Chapter 3

Working with Forms

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 form is the top-level object in a Visual Basic application, and every application starts with the form. Forms have 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 this with the mouse, or with the keyboard through the Control menu.
The forms that constitute the visible interface of your application are called ***Windows forms****;* this term includes both the regular forms and dialog boxes, which are simple forms you use for very specific actions, such as to prompt the user for a specific 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.

3.1 The anatomy of forms

Applications are made up of one or more forms, and the forms are what users see. You should create your forms carefully, make them functional, and keep them simple and spontaneous. The main characteristic of a form is the title bar on which the form's caption is displayed (see Figure).

Clicking the icon on the left end of the title bar opens the Control menu, which contains the commands shown in Table 3.1. On the right end of the title bar are three buttons: Minimize, Maximize, and Close. Clicking these buttons performs the associated function. When a form is maximized, the Maximize button is replaced by the Restore button. When clicked, this button resets the form to the size and position before it was maximized. The Restore button is then replaced by the Maximize button.

Table 3.1: Commands of the Control Menu

Command	Effect
Restore	Restores a maximized form to the size it was before it was maximized; available only if the form has been maximized
Move	Lets the user move the form around with the mouse
Size	Lets the user resize the form with the mouse
Minimize	Minimizes the form
Maximize	Maximizes the form
Close	Closes the current form

3.2 Forms Properties

You're familiar with the appearance of the forms, even if you haven't programmed in the Windows environment in the past; you have seen nearly all types of windows in the applications you're using every day. The dialog boxes that display critical information or prompt you to select the file to be opened are also forms. You can duplicate the look of any window or dialog box through the following properties of

the Form object. Like all controls, the form has many (over 50) properties. Some important properties of form are as follows.

Name:

Name used to identify form. It is the name by which the form is referred to in your code. Visual Basic by default names the forms Form1, Form2, Form3, etc. three letter prefix for form names is ***frm***

Appearance:

Return or set whether or not an object is painted at run time with 3D effects. There are two values of this property 0-Flat & 1- 3D. the default vale is 3D.

BackColor:

This property is used to sets the form background color.

BorderStyle

The BorderStyle property determines the style of the form's border and the appearance of the form; it takes one of the values shown in Table 3.2. You can make the form's title bar disappear altogether by setting the form's BorderStyle property to FixedToolWindow, the ControlBox property to False, and the Text property to an empty string. However, a form like this can't be moved around with the mouse and will probably disturb users.

Table 3.2: The FormBorderStyle Enumeration

Value	**Effect**
None	Borderless window that can't be resized; this setting should be avoided.
Sizable (default)	Resizable window that's used for displaying regular forms.
FixedDialog	A fixed window, used to create dialog boxes.
FixedSingle	A fixed window with a single line border.
FixedToolWindow	A fixed window with a Close button only. It looks like the toolbar displayed by the drawing and imaging applications.
SizableToolWindow	Same as the FixedToolWindow but resizable. In addition, its caption font is smaller than the usual.
Caption:	It is the text that appears in the title bar of form. A caption can be changed at runtime. The caption must be informative and meaningful.
ControlBox:	This property is also True by default. Set it to False to hide the 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.
Enabled:	If True, allows the form to respond to mouse and keyboard events; if False, disables form. Default value is true.
Font:	This property is used to sets font type, style, size.
ForeColor:	This property is used to sets color of text or graphics.
Height:	This property is used to sets the height of the form.
Left:	This property is used to sets the distance from left side of computer screen to left side of form.
Min Button, Max Button:	These two properties are True by default. Set them to False to hide the corresponding buttons on the title bar.
Movable:	The default value is true. If set to false, the form cannot be moved by the user during the runtime.

Picture: This property is used to places a bitmap picture in the form.

StartUpPosition: This property determines the initial position of the form when it's first displayed; it can have one of the values shown in Table 3.3.

