ALLAMA IQBAL OPEN UNIVERSITY

AIOU LIBRARY
aioulibrary.com

Name:

Registration No.:

Course Code: 3435

Program: Post Graduate

Semester: Autumn 2026

WhatsApp Orders: +92 324 5836626

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Visual Basic & Database Interface (3435) • Level: Post Graduate

Semester: Autumn 2026

Assignment No. 1

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant units of the course book, understand the concepts and the logic of every program, build and run each project on a computer in Visual Basic 6.0, and then write the assignment in their own words. Copying this guide word for word is not allowed; the printed instructions on the assignment booklet also require answers in the student's own words and warn against plagiarism. All Visual Basic programs in this guide are complete and self-contained: create a Standard EXE project, place the controls named in each answer, set the properties listed in the tables, type the code exactly, and run the project to confirm the sample run before writing the answer in the assignment.

Question 1: A First Visual Basic Project — Welcome Message, Saving, Running and Execution (20 Marks)

Question as printed: a) Create a simple Visual Basic project that displays a welcome message on a form. b) Save and run the project, and explain the steps involved in its execution. (20)

WhatsApp Orders: +92 324 5836626

Introduction

Visual Basic (VB) is Microsoft's event-driven programming language and integrated development environment (IDE) for building Windows applications with a graphical user interface (GUI). Instead of writing a long list of instructions that the computer follows from top to bottom, the VB programmer first draws the interface — forms, labels, text boxes and buttons — and then writes small pieces of code, called event procedures, which run only when something happens: the user clicks a button, types in a box, or the form is loaded. This question asks for the very first program almost every VB student writes: a project that displays a welcome message on a form. Although the program is small, completing it teaches the complete working cycle of Visual Basic — starting the IDE, designing a form, setting properties, writing code, saving the project in its several files, running it, understanding what the system does behind the scenes during execution, and finally compiling a stand-alone executable. This answer therefore gives the full project first, and then explains saving, running and execution step by step, because part (b) carries as much weight as part (a).

Part (a): Creating the Welcome Message Project

Start Visual Basic 6.0 from the Start menu. The New Project dialog appears; select Standard EXE and click Open. Visual Basic creates a project named Project1 containing a single blank form named Form1, and displays the IDE: the Toolbox on the left holding the controls, the Form Designer in the centre, the Properties window on the right, the Project Explorer listing the files of the project, and the Code window, opened by double-clicking the form or a control. The first discipline of VB programming is to name things properly before writing code, because code

refers to controls by name. Click the form, and in the Properties window set its Name to frmWelcome and its Caption to 'Welcome Program'. The Caption is the text shown in the title bar, while the Name is the identifier used in code; confusing the two is the commonest beginner's mistake.

From the Toolbox, place two controls on the form. First, double-click the Label tool and set the label's Name to lblMessage, its Caption to an empty string, its Alignment to 2 – Center, and its Font to a large bold size so the message is prominent; also set AutoSize or stretch the label across the form. Second, place a CommandButton, set its Name to cmdShow and its Caption to 'Show Welcome Message'. Optionally add a second button, cmdExit with Caption 'Exit', so the user can close the program politely instead of using the window's close cross. The design is now complete: the interface consists of one form, one label and two buttons, and the layout follows the golden rule of VB design — the user sees what to press, and the program responds in a clearly visible place.

AIOU LIBRARY aioulibrary.com

Table 1: Controls and properties for the welcome project

Control	Name	Key properties set
Form	frmWelcome	Caption = Welcome Program; StartUpPosition = 2 – CenterScreen
Label	lblMessage	Caption = (blank at design time); Alignment = 2 – Center; Font = MS Sans Serif, Bold, 14 pt; ForeColor = dark blue

CommandButton	cmdShow	Caption = Show Welcome Message; Default = True (responds to Enter key)
CommandButton	cmdExit	Caption = Exit; Cancel = True (responds to Esc key)

Now write the code. Double-click cmdShow to open the Code window inside its Click event procedure, and type the code below. Two ways of showing the message are given deliberately: the label method, which is what the question literally asks — the message displayed on the form — and the MsgBox method, which displays the message in a dialogue box. A complete answer should show the label method as the main solution and mention MsgBox as an alternative output method. The Form_Load procedure also sets an initial instruction in the label, which demonstrates that code can run automatically when the form loads, before any click takes place.

Code 1: frmWelcome — complete code for the welcome project

```
Private Sub Form_Load()
   lblMessage.Caption = "Press the button to see the welcome message."
End Sub


Private Sub cmdShow_Click()
   lblMessage.Caption = "Welcome to Visual Basic Programming!"
   MsgBox "Welcome to Allama Iqbal Open University!", _
      vbInformation, "Welcome"
```

```
End Sub


Private Sub cmdExit_Click()
   Unload Me
   End
End Sub
```

Read the code line by line. Private Sub cmdShow_Click() declares an event procedure: it runs automatically when the user clicks the button whose Name is cmdShow; the name of the procedure is built from the control name and the event name joined by an underscore, and Visual Basic itself creates this shell when the control is double-clicked, so the programmer only fills in the body. The statement lblMessage.Caption = "Welcome..." is an assignment to a property at run time: the dot notation Object.Property selects the Caption property of the lblMessage object and gives it a new value, and the form immediately repaints the label. MsgBox is a built-in function that displays a modal dialogue; vbInformation is a built-in constant selecting the information icon, and the third argument is the dialogue's title. In cmdExit_Click, Unload Me removes the current form from memory — Me is Visual Basic's keyword for the form that owns the code — and End stops execution cleanly.

