Forms, Controls, and Menus

The first step to creating an application with Visual Basic is to create the interface, the visual part of the application with which the user will interact. Forms and controls are the basic building blocks used to create the interface; they are the objects that you will work with to build your application.

Forms are objects that expose properties which define their appearance, methods which define their behavior, and events which define their interaction with the user. By setting the properties of the form and writing Visual Basic code to respond to its events, you customize the object to meet the requirements of your application.

Controls are objects that are contained within form objects. Each type of control has its own set of properties, methods and events that make it suitable for a particular purpose. Some of the controls you can use in your applications are best suited for entering or displaying text. Other controls let you access other applications and process data as if the remote application was part of your code.

This chapter introduces the basic concepts of working with forms and controls and their associated properties, methods, and events. Many of the standard controls are discussed, as well as form-specific items such as menus and dialog boxes.

Topics

Additional Controls

An introduction to the other standard Visual Basic controls.

Understanding Focus

A brief discussion of focus as it applies to controls.

Setting the Tab Order

An introduction to the concepts of tab order within a form.

Menu Basics

An introduction to menu controls and the menu editor.

Prompting the User with Dialog Boxes

An introduction to dialog boxes.

Sample application

Controls.vbp

The code examples in this chapter are taken from the Controls.vbp sample application which is listed in the Samples directory.

Send feedback to MSDN. *Look here for MSDN Online resources.*

Understanding Properties, Methods and Events

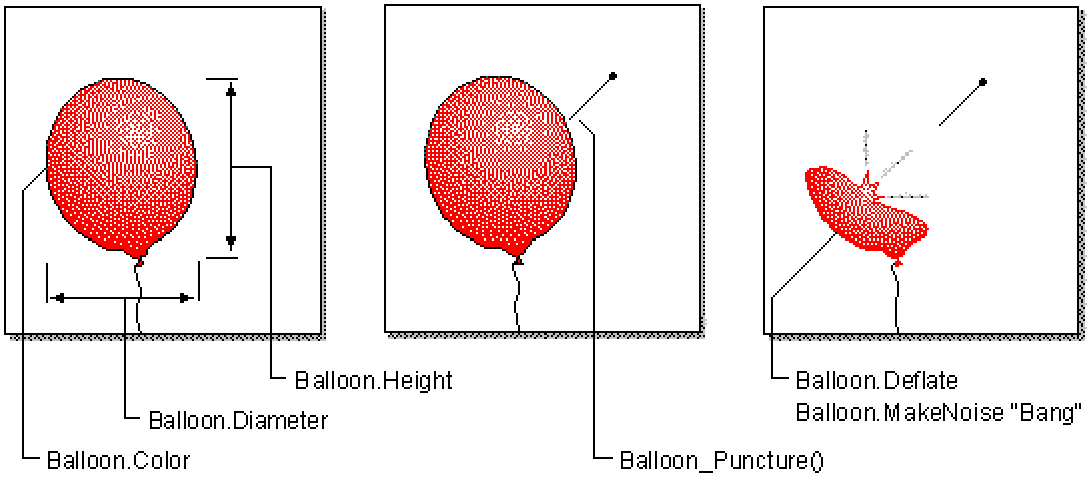
Visual Basic forms and controls are objects which expose their own properties, methods and events. Properties can be thought of as an object's attributes, methods as its actions, and events as its responses.

An everyday object like a child's helium balloon also has properties, methods and events. A balloon's properties include visible attributes such as its height, diameter and color. Other properties describe its state (inflated or not inflated), or attributes that aren't visible such as its age. By definition, all balloons have these properties; the settings of these properties may differ from one balloon to another.

A balloon also has inherent methods or actions that it might perform. It has an inflate method (the action of filling it with helium), a deflate method (expelling its contents) and a rise method (if you were to let go of it). Again, all balloons are capable of these methods.

Balloons also have predefined responses to certain external events. For instance, a balloon would respond to the event of being punctured by deflating itself, or to the event of being released by rising into the air.

Figure 3.1 Objects have properties, respond to events, and perform methods



If you were able to program a balloon, the Visual Basic code might look like the following. To set the balloon's properties:

```
Balloon.Color = Red
Balloon.Diameter = 10
Balloon.Inflated = True
```

Note the syntax of the code — the object (Balloon) followed by the property (.Color) followed by the assignment of the value (Red). You could change the color of the balloon from code by repeating this statement and substituting a different value. Properties can also be set in the Properties window while you are designing your application.

A balloon's methods are invoked like this:

```
Balloon.Inflate
Balloon.Deflate
Balloon.Rise 5
```

The syntax is similar to the property — the object (a noun) followed by the method (a verb). In the third example, there is an additional item, called an *argument*, which denotes the distance to rise. Some methods will have one or more arguments to further describe the action to be performed.