Table 3.3: The FormStartPosition Enumeration

Value	Effect
Manual	The location and size of the form will determine its starting position.
CenterOwner	The form is centered in the area of its parent form.
CenterScreen	The form is centered on the monitor.
WindowsDefaultBounds	The form is positioned at the default location and size determined by Windows.

Top: This property is used to sets the distance from top side of computer screen to top side of form.

Visible: This property is used to sets a value that determines whether an form is visible or hidden. Default value is true.

Width: This property is used to sets the width of the form.

WindowState: This property set the visual state of a form window at run time. Following table shows the different values of this property.

Table 3.4:

Value	Effect
Normal	Display form in normal mode.
Minimized	Display form in minimized mode.
Maximized	Display form in maximized mode.

3.3 Form Events

The form primarily acts as a container for other controls, but it does support events. That is it can respond to some user interactions. Some commonly used events are as follows:

Form Events:

Event	Description
Click	Event executed when user clicks on the form with the mouse.
DblClick	Dblclick event triggered when user double clicks on form.
Activate	Activate event is triggered when form becomes the active window.
Deactivate	Deactivate event is triggered when form becomes the deactivate window.
Load	Load event occurs when form is loaded. This is a good place to initialize variables and set any run time properties.
Unload	Unload event occurs when user close the form.
Terminate	This event is used to close or terminate the form. All the memory that was held for the form variables are released
Resize	This event is occurred when the user resizes form.
MouseMove	This event is occurred when the user mouse moves on the form.
Keypress	Tis event is occurred when the user presses key from the keyboard.

Besides these events there are some events these are initialize, mouseup, mousedown, keyup, keydown etc.

3.4 Form Methods

Let us now take look at the methods that are associated with the form. Forms have methods that are used in code to control their behavior. There are various methods which are as follows:

Move:
The move method allows the programmer to position the form at a desired location on the screen. The syntax of move method is as follows:
Formname.move [left],[top], [width], [hight]
For example:
Form1.move 300, 500, 3000, 5000
Show:
The show method displays the form to the user on the screen. It is like setting the visible property to "True". For example
Form1.show

Hide:
This method is used to hide the form which is displayed on the screen. This method works exactly apposite to show method. It will make the form invisible at runtime.
For Example:
Form1.hide
Refresh:
This method is used to referesh the form.
For example:
Form1.refresh
Cls:
Clears all graphics and text from the form. Does not clear any objects.
For Example:
Form1.cls
Print:
Prints text string on the form.
For example:
Form1.print "Dayanand Science College"

Form Drawing Methods:

Visual Basic provides several methods that support drawing graphical objects and text directly on the surface of the form. Several of these methods use a drawing pen that positioned at some X and Y position of the form. The origin of the form's axes (point 0,0) is at the upper left corner of the form. There are various drawing methods which are as follows:

Circle:

This method is used to draw a circle on the form.
Syntax:
FormName.circle [step], (X,Y), Radius, [Color]
The optional step parameter indicates the center of the circle relative to the form's currentX and currentY

properties.

Example:

```
Private Sub Command1_Click()
For i = 100 To 10000 Step 100
Me.Circle (i, i), i, RGB(255, 0, 0)
Next i
End Sub
```

Line:

The line method draws a line from a point defined by an X and Y coordinate to another point defined by a different X, Y coordinate.

Syntax:

Formname.Line (X1, Y1) – (X2, Y2), [color]

Example:

```
Private Sub Command2_Click()
Dim i As Integer
ScaleMode = 3
For i = 10 To 250 Step 3
Me.Line (10, i)-(250, i), RGB(0, 0, 255)
Me.Line (i, 10)-(i, 250), RGB(255, 0, 255)
Next i
End Sub
```

PSet:

The PSet method is easy to understand. Given X and Y position it draws a single point on the screen in a particular color.