Figure: Life cycle of a Visual Basic project — from design to executable



Figure 1: Life cycle of a Visual Basic project — design, properties, code, save, run, and make EXE.

Part (b): Saving the Project

A Visual Basic application is not one file but a project: a collection of files held together by the project file. Saving must therefore be done in the right order and in one dedicated folder, otherwise the project will complain about missing files when it is reopened on another machine — a practical point of great importance for students who prepare assignments on one computer and demonstrate them on another. The steps are: (1) create a folder, for example C:\VBPrograms\Welcome, before saving; (2) choose File → Save Project, or press Ctrl+S; (3) Visual Basic first asks for each form file — save the form as frmWelcome.frm; (4) it then asks for the project file — save it as Welcome.vbp; (5) if modules are later added they are saved as .bas files, and class modules as .cls files. The .frm file stores the form's design and its code (and a companion .frx file is created automatically if the form contains pictures or other binary data), while the .vbp file stores the list of files, references and project settings. To reopen the work, always open the .vbp file, never the .frm file alone, because only the project file knows the start-up form and the references.

Running the Project and the Steps of Execution

To run the project inside the IDE, press F5, click the Start button on the toolbar, or choose Run → Start. Running with full compile (Run → Start With Full Compile, or Ctrl+F5) is better practice because it checks the whole code at once. What then happens, step by step, is the execution story part (b) asks for. First, Visual Basic compiles the project: it checks every line for syntax errors — misspelt keywords, missing End Sub, undeclared variables if Option Explicit is on —

and reports the first error with the offending line highlighted; no program runs until all compile errors are removed. Second, the compiled instructions are loaded into memory together with the VB run-time library, which supplies built-in functions such as MsgBox. Third, the start-up form (set under Project → Project Properties → Startup Object) is loaded: its Load event fires, so Form_Load runs and the label receives its instruction text, and then the form is displayed and its Activate event fires. Fourth, the program enters the event loop, also called the message loop: it now waits, doing nothing, while Windows watches the mouse and keyboard. Execution in an event-driven program is therefore not a straight line but a wait-and-respond cycle.

Fifth, when the user clicks Show Welcome Message, Windows places a click message in the queue, Visual Basic matches it to the cmdShow control, and the cmdShow_Click procedure executes: the label's Caption changes on the form and the MsgBox dialogue appears, which is modal — the program pauses until the user presses OK. Sixth, control returns to the event loop and the program waits again; this click–respond–wait cycle repeats for as long as the application runs. Seventh, when the user clicks Exit, cmdExit_Click runs, Unload Me fires the form's Unload event and removes the form from memory, End terminates the program, and the IDE returns to design mode. Eighth, debugging is part of execution: if a run-time error occurs — for example division by zero — Visual Basic stops on the offending line in break mode, where the student can inspect variable values in the Immediate window and correct the code. Finally, to produce the finished product, choose File → Make Welcome.exe: Visual Basic compiles the project into a stand-alone executable file which can be run by double-clicking, without opening the IDE, although the VB run-time library (MSVBVM60.DLL) must be present on the target computer.

Sample Run / Output

```
(Form opens)  Press the button to see the welcome message.
[User clicks: Show Welcome Message]
(Label)       Welcome to Visual Basic Programming!
(Dialogue)    Welcome to Allama Iqbal Open University!   [ OK ]
[User clicks: Exit]  Program ends and the form closes.
```

Conclusion

In conclusion, this smallest of projects contains the whole method of Visual Basic in miniature: draw the interface, name the controls meaningfully, set properties at design time, change properties again in code at run time, save the form and project files together in one folder, and run the program, whose execution consists of compilation, loading of the start-up form, and then an event loop in which procedures such as cmdShow_Click fire in response to the user. A student who can explain not only what the code says but also what the system does when F5 is pressed — compiling, loading, waiting, responding and unloading — has understood the foundation on which every larger program in this course, including the database applications of Assignment 2, is built.