The balloon might respond to an event as follows:

```
Sub Balloon_Puncture()
   Balloon.Deflate
   Balloon.MakeNoise "Bang"
   Balloon.Inflated = False
   Balloon.Diameter = 1
End Sub
```

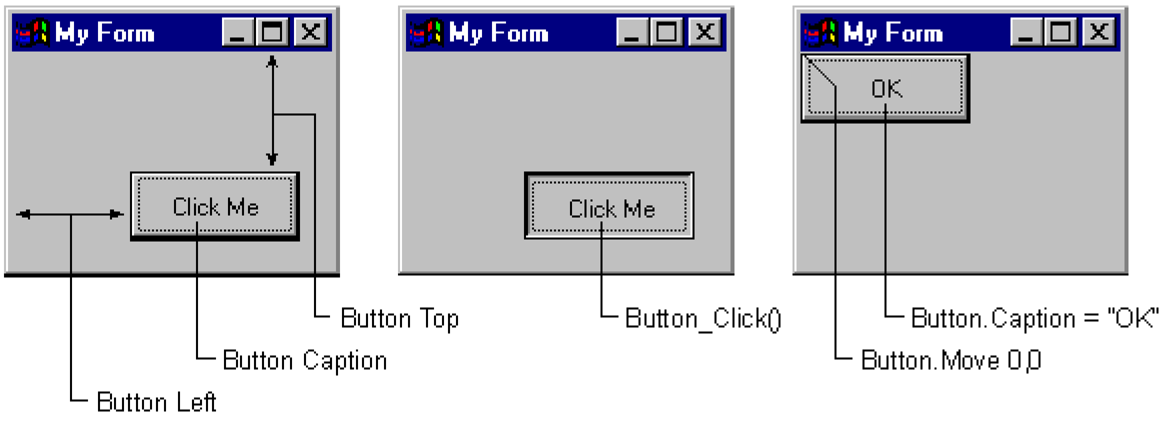
In this case, the code describes the balloon's behavior when a puncture event occurs: invoke the Deflate method, then invoke the MakeNoise method with an argument of "Bang" (the type of noise to make). Since the balloon is no longer inflated, the Inflated property is set to False and the Diameter property is set to a new value.

While you can't actually program a balloon, you can program a Visual Basic form or control. As the programmer, you are in control. You decide which properties should be changed, methods invoked or events responded to in order to achieve the desired appearance and behavior.

Designing a Form

Form objects are the basic building blocks of a Visual Basic application, the actual windows with which a user interacts when they run the application. Forms have their own properties, events, and methods with which you can control their appearance and behavior.

Figure 3.2   Forms and controls have their own properties, events, and methods



The first step in designing a form is to set its properties. You can set a form's properties at *design time* in the Properties window, or at *run time* by writing code.

> **Note**   You work with forms and controls, set their properties, and write code for their events at *design time*, which is any time you're building an application in the Visual Basic environment. *Run time* is any time you are actually running the application and interacting with the application as the user would.

Setting Form Properties

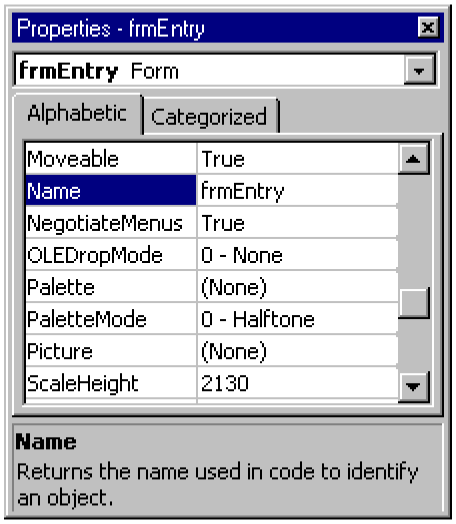
Many of a form's properties affect its physical appearance. The Caption property determines the text that is shown in the form's title bar; the Icon property sets the icon that is displayed when a form is minimized. The MaxButton and MinButton properties determine whether the form can be maximized or minimized. By changing the BorderStyle property, you can control the resizing behavior of the form.

Height and Width properties determine the initial size of a form; Left and Top properties determine the form's location in relation to the upper left-hand corner of the screen. The WindowState property can be set to start the form in a maximized, minimized, or normal state.

The Name property sets the name by which you will refer to the form in code. By default, when a form is first added to a project, its name is set to Form1, Form2, and so forth. It's a good idea to set the Name property to something more meaningful, such as "frmEntry" for an order entry form.

The best way to familiarize yourself with the many form properties is to experiment. Change some of the properties of a form in the Properties window (Figure 3.3), then run the application to see their effect. You can learn more about each property by selecting it and pressing F1 to view the context-sensitive Help.

Figure 3.3 The Properties window



Moveable	True
Name	frmEntry
NegotiateMenus	True
OLEDropMode	0 - None
Palette	(None)
PaletteMode	0 - Halftone
Picture	(None)
ScaleHeight	2130

Name
Returns the name used in code to identify an object.

Form Events and Methods

As objects, forms can perform methods and respond to events.

The Resize event of a form is triggered whenever a form is resized, either by user interaction or through code. This allows you to perform actions such as moving or resizing controls on a form when its dimensions have changed.

The Activate event occurs whenever a form becomes the active form; the Deactivate event occurs when another form or application becomes active. These events are convenient for initializing or finalizing the form's behavior. For example, in the Activate event you might write code to highlight the text in a particular text box; in the Deactivate event you might save changes to a file or database.

To make a form visible, you would invoke the Show method:

```
Form2.Show
```

Invoking the Show method has the same effect as setting a form's Visible property to True.

Many of a form's methods involve text or graphics. The Print, Line, Circle, and Refresh methods are useful for printing or drawing directly onto a form's surface. These methods and more are discussed in "Working with Text and Graphics."