Syntax:

FormName.PSet (X, Y), [color]

Example:

```
Private Sub Command3_Click()
Dim i, j, red, green, blue As Integer
ScaleMode = 3
For i = 1 To 200
    For j = 1 To 200
        red = CInt(Rnd * 255)
        green = CInt(Rnd * 255)
        blue = CInt(Rnd * 255)
        Me.PSet (i, j), RGB(red, green, blue)
    Next j
Next i
End Sub
```

Point:

The point method is the opposite of PSet. Rather than setting a point on the screen to a particular color, point returns the color of the screen at the location specified by its X and Y parameters.
Syntax:
LongInteger Variable = FormName.Point(X,Y)
Point returns a long integer value.

3.5 Working with MDI Form

The Multiple Document Interface (MDI) was designed to simplify the exchange of information among documents, all under the same roof. With an MDI application, you can maintain multiple open windows, but not multiple copies of the application. Data exchange is easier when you can view and compare many documents simultaneously.
We used Windows applications that can open multiple documents at the same time and allow the user to switch among them with a mouse-click. Microsoft Word is a typical example, although most people use it in single document mode. Each document is displayed in its own window, and all document windows have the same behavior. The main form, or MDI Form, is not duplicated, but it acts as a container for all other windows, and it's called the ***parent window***. The windows in which the individual documents are displayed are called ***child windows*** (or document windows). When you reposition the parent window on the Desktop, its child windows follow.
Child windows, however, exist independently of the parent window. You can open and close child windows as you want, and child windows can even have different functions.
MDI applications aren't very common; not too many applications lend themselves to MDI implementation. Most of them are easier to implement with multiple Forms, but some applications should be implemented with an MDI interface. These are the applications that can open multiple documents of the same type and use a common menu structure that applies to all open documents.

MDI Applications: The Basics:

An MDI application must have at least two Forms, the parent Form and one or more child Forms. Each of these Forms has certain properties. There can be many child Forms contained within the parent Form, but there can be only one parent Form.
The parent Form may not contain any controls. While the parent Form is open in design mode, the icons on the Toolbox aren't disabled, but you can't place any controls on the Form. The parent Form can, and usually does, have its own menu.
To create an MDI application, follow these steps:

1. Start a new project and then choose Project → Add MDI Form to add the parent Form.
2. Set the Form's caption to MDI Window.
3. Choose Project → Add Form to add a regular Form.
4. Make this Form the child Form by setting its ***MDIChild property to True***. To denote that this is a child Form, set its Caption property to MDI Child Form.

Visual Basic automatically associates this new Form with the parent Form. This child Form can't exist outside the parent Form; in other words, it can only be opened within the parent Form.

3.6 Designing Menus – Menu Editor

Menus are one of the most common and characteristic elements of the Windows user interface. Even in the old days of character-based displays, menus were used to display methodically organized choices and guide the user through an application. Despite the visually rich interfaces of Windows applications and the many alternatives, menus are still the most popular means of organizing a large number of options. Many applications duplicate some or all of their menus in the form of icons on a toolbar, but the menu is a standard fixture of a Form. You can turn the toolbars on and off, but not the menus.

The Menu Editor

Menus can be attached only to Forms, and you design them with the Menu Editor. To see how the Menu Editor works,

- start a new project, and when Form1 appears in the design window.
- Choose Tools ➢ Menu Editor to open the Menu Editor, as shown in following Figure.
- Alternatively, you can click the Menu Editor Button on the toolbar.