AIOU LIBRARY aioulibrary.com

Question 2: Student Information Form — Labels, Text Boxes, Command Button and Display Code (20 Marks)

Question as printed: a) Design a form with labels, textboxes, and a command button to input and display student information. b) Write code to display the entered data when the button is clicked. (20)

Introduction

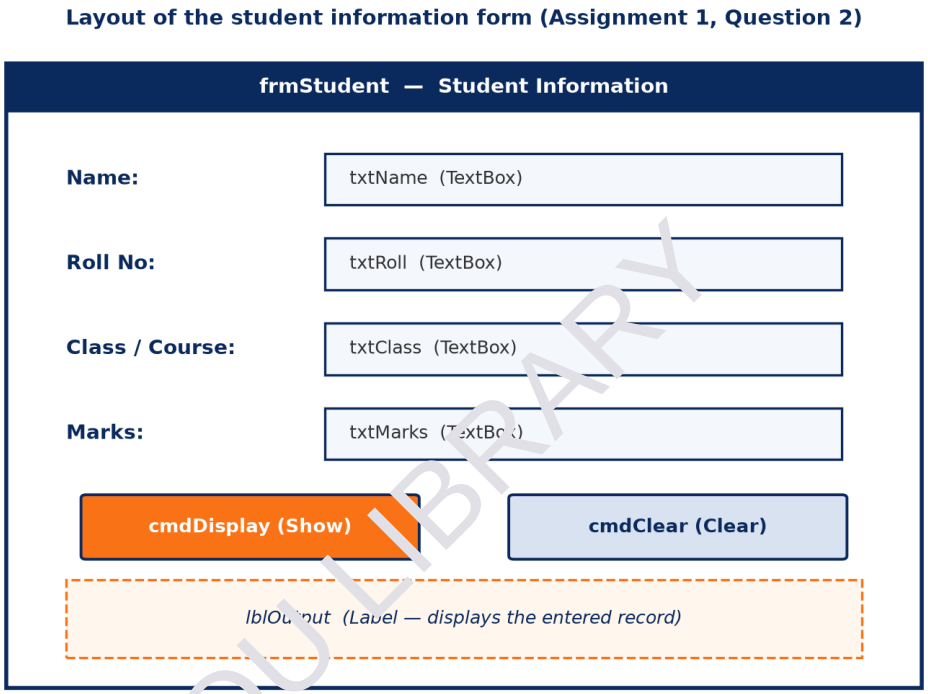
Almost every business and educational application begins as a data-entry form: a screen that invites the user to type values into labelled boxes and then does something with those values when a button is pressed. The student information form in this question is the classic first example. It teaches the three most fundamental controls of Visual Basic — the Label, which displays text the user cannot edit; the TextBox, which accepts typed input; and the CommandButton, which starts the action — and it teaches the central idea of GUI programming, namely that input lives in the Text property of a text box and output is produced by reading that property in the button's Click event. Part (a) designs the form carefully, because a form that is badly aligned or ambiguously labelled produces bad data however good the code is; part (b) then writes the display code, together with the small but important extras — clearing the form, checking for empty boxes, and returning the cursor to the first box — that separate a classroom exercise from a robust program.

AIOU LIBRARY aioulibrary.com

Part (a): Designing the Form

Create a new Standard EXE project and rename Form1 as frmStudent with Caption 'Student Information'. The form records four items about a student: Name, Roll Number, Class/Course and Marks. For each item place one Label and one TextBox in a row: the label on the left announces what is required, and the text box on the right receives the typing. Keeping every label the same width and every text box left-aligned in a column — using Format → Align and Make Same Size — gives the tidy, professional appearance examiners expect. Below the four rows place two CommandButtons side by side: cmdDisplay ('Show Information') to display the record, and cmdClear ('Clear') to empty the boxes for the next student. At the bottom place one wide Label, lblOutput, with a border (BorderStyle = 1 – Fixed Single), which will act as the display area for the entered record. Set the TabIndex property of the four text boxes to 0, 1, 2 and 3 in order, so the user can move from box to box with the Tab key in the natural sequence — a small property that greatly improves the feel of a form.

AIOU LIBRARY aioulibrary.com



AIOU LIBRARY
aioulibrary.com

Figure 1: Layout of the student information form — four label/text box pairs, two command buttons and an output label.

Table 1: Controls and properties for frmStudent

Control	Name	Key properties set
Label × 4	lblName, lblRoll, lblClass, lblMarks	Caption = Name: / Roll No: / Class / Course: / Marks:
TextBox × 4	txtName, txtRoll, txtClass, txtMarks	Text = (blank); TabIndex = 0–3 in order; txtMarks.MaxLength = 3
CommandButton	cmdDisplay	Caption = Show Information; Default = True
CommandButton	cmdClear	Caption = Clear
Label (output)	lblOutput	Caption = (blank); BorderStyle =

		1 – Fixed Single; WordWrap = True

Three design decisions deserve explanation. First, meaningful names: txtName can be understood at a glance in code months later, whereas Text1 cannot; the three-letter prefixes (txt, lbl, cmd, frm) are the standard Visual Basic naming convention and should be used from the very first program. Second, each input has its own text box rather than one large box, because separate boxes let the program validate, store and print each field separately — for instance, Marks can be checked as a number while Name is checked as text. Third, the output label is given a border and WordWrap, so the displayed record appears as a neat panel and long entries wrap instead of being cut off.

AIOU LIBRARY aioulibrary.com

Part (b): Code to Display the Entered Data

Double-click cmdDisplay and write its Click procedure as below. The logic has three stages: read, check, display. Reading means copying the Text property of each box into the output; the ampersand operator (&) joins (concatenates) separate pieces of text into one string, and vbCrLf is the built-in constant for a carriage return plus line feed, which starts a new line inside the label. Checking means refusing to display an incomplete record: if the name box is empty, the procedure warns the user, places the cursor in that box with SetFocus, and leaves the procedure with Exit Sub. Displaying then assembles the four fields, each on its own labelled line, into lblOutput, and also echoes the record in a MsgBox to show a second display method.