For More Information For additional information on forms, see "More About Forms" in "Creating a User Interface."

Clicking Buttons to Perform Actions

The easiest way to allow the user to interact with an application is to provide a button to click. You can use the command button control provided by Visual Basic, or you can create your own "button" using an image control containing a graphic, such as an icon.

Using Command Buttons

Most Visual Basic applications have command buttons that allow the user to simply click them to perform actions. When the user chooses the button, it not only carries out the appropriate action, it also looks as if it's being pushed in and released. Whenever the user clicks a button, the Click event procedure is invoked. You place code in the Click event procedure to perform any action you choose.

There are many ways to choose a command button at run time:

- Use a mouse to click the button.
- Move the focus to the button by pressing the TAB key, and then choose the button by pressing the SPACEBAR or ENTER. (See "Understanding Focus" later in this chapter.)
- Press an access key (ALT+ the underlined letter) for a command button.
- Set the command button's Value property to True in code:

  ```
  cmdClose.Value = True
  ```

- Invoke the command button's Click event in code:

  ```
  cmdClose_Click
  ```

- If the command button is the *default command button* for the form, pressing ENTER chooses the button, even if you change the focus to a different control other than a command button. At design time, you specify a default command button by setting that button's Default property to True.
- If the command button is the default *Cancel button* for the form, then pressing ESC chooses the button, even if you change the focus to another control. At design time, you specify a default Cancel button by setting that button's Cancel property to True.

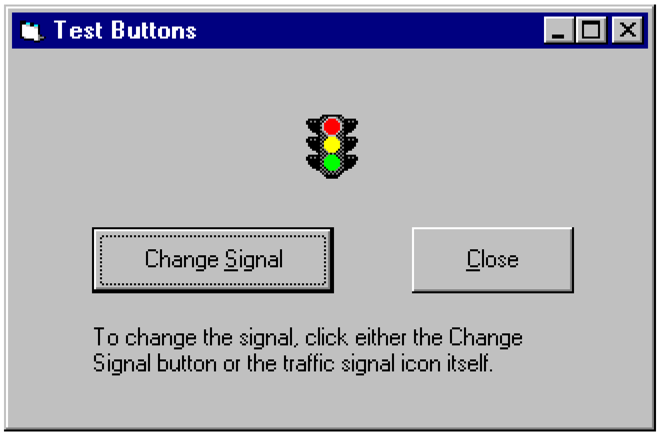
All these actions cause Visual Basic to invoke the Click event procedure.

The Test Buttons Application

You use the Caption property to display text on the button to tell the user what the button does. In Figure 3.4, the Test Buttons example from the Controls sample application contains a command button with its Caption property set to "Change Signal." (For a working version of this example, see Button.frm in the Controls.vbp sample application.)

Notice that 'S' is the access key for this button, denoted by an underline. Inserting an ampersand (&) in the text of the Caption property makes the character following it the access key for that button (for example, Change &Signal).

Figure 3.4 Command button with a caption



To change the signal, click either the Change Signal button or the traffic signal icon itself.

When a user clicks the command button, the code in the command button's Click event procedure is executed. In the example, a different traffic light icon is displayed each time the button is clicked.

For More Information For information on additional properties of the command button, see "Using Visual Basic's Standard Controls."

Send feedback to MSDN. Look here for MSDN Online resources.

Visual Basic Concepts

Controls for Displaying and Entering Text

Label and text box controls are used to display or enter text. Use labels when you want your application to display text on a form, and text boxes when you want to allow the user to enter text. Labels contain text that can only be read, while text boxes contain text that can be edited.

To provide this feature	Use this control
Text that can be edited by the user, for example an order entry field or a password box	Text box

Text that is displayed only, for example to identify a field on a form or display instructions to the user	Label

Labels and text boxes are discussed in the following sections:

- Using Labels to Display Text   The basics of using the label control.
- Working with Text Boxes   The basics of using text boxes.

Send feedback to MSDN.   Look here for MSDN Online resources.

Using Labels to Display Text

A label control displays text that the user cannot directly change. You can use labels to identify controls, such as text boxes and scroll bars, that do not have their own Caption property. The actual text displayed in a label is controlled by the Caption property, which can be set at design time in the Properties window or at run time by assigning it in code.

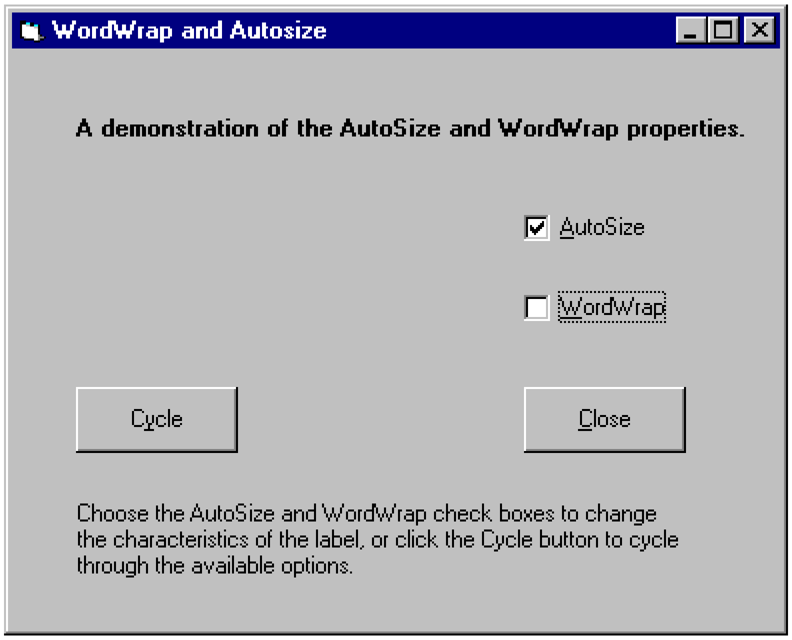
By default, the caption is the only visible part of the label control. However, if you set the BorderStyle property to 1 (which you can do at design time), the label appears with a border — giving it a look similar to a text box. You can also change the appearance of the label by setting the BackColor, BackStyle, ForeColor, and Font properties.