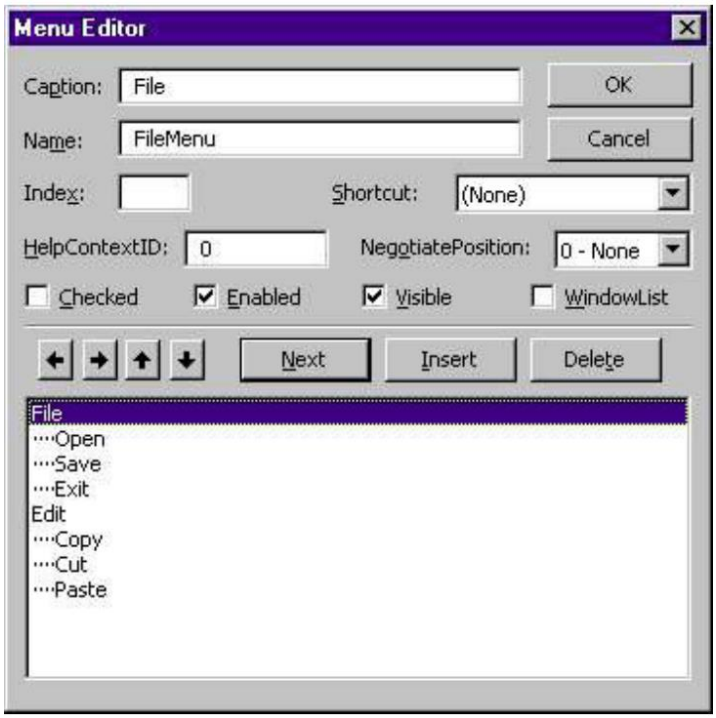


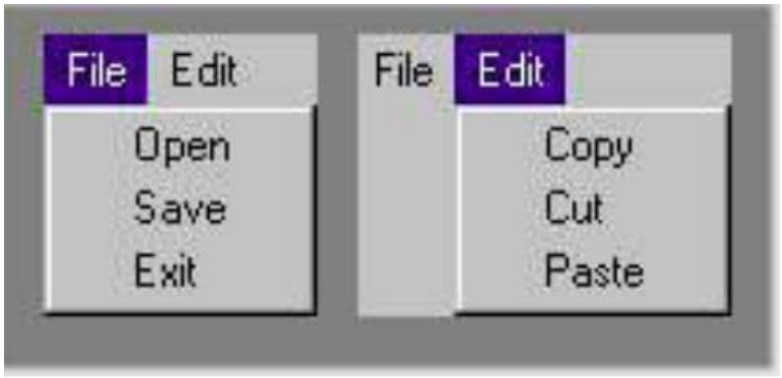
Figure 1: The Menu Editor's window displays a simple menu structure.

In the Menu Editor window you can specify the structure of your menu by adding one command at a time. Each menu command has two mandatory properties:

- **Caption** This is the string that appears on the application's menu bar.
- **Name** This is the name of the menu command. This property doesn't appear on the screen, but your code uses it to program the menu command.

The Caption and Name properties of a menu item are analogous to the properties that have the same name as the Command button or Label control. Caption is what the user sees on the Form, and Name is the means of accessing the control from within the code. As far as your code is concerned, each menu

command is a separate object, just like a Command button or Label control.
To add commands to the Form's menu bar, enter a caption and a name for each command. As soon as you start typing the command's caption, it also appears in a new line in the list at the bottom of the Menu Editor window. Let's create the menu structure shown in above figure. This menu contains two commands, File and Edit. When the user clicks on either one, the submenus are displayed.



Following Table shows the Caption and Name properties for each command.

TABLE: Caption and Name Properties for File and Edit Commands

CAPTION	**NAME**
File	FileMenu
Open	FileOpen
Save	FileSave
Exit	FileExit
Edit	EditMenu
Copy	EditCopy
Cut	EditCut
Paste	EditPaste

The commands that belong to each menu form the corresponding submenu and are indented from the left. To design the menu, follow these steps:

1. Open a new Form in the Design pane, and choose Tools ➢ Menu Editor to open the Menu Editor window.
2. In the Caption box, type the caption of the first command (**File**).
3. In the Name box, enter the command's name (**FileMenu**).
4. Press Enter or click Next to enter the next command. Repeat steps 1 through 3 for all the commands listed.