Code 1: frmStudent — display and clear procedures

```
Private Sub cmdDisplay_Click()
   If Trim(txtName.Text) = "" Then
      MsgBox "Please enter the student name.", vbExclamation, "Missing Data"
      txtName.SetFocus
      Exit Sub
   End If
   lblOutput.Caption = "Name: " & txtName.Text & vbCrLf & _
      "Roll No: " & txtRoll.Text & vbCrLf & _
      "Class / Course: " & txtClass.Text & vbCrLf & _
      "Marks: " & txtMarks.Text
End Sub

Private Sub cmdClear_Click()
   txtName.Text = ""
   txtRoll.Text = ""
   txtClass.Text = ""
   txtMarks.Text = ""
   lblOutput.Caption = ""
   txtName.SetFocus
End Sub
```

A worked example makes the behaviour concrete. Suppose the user types Ahmed Raza in txtName, 2026-CS-014 in txtRoll, BS Computer Science in txtClass and 87 in txtMarks, and clicks Show Information. The procedure reads the four Text properties, joins them with labels and line breaks, and the output panel shows the record in four lines. Note what the program does with

the values: at this stage they are all text, because the Text property is always a string, even when the user types digits. That is why marks entered here cannot be added or averaged until they are converted with Val() or checked with IsNumeric() — a point developed fully in Questions 3 and 5 of this assignment. Note also the underscore (_) line-continuation character in the code, which lets one long logical statement be written over several physical lines for readability.

Sample Run / Output

```
Name: Ahmed Raza
Roll No: 2026-CS-014
Class / Course: BS Computer Science
Marks: 87
```

AIOU LIBRARY aioulibrary.com

The design can be extended in several instructive directions, and a good assignment answer should mention at least two. A ListBox or a data grid could collect many students one after another — each click of an Add button appending the record as a new line — turning a single-record display into a class list. The record could be printed with the Printer object or shown on a second form, which is exactly the multi-form technique examined in Question 4. Combo boxes could replace free typing for Class/Course, offering a fixed list of programmes and thereby preventing spelling variations. Finally, the same four fields become the columns of the Students table in the database application of Assignment 2, Question 5, where the display code grows into full record saving through ADO.

Conclusion

In conclusion, the student information form demonstrates the complete input–process–output cycle of a Visual Basic application in its simplest form: labels instruct, text boxes collect, the command button triggers, and code in the Click event reads the Text properties, checks them, and presents the record neatly in an output label. Mastery of these three controls, of the naming convention, and of string joining with & and vbCrLf is the prerequisite for everything that follows in this course — arithmetic programs, multi-form systems, validated data entry and database front-ends all reuse exactly this pattern with more controls and stricter checks.

Question 3: Arithmetic Operations on Two Numbers and Methods of Displaying Results (20 Marks)

Question as printed: a) Develop a program that takes two numbers as input and performs basic arithmetic operations. b) Display results using appropriate output methods. (20)

Introduction

A two-number calculator is the standard first computational program in Visual Basic, and it is more subtle than it looks. The user types characters into text boxes; the program must convert those characters into numbers, perform the arithmetic with the correct operators, protect itself against impossible operations such as division by zero, and present the answers clearly. Each of those four stages contains a concept the examiner is really testing: the Text property is a string, so the Val() function (or CDbl) is needed for conversion; Visual Basic has distinct operators for ordinary division (/), integer division (\), and remainder (Mod), which beginners constantly

confuse; a single unguarded division can crash the program with run-time error 11; and output can be sent to several destinations — a label, a message box, a list box, or the form itself with Print — each suited to a different purpose. This answer therefore builds one complete program that performs addition, subtraction, multiplication, division, integer division, remainder and power, and then compares the output methods and recommends the appropriate one for each situation, as part (b) requires.

Part (a): The Arithmetic Program

Design a form frmCalc with Caption 'Arithmetic Operations'. Place two labels ('First Number:' and 'Second Number:') and two text boxes, txtNum1 and txtNum2, for the input. Place one CommandButton, cmdCalculate ('Calculate'), and one output label, lblResult, with a border, stretched wide and tall enough for seven lines of results; add cmdClear to reset the form. In the Click procedure, declare two variables As Double — Double is chosen rather than Integer so that division can produce fractional answers and large values do not overflow. Read the inputs with Val(), which converts the leading numeric part of a string to a number and returns 0 for empty or non-numeric text, making the program forgiving of careless typing. Division needs a guard: before dividing, test whether the second number is zero, and if it is, report that division is not possible instead of letting the program crash.