Sizing a Label to Fit Its Contents

Single-line label captions can be specified at design time in the Properties window. But what if you want to enter a longer caption, or a caption that will change at run time? Labels have two properties that help you size the controls to fit larger or smaller captions: AutoSize and WordWrap.

The AutoSize property determines if a control should be automatically resized to fit its contents. If set to True, the label grows horizontally to fit its contents, as shown in Figure 3.5.

Figure 3.5   AutoSize example

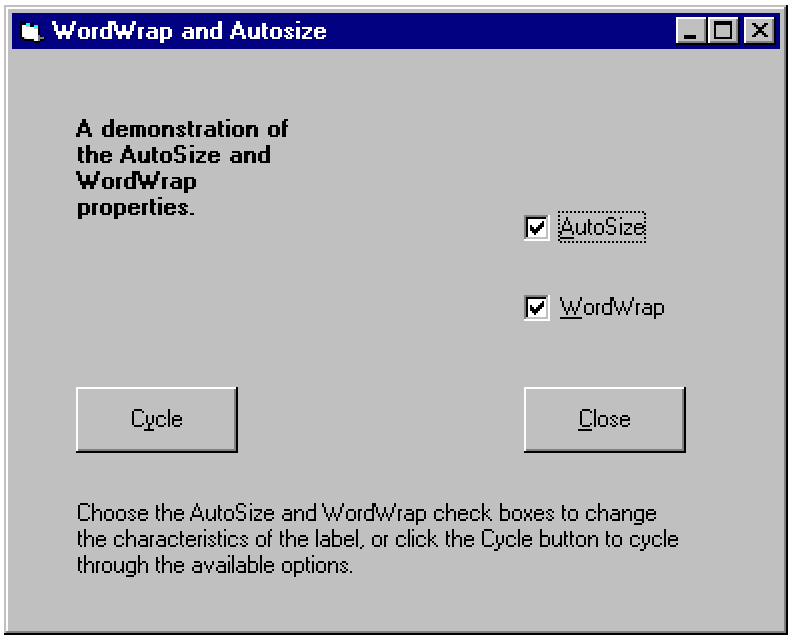


A demonstration of the AutoSize and WordWrap properties.

Choose the AutoSize and WordWrap check boxes to change the characteristics of the label, or click the Cycle button to cycle through the available options.

The WordWrap property causes the label to grow vertically to fit its contents, while retaining the same width, as shown in Figure 3.6. For a working version of this example, see Wordwrap.frm in the Controls.vbp sample application.

Figure 3.6 WordWrap example



A demonstration of the AutoSize and WordWrap properties.

Choose the AutoSize and WordWrap check boxes to change the characteristics of the label, or click the Cycle button to cycle through the available options.

> **Note** If you run the AutoSize example from Controls.vbp, you'll notice that for the WordWrap example to actually work, both check boxes must be selected. This is because, for the label's WordWrap property to take effect, AutoSize must be set to True. The width of the label is increased only if the width of a single word exceeds the current width of the control.

For More Information For additional information on the label control's properties, see "Using Visual Basic's Standard Controls."

Working with Text Boxes

Text boxes are versatile controls that can be used to get input from the user or to display text. Text boxes should not be used to display text that you don't want the user to change, unless you've set the Locked property to True.

The actual text displayed in a text box is controlled by the Text property. It can be set in three different ways: at design time in the Property window, at run time by setting it in code, or by input from the user at run time. The current contents of a text box can be retrieved at run time by reading the Text property.

Multiple-Line Text Boxes and Word Wrap

By default, a text box displays a single line of text and does not display scroll bars. If the text is longer than the available space, only part of the text will be visible. The look and behavior of a text box can be changed by setting two properties, MultiLine and ScrollBars, which are available only at design time.

> **Note** The ScrollBars property should not be confused with scroll bar controls, which are not attached to text boxes and have their own set of properties.

Setting MultiLine to True enables a text box to accept or display multiple lines of text at run time. A multiple-line text box automatically manages word wrap as long as there is no horizontal scroll bar. The ScrollBars property is set to 0-None by default. Automatic word wrap saves the user the trouble of inserting line breaks at the end of lines. When a line of text is longer than what can be displayed on a line, the text box wraps the text to the next line.