If you run the application now, all the commands you've entered will be displayed along the Form's menu bar. If the window isn't wide enough to fit the entire menu, some of the menu's commands will be

wrapped to a second line.
To create the menu hierarchy (make the commands appear under the File and Edit headings), you must indent them. Select the Open command in the list and click the button that has the right-pointing arrow. Do the same for the Save, Copy, Cut, and Paste commands. Your Menu Editor window should look like the one in Figure 1.
If you run the application now, the menu has the proper appearance. Each subordinate command appears under the first-level menu command to which it belongs. To view the subordinate commands, click the corresponding top-level command. For example, to select the Paste command, first open the Edit menu. You can nest menu commands to more than two levels by selecting a command and pressing the button with the right-pointing arrow more than once. When a menu command leads to a submenu, an arrow appears to its right, indicating that if selected, it will lead to a submenu.
The remaining fields on the Menu Editor window are optional and are described next.

The Index Field

Use the Index field to create an array of menu commands. All the commands of the array have the same name and a unique index that distinguishes them, just like arrays of controls.

The Checked Field

Some menu commands act as toggles, and they are usually checked to indicate that they are on or unchecked to indicate that they are off. To display a checkmark next to a menu command initially, select the command from the list by clicking its name, and then check the Checked box in the Menu Editor window. You can also access this property from within your code to change the checked status of a menu command at runtime.

The Enabled Field

Some menu commands aren't always available. The Paste command, for example, has no meaning if the Clipboard is empty. To indicate that a command can't be used at the time, you set its Enabled property to False. The command then appears grayed in the menu, and although it can be highlighted, it can't be activated. You can set the initial status of a command by checking or clearing the Enabled box in the Menu Editor window. You can also toggle the status of a menu command from within your code by manipulating its Enabled property, similar to the Checked command.

The Visible Field

To remove a command temporarily from the menu, set the command's Visible property to False. The Visible property isn't used frequently in menu design. In general, you should prefer to disable a command to indicate that it can't be used (some other action is required to enable it). Making a command invisible frustrates users, who may try to locate the command in another menu.

The Window Field

This option is used with MDI (Multiple Document Interface) applications to maintain a list of all open windows.

Adding Code to Menus:

Menu commands are similar to controls. They have certain properties that you can manipulate from within your code, and they recognize a single event, the Click event. If you select a menu command at

design time, Visual Basic opens the code for the Click event in the Code window. The name of the event handler for the Click event is composed of the command's name followed by an underscore character and the event's name, as with all other controls.
You can also manipulate the menu command's properties from within your code. These properties are the ones you can set at design time, through the Menu Editor window. Menu commands don't have methods you can call.
Most menu object properties are toggles. To change the Checked property of the FontBold command, for instance, use the following statement:
FontBold.Checked = Not FontBold.Checked
If the command is checked, the checkmark will be removed. If the command is unchecked, the checkmark will be inserted in front of its name.
You can also change the command's caption at runtime, although this practice isn't common. The Caption property is manipulated only when you create dynamic menus by adding and removing commands at runtime.

3.7 Building Dynamic Forms at Runtime

There are situations when you won't know in advance how many instances of a given control may be required on a form. Since every table consist of different fields, it will be difficult to build a single form to accommodate all the possible tables a user may throw at your application.

In these situation, it is possible to design dynamic forms, which are populated at runtime. The simplest approach is to create more controls than you'll ever need and set their visible property to false at design time. At run time, you can display the controls by switching their visible property to true. The proper method to create dynamic forms at run time is as follows.
Just as you can load (existing) Forms with the Load statement, you can also load new controls on a Form. The controls to be loaded don't need to exist on the Form; they can be members of an array of controls. At design time, you can create one control and make it the first member of an array. To do so, assign the value 0 to its Index property at design time. Figure shows the Cloud project, which demonstrates this technique.