Code 1: frmCalc — all operations in one procedure

```
Private Sub cmdCalculate_Click()
    Dim a As Double, b As Double
    a = Val(txtNum1.Text)
    b = Val(txtNum2.Text)
```

```
    lblResult.Caption = "Sum = " & (a + b) & vbCrLf & _
       "Difference = " & (a - b) & vbCrLf & _
       "Product = " & (a * b) & vbCrLf
    If b <> 0 Then
       lblResult.Caption = lblResult.Caption & _
          "Quotient = " & (a / b) & vbCrLf & _
          "Integer Division = " & (a \ b) & vbCrLf & _
          "Remainder = " & (a Mod b) & vbCrLf & _
          "Power = " & (a ^ b)
    Else
       lblResult.Caption = lblResult.Caption & _
          "Division not possible: second number is zero."
    End If
End Sub
```

AIOU LIBRARY aioulibrary.com

Table 1: Arithmetic operators in Visual Basic

Operator	Meaning	Example (a = 17, b = 5)	Result
+	Addition	a + b	22
− (hyphen -)	Subtraction	a − b	12
*	Multiplication	a * b	85
/	Ordinary (floating-point) division	a / b	3.4
\	Integer division — quotient only, fraction	a \ b	3

	discarded		
Mod	Remainder after integer division	a Mod b	2
^	Power (exponentiation)	2 ^ 3 in general a ^ b here	17^5 = 1419857

The worked example in the table repays study. With a = 17 and b = 5, ordinary division gives 3.4, integer division gives 3, and Mod gives 2, and the three answers are consistent: 17 = (5 × 3) + 2. Integer division and Mod are not classroom curiosities — Mod is how programs test even and odd numbers (n Mod 2), split a total into dozens and singles, and extract digits, while integer division is used in page counts, timetables and currency splitting. Equally important is operator precedence: multiplication and division are evaluated before addition and subtraction, and ^ before both, so 2 + 3 * 4 gives 14, not 20; parentheses must be used whenever the programmer intends a different order, as in (a + b) / 2 for an average. Variables should also be declared with Dim and typed properly — Option Explicit at the top of the code window forces declaration and catches misspelt variable names at compile time instead of silently creating new empty variables.

Sample Run / Output

```
Inputs: First Number = 17, Second Number = 5
Sum = 22
Difference = 12
Product = 85
Quotient = 3.4
```

```
Integer Division = 3
Remainder = 2
Power = 1419857
```

Part (b): Appropriate Output Methods

Visual Basic offers several output methods, and 'appropriate' in the question means choosing according to purpose. The Label is the best default for results that must stay visible: the answer remains on the form, can be formatted with Font and ForeColor, and updates simply by assigning to Caption, as in the program above. The MsgBox suits a single important result or warning that the user must acknowledge — for example the division-by-zero message — but it is modal and annoying if used for every line of output. The ListBox (with its AddItem method) is the right choice when many results accumulate, such as a history of calculations, because it scrolls and keeps earlier results instead of overwriting them. The Print method writes directly onto a form or PictureBox (for example Print "Sum = "; a + b) and is useful for quick, free-format output and for teaching, but the text is not a control: it disappears when the form is repainted unless AutoRedraw is set, and it cannot be selected or copied. Finally, results can be placed back into a TextBox, which has the unique advantage that the user can edit or copy the answer; setting its Locked or read-only style properties appropriately keeps that flexibility safe.

Table 2: Output methods compared

Method	How it is used	Stays visible?	Best used for
Label (Caption)	lblResult.Caption = ...	Yes	Main results panel of a form

MsgBox	MsgBox "Sum = " & s, vbInformation	Until OK is pressed	Single important result; warnings
ListBox (AddItem)	lstResults.AddItem ...	Yes, scrollable history	Many successive results
Print method	Print / PictureBox.Print	Only with AutoRedraw	Free-format quick output, exercises
TextBox (Text)	txtResult.Text = ...	Yes, editable/copyable	Results the user may copy or reuse

AIOU LIBRARY aioulibrary.com

Two refinements complete a polished answer. First, formatting: raw division results can show many decimal places, so the Format function should be used, for example Format(a / b, "0.00") to display exactly two decimal places — essential for marks, money and measurements. Second, input discipline: Val() quietly turns 'abc' into 0, which hides typing mistakes; a stricter program first tests IsNumeric(txtNum1.Text) and warns the user instead of calculating with a silent zero. Both refinements connect this question to the validation theme of Question 5, and both are cheap to add once the basic pattern is understood.

Conclusion

In conclusion, the arithmetic program unites the four essentials of computational Visual Basic: typed variables, conversion of text input into numbers with Val(), the full operator set — including the frequently confused /, \ and Mod, whose results for 17 and 5 are 3.4, 3 and 2 respectively — and a guarded division that reports zero denominators instead of crashing. For output, the label is the appropriate main method because results remain visible and formatted

on the form, supported by MsgBox for warnings, a list box for accumulated results, and Format() for tidy decimal display. With these habits in place, the student can extend the same form into a scientific calculator, a marks-average program, or the calculation module of Assignment 2.

Question 4: Multi-Form Application — Navigation, Event Handling and State Management (20 Marks)

Question as printed: c) Design and implement a multi-form Visual Basic application that demonstrates navigation between forms using appropriate controls. d) Apply proper event-handling and state management to ensure seamless data transfer between forms. (20) [Note: the two parts of Question 4 are lettered (c) and (d) in the printed booklet; they are answered here as the two parts of this question.]