Line breaks cannot be entered in the Properties window at design time. Within a procedure, you create a line break by inserting a carriage return followed by a linefeed (ANSI characters 13 and 10). You can also use the constant vbCrLf to insert a carriage return/linefeed combination. For example, the following event procedure puts two lines of text into a multiple-line text box (Text1) when the form is loaded:

```
Sub Form_Load ()
   Text1.Text = "Here are two lines" _
   & vbCrLf & "in a text box"
End Sub
```

Working with Text in a Text Box

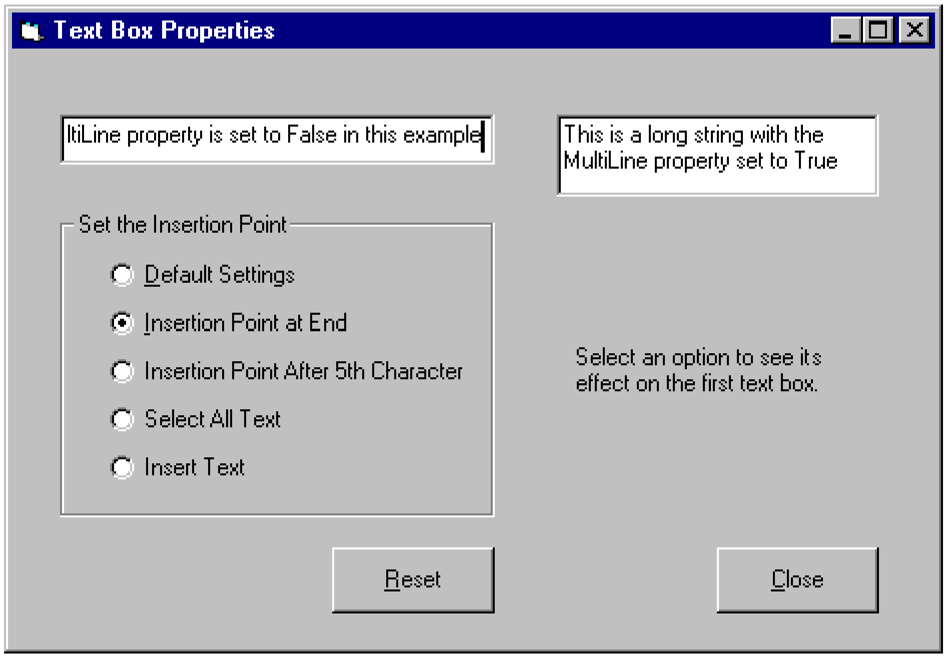
You can control the insertion point and selection behavior in a text box with the SelStart, SelLength and SelText properties. These properties are only available at run time.

When a text box first receives the focus, the default insertion point or cursor position within the text box is to the left of any existing text. It can be moved by the user from the keyboard or with the mouse. If the text box loses and then regains the focus, the insertion point will be wherever the user last placed it.

In some cases, this behavior can be disconcerting to the user. In a word processing application, the user might expect new characters to appear after any existing text. In a data entry application, the user might expect their typing to replace any existing entry. The SelStart and SelLength properties allow you to modify the behavior to suit your purpose.

The SelStart property is a number that indicates the insertion point within the string of text, with 0 being the left-most position. If the SelStart property is set to a value equal to or greater than the number of characters in the text box, the insertion point will be placed after the last character, as shown in Figure 3.7. For a working version of this example, see Text.frm in the Controls.vbp sample application.

Figure 3.7 Insertion point example


Text Box Properties

ltiLine property is set to False in this example

This is a long string with the MultiLine property set to True

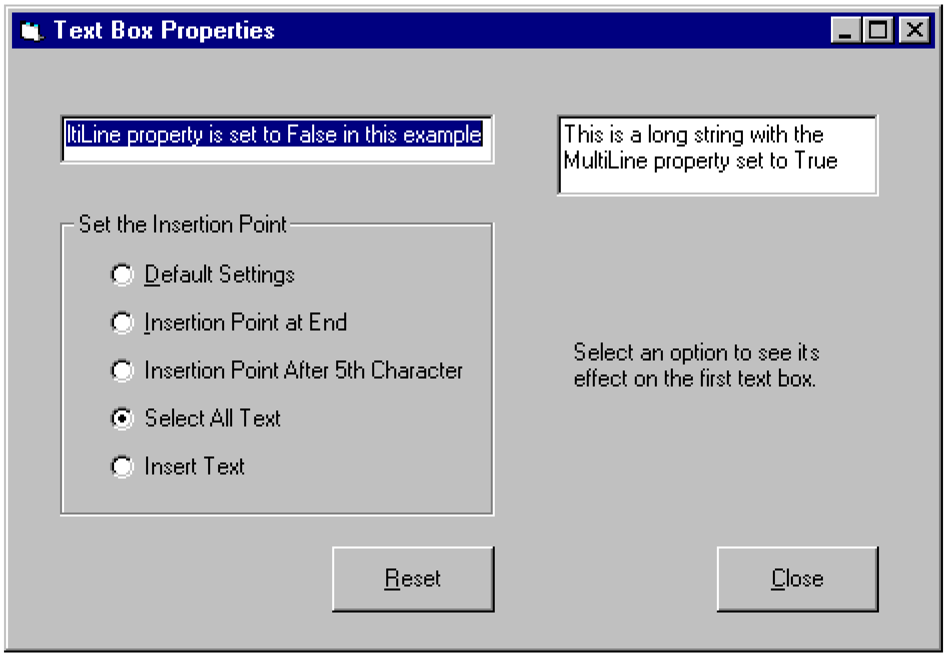
Set the Insertion Point

- Default Settings
- [✓ Insertion Point at End]
- Insertion Point After 5th Character
- Select All Text
- Insert Text

Select an option to see its effect on the first text box.