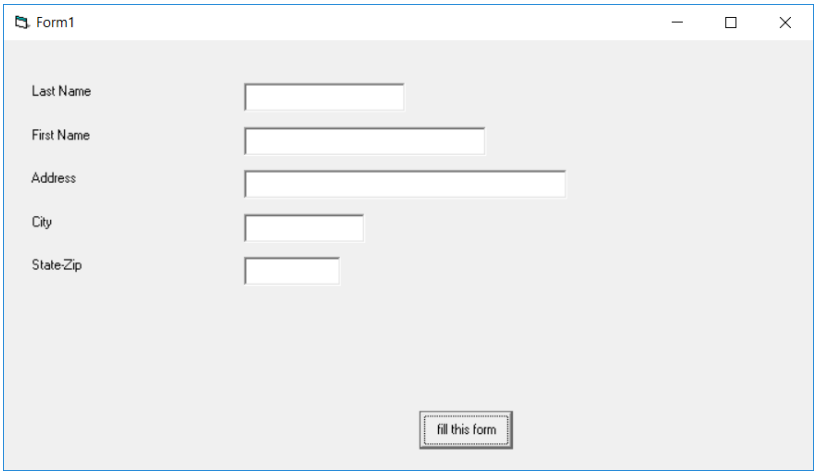
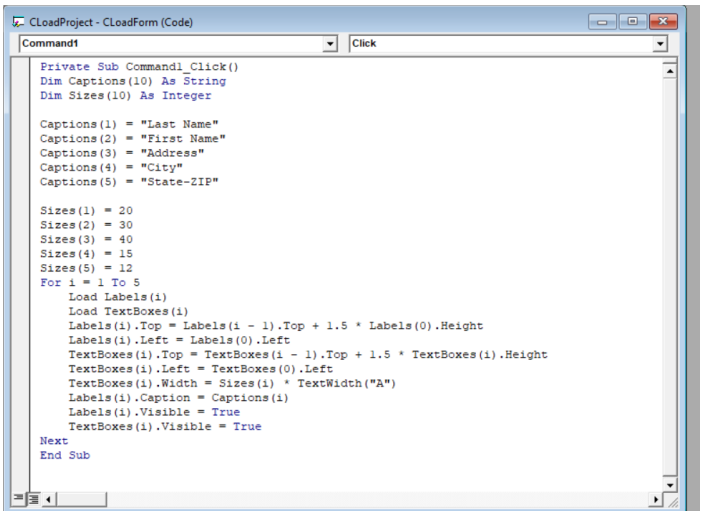


Fig. : The CLoad application creates this Form with code when it's loaded.

The Form of figure is a data entry Form and the fields are placed on the Form at runtime. The CLoad project builds the same Form every time it's executed, but in a practical situation, you would adjust the fields to reflect the structure of a table or a structure defined by the user on another Form.
To design the data entry Form of the CLoad application, you must place a Lable and TextBox control on the Form. Place them near the top of the Form, so that they can be used as guides for aligning the remaining controls. These two cantrols will remain invisible at all times. Then complete the following step:

1. Name the Label control Labels and set its Index property to 0.
2. Set its Alignment property to 1 (Right Justify).
3. Name the TextBoxes control and set its Index to 0.
4. Then select both controls with the mouse and set their Visible property to False.

So far, you've created the structure necessary for loading new controls at run-time. These controls will be placed on the Form in pairs with the Load command and they'll be aligned with the two controls placed on the Form at design time. The code behind the Fill This Form button is shown in Code 4.10



The arrays Captions and Sizes hold the captions of the various fields and their lengths in number of characters. These two arrays don't change in the CLoad application. However, you can either extract these values from a table in a database or let the user design the structure of the desired record and then use it to construct the arrays. It's a simple method for creating data entry Forms that can be reused in many situations. By the way, this is how many Wizards work.
The code then places another pair of controls on the Form, aligns them with the previous ones, adjusts the size of the TextBox control according to the value of the corresponding element of the sizes array, and sets the caption of the Label control to the value of the corresponding element of the Caption array. Finally, it displays them by setting their Visible property to True. Controls placed on a Form with the Load statement are invisible by default. Even if the first items in a control array were visible, the controls loaded with the Load statement would be invisible, so you must always turn on the Visible property from within your code.
This subroutine relies on the placement of the initial controls on the Form. There are more ways to position the controls on the Form, but this is probably the simplest method. Place a few controls on the

Form at design time, and then use them as guides for aligning the new controls at runtime.