AIOU LIBRARY aioulibrary.com

Introduction

Real applications are almost never a single screen. A student system has a main menu, a data-entry screen, a search screen and a report screen; a shop system separates billing from stock. Visual Basic supports this naturally because a project may contain many forms, and the programmer's task divides into three problems, which are exactly the two parts of this question: navigation — how the user moves from one form to another using buttons and menus; event handling — making each form respond correctly when it is loaded, shown, hidden and closed; and state management — how data typed on one form reaches another form intact, so the user never retypes it and the program never shows stale values. This answer

designs a small but complete two-form student application — a main form that collects the student's name and a detail form that greets the student and shows a summary — with a shared standard module holding the common data. The same pattern scales directly to five or ten forms: add the form, add a navigation button or menu item, and pass its data through the module or through public properties.

Part (c): Design and Navigation Between Forms

Create the project MultiFormDemo with three components, added through the Project menu: frmMain (the start-up form, set under Project → Project Properties → Startup Object), frmDetails (Project → Add Form), and a standard module modGlobal (Project → Add Module). On frmMain place a label, a text box txtStudentName, and three command buttons: cmdGo ('Go to Details'), cmdDetailsModal ('Details as Dialogue') and cmdExit. On frmDetails place a large label lblGreeting, a label lblSummary, and buttons cmdBack ('Back to Main') and cmdClose. Navigation in Visual Basic is performed by four statements that every student must distinguish precisely. Show displays a form, loading it first if necessary; Hide makes a form invisible but keeps it, and all its data, alive in memory; Unload removes the form from memory completely, destroying the values in its controls; and the modal form of Show — FormName.Show vbModal — presents the form as a dialogue that must be closed before the user can touch any other form, which is ideal for confirmations and settings screens.

Table 1: Form navigation statements compared

Statement	Effect on form	Data in controls	Typical use
frmDetails.Show	Loads (if needed) and	Preserved while form	Open a working screen

	displays, modeless	exists	
frmDetails.Show vbModal	Displays as a dialogue; other forms locked	Preserved	Confirmations, settings, login
Me.Hide / frmMain.Hide	Invisible but still loaded	Fully preserved	Wizard-style step screens
Unload Me	Removed from memory; Unload event fires	Destroyed	Closing a screen for good
frmDetails.SetFocus / ZOrder	Brings an open form to the front	Preserved	Switching among open windows

AIOU LIBRARY aioulibrary.com

Code 1: Navigation and data transfer (frmMain, modGlobal, frmDetails)

```
'===== modGlobal (standard module) =====
Public gStudentName As String
Public gProgramme As String


'===== frmMain =====
Private Sub cmdGo_Click()
   gStudentName = Trim(txtStudentName.Text)
   If gStudentName = "" Then
      MsgBox "Please type a student name first.", vbExclamation
      txtStudentName.SetFocus
      Exit Sub
   End If
```

```
    frmDetails.Show
    Me.Hide
End Sub

Private Sub cmdExit_Click()
    Unload frmDetails
    Unload Me
    End
End Sub

'===== frmDetails =====
Private Sub Form_Activate()
    lblGreeting.Caption = "Welcome, " & gStudentName & "!"
    lblSummary.Caption = "Programme: " & gProgramme
End Sub

Private Sub cmdBack_Click()
    frmMain.Show
    Me.Hide
End Sub

Private Sub cmdClose_Click()
    Unload Me
    frmMain.Show
End Sub
```

AIOU LIBRARY aioulibrary.com

Notice the pairing discipline in the navigation code: whenever a form is hidden to show another, the return button performs the mirror-image operation, so the user is never stranded on a screen with no way back, and only one main screen is visible at a time — the standard behaviour of wizard and menu-driven systems. An alternative navigation control is the menu: frmMain can carry a Navigate menu (Menu Editor) whose items Show the same forms, which gives keyboard shortcuts and a more professional appearance; buttons and menus can coexist and call the same procedures. Larger applications often use an MDI (Multiple Document Interface) parent form whose child forms open inside it, but for course-level systems the Show/Hide pattern above is clearer and is what this question expects.

AIOU LIBRARY aioulibrary.com

Part (d): Event Handling and State Management

Event handling in a multi-form program means choosing the right form event for each job. Load fires once when a form is loaded into memory and is the place for one-time initialisation — filling combo boxes, setting default values, opening connections. Activate fires every time a form becomes the active window, including each time the user returns to it, and is therefore the correct place to refresh displayed data — which is why frmDetails reads gStudentName in Form_Activate, not Form_Load: if the user goes back, changes the name, and comes forward again, Load would not fire a second time (the form was only hidden, not unloaded) and the greeting would show the old name, while Activate fires on every arrival and always shows the current value. Unload and QueryUnload handle departure: QueryUnload can ask 'Are you sure?' and cancel the closing by setting Cancel = 1 if the user has unsaved work. Getting these choices

right is the difference between an application that merely opens windows and one that behaves seamlessly.