The SelLength property is a numeric value that sets the width of the insertion point. Setting the SelLength to a number greater than 0 causes that number of characters to be selected and highlighted, starting from the current insertion point. Figure 3.8 shows the selection behavior.

Figure 3.8 Selection example


Text Box Properties
ltiLine property is set to False in this example
This is a long string with the MultiLine property set to True
Set the Insertion Point
Default Settings
Insertion Point at End
Insertion Point After 5th Character
Select All Text
Insert Text
Select an option to see its effect on the first text box.
Reset
Close

If the user starts typing while a block of text is selected, the selected text will be replaced. In some cases, you might want to replace a text selection with new text by using a paste command. The SelText property is a string of text that you can assign at run time to replace the current selection. If no text is selected, SelText will insert its text at the current insertion point.

For More Information For additional information on the text box control's properties, see "Using Visual Basic's Standard Controls."

Controls That Present Choices to Users

Most applications need to present choices to their users, ranging from a simple yes/no option to selecting from a list containing hundreds of possibilities. Visual Basic includes several standard controls that are useful for presenting choices. The following table summarizes these controls and their appropriate uses.

To provide this feature	Use this control
A small set of choices from which a user can choose one or more options.	Check boxes
A small set of options from which a user can choose just one.	Option buttons (use frames if additional groups are needed)
A scrollable list of choices from which the user can choose.	List box

A scrollable list of choices along with a text edit field. The user can either choose from the list or type a choice in the edit field.	Combo box

Check boxes, option buttons, list boxes, and combo boxes are discussed in the following sections:

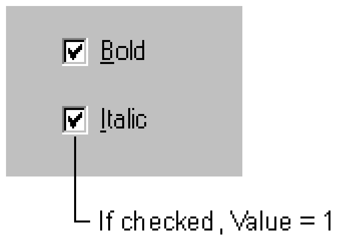
- Selecting Individual Options with Check Boxes   The basics of using the check box control.
- Grouping Options with Option Buttons   The basics of using the option button control.
- Using List Boxes and Combo Boxes   An introduction to the list box and combo box controls.
- Using Scroll Bars as Input Devices   A brief introduction to the scroll bar control.

Send feedback to MSDN.   *Look here for MSDN Online resources.*

Selecting Individual Options with Check Boxes

A check box indicates whether a particular condition is on or off. You use check boxes in an application to give users true/false or yes/no options. Because check boxes work independently of each other, a user can select any number of check boxes at the same time. For example in Figure 3.9, Bold and Italic can both be checked.

Figure 3.9   Check boxes

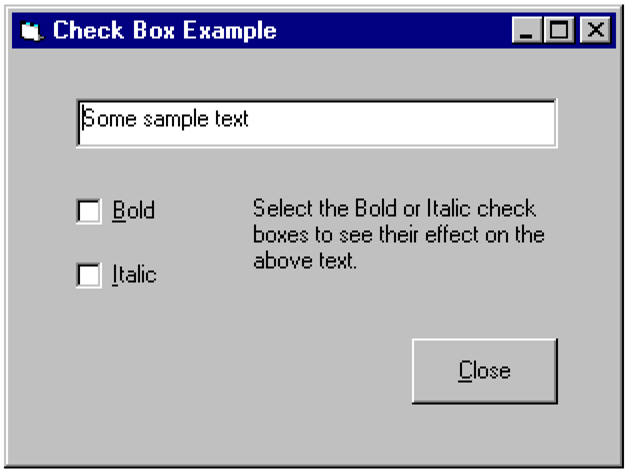


The Check Box Application

The Check Box example uses a check box to determine whether the text is displayed in regular or italic font. For a working version of this example, see Check.frm in the Controls.vbp sample application.

The application has a text box, a label, a command button, and two check boxes, as shown in Figure 3.10.

Figure 3.10   Check box example


Check Box Example
Some sample text
Bold
Italic
Select the Bold or Italic check boxes to see their effect on the above text.
Close

The following table lists the property settings for the objects in the application.

Object	Property	Setting
Form	Name Caption	frmCheck Check Box Example
Text box	Name Text	txtDisplay Some sample text
First Check box	Name Caption	chkBold &Bold
Second Check box	Name Caption	chkItalic &Italic
Command button	Name Caption	cmdClose &Close

When you check Bold or Italic, the check box's Value property is set to 1; when unchecked, its Value property is set to 0. The default Value is 0, so unless you change Value, the check box will be unchecked when it is first displayed. You can use the constants vbChecked and vbUnchecked to represent the values 1 and 0.

