it is based on relationships among the data it contains. The data is stored in tables, and tables contain related data, or entities, such as people, products, orders, and so on. Of course, entities are not independent of each other. For example, orders are placed by specific customers, so the rows of the Customers table must be linked to the rows of the Orders table that stores the orders of the customers.

4. 2. **Introductions to Active X Data Objects (ADO.NET)**

The component of the Framework we use to access databases is known as ADO.NET (ADO stands for Active Data Objects) and it provides two basic methods of accessing data: *stream-based* data access, which establishes a stream to the database and retrieves the data from the server, and *set-based* data access, which creates a special data structure at the client and fills

it with data. This structure is the `DataSet`, which resembles a section of the database: It contains one or more `DataTable` objects, which correspond to tables and are made up of `DataRow` objects. These `DataRow` objects have

the same structure as the rows in their corresponding tables. `DataSets` are populated by retrieving data from one or more database tables into the corresponding `DataTables`. As for submitting the data to the database with the stream-based approach, you must create the appropriate INSERT/UPDATE/DELETE statements and then execute them against the database.

The stream-based approach relies on the DataReader object, which makes the data returned by the database available to your application. The client application reads the data returned by a query through the DataReader object and must store it somehow at the client.

The set-based approach also uses the same object, DataReader object, but provides higher level of abstractions, like hiding the work needed to create a link to a database, retrieve data, and store it to the client's computer memory.

4. 3. The Basic Data-Access Classes

A data-driven application should be able to connect to a database and execute queries against it. The selected data is displayed on the appropriate interface, where the user can examine it or edit it. Finally, the edited data is submitted to the database.

The following is the cycle of a data-driven application:

1. Retrieve data from the database.

2. Present data to the user.

3. Allow the user to edit the data.

4. Submit changes to the database.

Designing the appropriate interface for navigating through the data can be quite a task. Developing a functional interface for editing the data at the client is also a challenge, especially if several related tables are involved. We must also take into consideration that there are other users accessing the same database.

To connect to a database, you must create a **`Connection`** object, initialize it, and then call its **`Open`** method to establish a connection to the database.

The Connection object is the channel between your application and the database; every command you want to execute against the same database must use this Connection object. When you're finished, you must close the connection by calling the Connection object's **`Close`** method. Because ADO.NET maintains a pool of Connection objects that are reused as needed, it's very important that you keep connections open for the shortest possible time.

The object that will actually execute the command against the database is the **`Command`** object, which you must configure with the statement you want to execute and associate with a Connection object. To execute the statement, you can call one of the Command object's methods such as the:

- **ExecuteReader** method (returns a **DataReader** object that allows you to read the data returned by the selection query, one row at a time) and
- **ExecuteNonQuery** method (to execute a statement that updates a database table but doesn't return a set of rows but the number of rows affected by the statement).

To put it shortly, ADO.NET provides three core classes for accessing databases: the Connection, Command, and DataReader classes. There are more data access–related classes, but they're all based on these three basic classes.

4. 4. The Connection Class

The **Connection** class is an abstract class, so you can't use it directly. Instead, you must use one of the classes that derive from the Connection class (which are *SqlConnection, OracleConnection, and OleDbConnection*).

Likewise, the **Command** class is an abstract class with three derived classes (*SqlCommand, OracleCommand, and OleDbCommand*).

The SqlConnection and SqlCommand classes belong to the **SqlClient** namespace, which you must import into your project via the following statement:

```
Imports System.Data.SqlClient
```

To connect the application to a database, the Connection object must know the following:

- the name of the server on which the database resides,
- the name of the database itself, and
- the credentials that will allow it to establish a connection to the database (which can be username and password or a Windows account that has been granted rights to the database).

You obviously have to know what type of DBMS you're going to connect to (sql server, oracle, access etc) so you can select the appropriate Connection class.

The following is an example on how to create and initialize a connection object to a database managed by sql server DBMS:

```
Dim cnn As New SqlConnection("server = localhost;" &
                             "data source = mypc\myinstance;"
                             & "database = mystud;" &
                             "integrated security = sspi")
```

Above, *localhost* is a universal name for the local computer, *mypc\myinstance* is the name of the database server instance that manages the database, *mystud* is the name of the database. The *integrated security* parameter is used to tell the connection object about

how to authenticate the client that requests connection to the database. It can be either of the two types:

- If it is set to **True** or equivalently **sspi** (security support provider interface), then it indicates that NT security should be used, i.e. Windows authentication which is based on the local machine's logged in user account.
- If it is set to **False**, sql server authentication is used.

An alternate method of setting up a Connection object is to set its **ConnectionString** property as follows:

```
Dim cnn As New SqlConnection
cnn.ConnectionString = "server = localhost;" &
                       "data source = mypc\myinstance;"
                       & "database = mystud;" &
                       "integrated security = sspi"
```

4. 5. The Command Class

The second major component of the ADO.NET model is the Command class, which allows you to execute SQL statements against the database.