State management is the discipline of deciding where shared data lives. Visual Basic offers three main techniques, in increasing order of formality. First, public variables in a standard module — gStudentName and gProgramme above — are visible to every form, survive hiding and showing, and are the simplest correct place for session-wide data such as the logged-in user or the current student; their weakness is that any code anywhere can change them, so they should be few and well named. Second, one form can read another form's controls directly while that form is loaded — for example frmMain.txtStudentName.Text — which avoids duplication but couples the forms tightly and breaks with an error if the owner form has been unloaded. Third, a form can expose its own Public properties and variables, set by the caller before Show, which is the cleanest object-oriented style. Whichever technique is chosen, three rules keep state seamless: store each fact in exactly one place (the module), refresh every display in Activate, and decide deliberately between Hide (state kept, memory used) and Unload (state discarded, memory freed) for each screen.

AIOU LIBRARY aioulibrary.com

Sample Run / Output

```
[frmMain]  Student Name: Sara Khan   [Go to Details]
[frmDetails opens]  Welcome, Sara Khan!
                    Programme: BS Computer Science
[Back to Main] -> name changed to Sara Khan (Revised) -> [Go to Details]
[frmDetails]  Welcome, Sara Khan (Revised)!   <- refreshed in Form_Activate
```

Conclusion

In conclusion, a multi-form Visual Basic application rests on a small vocabulary used precisely: Show and Show vbModal to open forms modelessly or as dialogues, Hide to park a form with its data intact, and Unload to destroy it. Navigation buttons (or menu items) move the user in mirrored pairs so no screen is a dead end. Seamlessness then comes from event handling and state management rather than from more code: initialisation in Load, refreshing in Activate, guarded closing in QueryUnload, and shared facts kept once in a standard module's public variables and read wherever they are displayed. The login → main → details flow of almost every real VB system, including the database application in Assignment 2, is this exact pattern with more forms added.

Question 5: Robust Data Entry — Validation Rules and Exception Handling (20 Marks)

Question as printed: a) Develop a robust data entry system with validation rules (e.g., required fields, data type checks, range validation). b) Implement exception handling mechanisms to manage runtime errors effectively. (20)

Introduction

A data-entry form that accepts anything is worse than no form at all, because bad data does not announce itself at entry time — it surfaces weeks later as a wrong merit list, an unprintable certificate, or a database full of marks recorded as 870 instead of 87. Robust data entry therefore rests on two defensive layers, which are the two parts of this question. The first layer

is validation: checking every field against explicit rules at the moment of entry — required fields must not be empty, numeric fields must contain numbers, values must fall inside sensible ranges, and text must respect length and format limits — and refusing to proceed, with a clear message and the cursor placed in the offending box, until the data is clean. The second layer is exception handling: Visual Basic's structured response to the run-time errors that validation cannot prevent, such as a file that does not exist or a database that is unavailable, using On Error GoTo, the Err object and Resume, so that the program reports a problem gracefully instead of crashing and losing the user's work. This answer builds a student marks entry system that demonstrates both layers in working code.

Part (a): Validation Rules

Validation is best designed as a table of rules before any code is written, because the rules come from the meaning of the data, not from the programming language. For a student entry form with Name, Roll Number, Marks and Age, the rules are as follows. Name is required, must contain at least 3 characters, and should not contain digits. Roll Number is required and must follow the institutional pattern in length. Marks are required, must be numeric (data type check), and must lie in the range 0 to 100 (range validation). Age, if collected, must be numeric and within 15 to 60. Each rule should be checked at two moments for the best user experience: immediately, in the field's LostFocus or Validate event, so the user learns of a problem while the field is still fresh in mind; and finally, in the Save button, where all fields are re-checked in order, because a user can sometimes reach Save without ever visiting a field. The Validate event is particularly elegant in VB6: setting its Cancel argument to True keeps the focus in the invalid control automatically.

Table 1: Validation rules for the student entry form

Field	Required?	Type / format check	Range / length rule	Message on failure
Name (txtName)	Yes	Text; letters and spaces	3–40 characters	"Name is required (3–40 characters)."
Roll No (txtRoll)	Yes	Alphanumeric pattern	5–15 characters	"Enter a valid roll number."
Marks (txtMarks)	Yes	IsNumeric must be True	0 to 100 inclusive	"Marks must be a number between 0 and 100."
Age (txtAge)	No	IsNumeric if filled	15 to 60	"Age must be between 15 and 60."