Events in the Check Box Application

The Click event for the check box occurs as soon as you click the box. This event procedure tests to see whether the check box has been selected (that is, if its Value = vbChecked). If so, the text is converted to bold or italic by setting the Bold or Italic properties of the Font object returned by the Font property of the text box.

```
Private Sub chkBold_Click ()
   If ChkBold.Value = vbChecked Then    ' If checked.
      txtDisplay.Font.Bold = True
   Else                             ' If not checked.
      txtDisplay.Font.Bold = False
   End If
End Sub

Private Sub chkItalic_Click ()
```

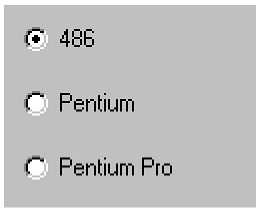
```
   If ChkItalic.Value = vbChecked Then    ' If checked.
      txtDisplay.Font.Italic = True
   Else                            ' If not checked.
      txtDisplay.Font.Italic = False
   End If
End Sub
```

Grouping Options with Option Buttons

Option buttons present a set of two or more choices to the user. Unlike check boxes, however, option buttons should always work as part of a group; selecting one option button immediately clears all the other buttons in the group. Defining an option button group tells the user, "Here is a set of choices from which you can choose one and only one."

For example, in the option button group shown in Figure 3.11, the user can select one of three option buttons.

Figure 3.11   Selecting an option button

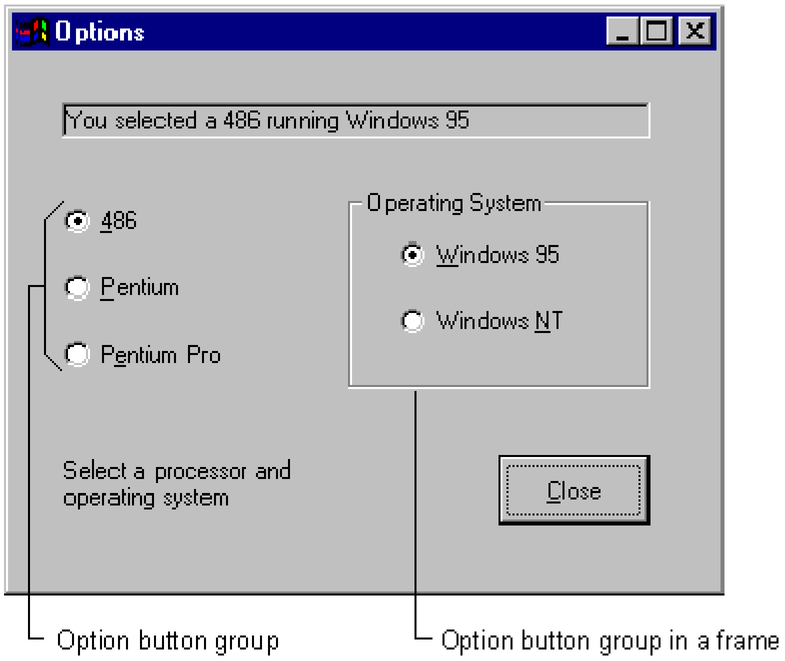


Creating Option Button Groups

All of the option buttons placed directly on a form (that is, not in a frame or picture box) constitute one group. If you want to create additional option button groups, you must place some of them inside frames or picture boxes.

All the option buttons inside any given frame constitute a separate group, as do all the option buttons inside a picture box. When you create a separate group this way, always draw the frame or picture box first, and then draw the option buttons on top of it. Figure 3.12 shows a form with two option button groups.

Figure 3.12   Option button groups


Options
You selected a 486 running Windows 95
486
Pentium
Pentium Pro
Operating System
Windows 95
Windows NT
Select a processor and operating system
Close
Option button group
Option button group in a frame

A user can select only one option button in the group when you draw option buttons in a frame.

To group controls in a frame

1. Select the frame control from the toolbox and draw the frame on the form.
2. Select the option button control from the toolbox and draw the control within the frame.
3. Repeat step 2 for each additional option button you wish to add to the frame.

Drawing the frame first and then drawing each control on the frame allows you to move the frame and controls together. If you try to move existing controls onto a frame, the controls will not move with the frame.

> **Note** If you have existing controls that you want to group in a frame, you can select all the controls and cut and paste them into a frame or picture control.

Containers for Controls

While controls are independent objects, a certain *parent and child relationship* exists between forms and controls. Figure 3.12 demonstrates how option buttons can be contained within a form or within a frame control.

To understand the concept of containers, you need to understand that all controls are children of the form on which they are drawn. In fact, most controls support the read-only Parent property, which returns the form on which a control is located. Being a child affects the placement of a control on the parent form. The Left and Top properties of a control are relative to the parent form, and controls cannot be moved outside the boundaries of the parent. Moving a container moves the controls as well, and the control's position relative to the container's Left and Top properties does not change because the control moves with the container.

Selecting or Disabling Option Buttons

An option button can be selected by:

- Clicking it at run time with the mouse.
- Tabbing to the option button group and then using the arrow keys to select an option button within the group.
- Assigning its Value property to True in code:

  ```
  optChoice.Value = True
  ```

- Using a shortcut key specified in the caption of a label.