To execute a SQL statement against a database, you must initialize a Command object and set its Connection property to the appropriate Connection object. There are also other properties such as **CommandType** (which tells how the command string is to be interpreted) and **CommandText** (which is the actual sql statement that must run on the database).

The **CommandType** property can be set to the following values:

- Text : if the command is sql statement,
- storedprocedure: name of stored procedure, if the command is given as a stored procedure , or
- TableDirect: name of a database table, if the command is to read all records of a table

The following code creates and sets up a command object:

```
Dim cmd1 As New SqlCommand()
Dim depReader as SqlDataReader
cmd1.connection = cnn
cmd1.commandType = commandType.text
cmd1.commandText = "select title from department"
depReader = cmd1.ExecuteReader
```

The above code reads department titles and assigns it to the SqlDataReader object **depReader**.

The commandText property can be set to a sql statement that needs a parameter, for instance when you want to read titles of departments in ONLY the technology faculty. To do so you use a parameter and the commandText property will be:

```
cmd1.commandText = "select title from department where faculty=@faculty"
```

@faculty above is a parameter that must be set to a value before executing the sql command. To set a value to this parameter, you can use the following among other several methods:

```
cmd1.Parameters.AddWithValue("@faculty", "Technology")
```

In addition to the `ExecuteReader` method of the command object which returns a set of records, there are also other methods that include `ExecuteScalar` (which return a single value which for example can be the result of the sql 'count' function) and `ExecuteNonQuery` (which is used when the sql command is an update command and after executing it, the number of updated rows will be returned).

4. 6. **DataSet and DataAdapter**

Working with a database from your application is not as such difficult task as long as most issues are to a large extent abstracted by the Connection, Command, and DataReader classes.

The problem with these classes is that they don't offer a consistent method for storing the data at the client. The approach of converting the

data into business objects and working with classes is fine, but you must come up with a data-storage mechanism at the client.

You can store the data in a ListBox control. You can also create a List of custom objects. The issue of storing data at the client isn't pressing when the client application is connected to the database and all updates take place in real time. The problem comes when the connection to the database from your application is broken. Still, you can update, delete, and insert the local data that was previously read from the database but you can't save it to the database immediately. For such a problem, ADO.net provides a mechanism which is called DataSet to store the modification made on the data in the client machine and saves it to the database on the server when there is connection.

You can think of the DataSet as a small database that lives in the local memory. It's not actually a database, but it's made up of related tables that have the same structure as database tables. The similarities end there, however, because the DataSet doesn't impose all types of constraints, and you can't exploit its data with SQL statements. It's made up of DataTable objects, and each DataTable in the DataSet corresponds to a separate query. Like database tables, the DataTable objects consist of DataColumn and DataRow objects.

The real power of the DataSet is that it keeps track of the changes made to its data. It knows which rows have been modified, added, or deleted,

and it provides a mechanism for submitting the changes automatically. It is actually another class, the DataAdapter class, that submits the changes made to the data.

DataSets are filled with DataAdapters, and there are two ways to create a DataSet:

- You can use the visual tools of Visual Studio or
- create a DataSet entirely from within your code.

DataSets created at runtime are not typed, because the compiler doesn't know what type of information you're going to store in them. DataSets created at design time with the visual tools are strongly typed, because the compiler knows what type of information will be stored in them.

To use DataSets in your application, you must first create a DataAdapter object, which is the preferred technique for populating the DataSet. The DataAdapter is nothing more than a collection of Command objects that are needed to execute the various SQL statements against the database.

These command objects are:

`InsertCommand`	A Command object that's executed to insert a new row
`UpdateCommand`	A Command object that's executed to update a row
`DeleteCommand`	A Command object that's executed to delete a row
	A Command object that's executed to retrieve selected

The DataAdapter class performs the two basic tasks of a data-driven application:

- It retrieves data from the database to populate a DataSet and
- submits the changes to the database

To populate a DataSet, use the Fill method, which fills a specific DataTable object. There's one DataAdapter per DataTable object in the DataSet, and you must call the corresponding Fill method to populate each DataTable.

To submit changes to the database, use the Update method of the appropriate DataAdapter object. The Update method is overloaded, and you can use it to submit a single row to the database or all edited rows in a DataTable. The Update method uses the appropriate Command object to interact with the database.

4. 7. **The DataGridView Control**

Data binding relieves you from having to map field values to controls on the form when a row is selected and from moving values from the controls
back to the DataSet when a row is edited.

In addition to binding simple controls such as TextBox controls to a single field, you can bind an entire column of a DataTable to a list control, such

as the ListBox or ComboBox control. And of course, you can bind an entire DataTable to a special control, the DataGridView control.

Grids display data in a series of rows and columns. The available DataGrid displays information from a table in a database it is bound to. Data from the table fills the rows and columns, in the same fashion that it appears in the table.

The `DataGrid` can be bound to data with multiple related tables, and if navigation is enabled on the grid, the grid will display expanders in each row. An expander allows navigation from a parent table to a child table.

You can build a data-browsing and data-editing application by binding a DataTable to a DataGridView control without a single line of code.