AIOU LIBRARY aioulibrary.com

Code 1: Validation function and its use in the Save button

```
Private Function ValidateForm() As Boolean
   ValidateForm = False
   If Trim(txtName.Text) = "" Or Len(Trim(txtName.Text)) < 3 Then
      MsgBox "Name is required (at least 3 characters).", vbExclamation
      txtName.SetFocus
      Exit Function
   End If
   If Not IsNumeric(txtMarks.Text) Then
```

```
        MsgBox "Marks must be a number.", vbExclamation
        txtMarks.SetFocus
        Exit Function
    End If
    If Val(txtMarks.Text) < 0 Or Val(txtMarks.Text) > 100 Then
        MsgBox "Marks must be between 0 and 100.", vbExclamation
        txtMarks.SetFocus
        Exit Function
    End If
    ValidateForm = True
End Function

Private Sub txtMarks_LostFocus()
    If txtMarks.Text <> "" And Not IsNumeric(txtMarks.Text) Then
        MsgBox "Please enter numbers only in Marks.", vbExclamation
        txtMarks.SetFocus
    End If
End Sub
```

Several details of this code deserve attention. Trim() removes spaces, so a box containing only spaces — which looks filled to a careless check — is correctly treated as empty. IsNumeric is the proper data type check because Val() alone would silently convert '12abc' to 12 and 'abc' to 0, hiding the mistake. Returning a Boolean from a single ValidateForm function keeps all rules in one place, so the Save button's logic is simply: if validation fails, stop; if it succeeds, save — and when a rule changes (the institution raises total marks to 110, for example) only one function is

edited. Preventive design reduces the validation burden further: setting txtMarks.MaxLength, using a masked or drop-down control where the choices are fixed, and converting case consistently all stop errors before they need to be detected.

Part (b): Exception Handling for Run-Time Errors

Validation cannot prevent every failure. The file holding the data may be missing, the disk may be full, the database server may be off, or the user may cancel a common dialogue — none of these is the user's typing mistake, and all of them raise run-time errors that, unhandled, terminate the program abruptly with Visual Basic's own unfriendly message and the loss of everything typed. Visual Basic's classic mechanism is the On Error statement. On Error GoTo Handler switches on error trapping and directs execution to a labelled block when an error occurs; inside the handler, the Err object supplies Number and Description — for example error 11 'Division by zero', 53 'File not found', 71 'Disk not ready' and 94 'Invalid use of Null'. The handler should inform the user in plain language, log or preserve the entered data where possible, and then either Resume (retry the failed statement, after the cause is fixed), Resume Next (skip the statement), or exit the procedure cleanly. On Error Resume Next has a legitimate narrow use — testing whether an optional operation succeeded — but used carelessly it hides real bugs and should never blanket a whole procedure.

Code 2: Save procedure with exception handling

```
Private Sub cmdSave_Click()
    On Error GoTo SaveError
    If Not ValidateForm() Then Exit Sub
    Open App.Path & "\students.txt" For Append As #1
```

```
    Print #1, txtName.Text & "," & txtRoll.Text & "," & txtMarks.Text
    Close #1
    MsgBox "Record saved successfully.", vbInformation
    Exit Sub
SaveError:
    MsgBox "The record could not be saved." & vbCrLf & _
       "Error " & Err.Number & ": " & Err.Description, vbCritical, "Save
       Failed"
    If Err.Number = 71 Or Err.Number = 61 Then
       MsgBox "Check the disk/drive and press Save again; your typed data is
       still on the form.", vbInformation
    End If
End Sub
```

AIOU LIBRARY aioulibrary.com

Table 2: Common run-time errors and the correct response

Error	Meaning	Typical cause in data entry	Handler response
11	Division by zero	Average computed with zero records	Guard before dividing; report 'no records yet'
53	File not found	Data file moved or deleted	Create the file or ask for its location
71 / 61	Disk not ready / full	Saving to a removed USB or full disk	Keep form data; ask user to fix and retry (Resume)

94	Invalid use of Null	Reading an empty database field	Test with IsNull; substitute a default
380	Invalid property value	Assigning an out-of-range value to a control	Validate before assigning

Good exception handling follows four principles which examiners look for. First, trap errors only where they can genuinely occur — around file, database and conversion operations — rather than switching trapping on everywhere, which makes programs hard to reason about. Second, never show the user a bare error number: translate it into what happened and what to do next, while keeping the number and description available for the technician, exactly as Code 2 does. Third, preserve the user's work: note that in Code 2 the form is never cleared on failure, so re-saving after fixing the disk loses nothing. Fourth, distinguish the three error families — compile-time errors (found before running), run-time errors (handled here), and logic errors (wrong answers with no crash, prevented by testing and by the validation layer) — because each family has a different remedy, and confusing them produces programs that are trapped everywhere yet still wrong.

Sample Run / Output

```
Case 1: Marks box empty -> "Marks must be a number." (focus returns to Marks)
Case 2: Marks = 140 -> "Marks must be between 0 and 100."
Case 3: Marks = 87, disk removed -> "The record could not be saved. Error 71:
Disk not ready" + retry advice; typed data retained
Case 4: All valid -> "Record saved successfully."
```

Conclusion

In conclusion, robustness is built in two layers that must not be confused. Validation enforces the rules of the data itself — required fields filled, types correct, values in range — checked field by field in LostFocus/Validate and again as a whole in a single ValidateForm function behind the Save button, always with a clear message and the focus returned to the offending box. Exception handling then deals with the failures of the environment — missing files, unavailable disks and databases, Null values — through On Error GoTo, the Err object, and a considered choice among Resume, Resume Next and a clean exit, while preserving the user's typed work. A data entry system with both layers neither accepts corrupt data nor collapses at the first surprise, which is precisely the standard expected of the database applications developed in Assignment 2.

AIOU LIBRARY aioulibrary.com