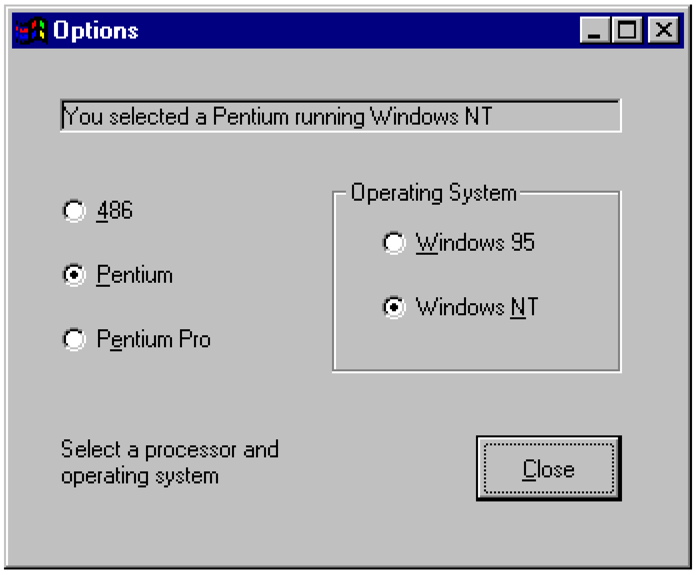
To make a button the default in an option button group, set its Value property to True at design time. It remains selected until a user selects a different option button or code changes it.

To disable an option button, set its Enabled property to False. When the program is run it will appear dimmed, meaning that it is unavailable.

The Options Application

The form shown in Figure 3.13 uses option buttons to determine the processor type and operating system for a fictional computer. When the user selects a option button in either group, the caption of the label is changed to reflect the current choices. For a working version of this example, see Options.frm in the Controls.vbp sample application.

Figure 3.13 Option button example


Options
You selected a Pentium running Windows NT
486
Pentium
Pentium Pro
Operating System
Windows 95
Windows NT
Select a processor and operating system
Close

The following table lists the property settings for the objects in the application.

Object	Property	Setting
Label	Name Caption	lblDisplay (Empty)

Command button	Name Caption	cmdClose &Close
First option button	Name Caption	opt486 &486
Second option button	Name Caption Value	opt586 &Pentium True
Third option button	Name Caption	opt686 P&entium Pro
Frame	Name Caption	fraSystem &Operating System
Fourth option button	Name Caption	optWin95 Windows 95
Fifth option button	Name Caption Value	optWinNT Windows NT True

Events in the Options Application

The Options application responds to events as follows:

- The Click events for the first three option buttons assign a corresponding description to a form-level string variable, `strComputer`.
- The Click events for the last two option buttons assign a corresponding description to a second form-level variable, `strSystem`.

The key to this approach is the use of these two form-level variables, `strComputer` and `strSystem`. These variables contain different string values, depending on which option buttons were last selected.

Each time a new option button is selected, the code in its Click event updates the appropriate variable:

```
Private Sub opt586_Click()
   strComputer = "Pentium"
   Call DisplayCaption
End Sub
```

It then calls a sub procedure, called DisplayCaption, that concatenates the two variables and updates the label's Caption property:

```
Sub DisplayCaption()
   lblDisplay.Caption = "You selected a " & _
   strComputer & " running " & strSystem
End Sub
```

A sub procedure is used because the procedure of updating the Caption property is essentially the same for all five option buttons, only the value of the variables change from one instance to the next. This saves you from having to repeat the same code in each of the Click events.

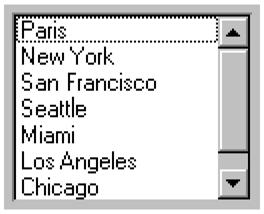
For More Information Variables and sub procedures are discussed in detail in "Programming Fundamentals."

Send feedback to MSDN. Look here for MSDN Online resources.

Using List Boxes and Combo Boxes

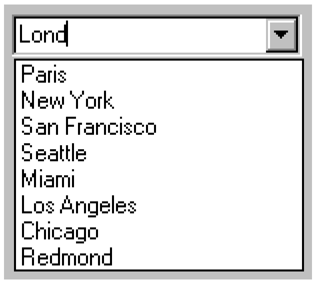
List boxes and combo boxes present a list of choices to the user. By default, the choices are displayed vertically in a single column, although you can set up multiple columns as well. If the number of items exceeds what can be displayed in the combo box or list box, scroll bars automatically appear on the control. The user can then scroll up and down or left to right through the list. Figure 3.14 shows a single-column list box.

Figure 3.14 Single-column list box



A combo box control combines the features of a text box and a list box. This control allows the user to select either by typing text into the combo box or by selecting an item from its list. Figure 3.15 shows a combo box.

Figure 3.15 Combo box



In contrast to some other controls that contain a single value; for example the label's Caption property or the text box's Text property, list boxes and combo boxes contain multiple values or a collection of values. They have built-in methods for adding, removing and retrieving values from their collections at run time. To add several items to a list box named List1, the code would look like this:

```
List1.AddItem "Paris"
```

```
List1.AddItem "New York"
List1.AddItem "San Francisco"
```

List boxes and combo boxes are an effective way to present a large number of choices to the user in a limited amount of space.

For More Information For additional information on the list box and combo box controls, see "Using Visual Basic's Standard Controls."

Using Scroll Bars as Input Devices

Although scroll bars are often tied to text boxes or windows, you'll sometimes see them used as input devices. Because these controls can indicate the current position on a scale, scroll bar controls can be used individually to control program input — for example, to control the sound volume or to adjust the colors in a picture. The HScrollBar (horizontal) and VScrollBar (vertical) controls operate independently from other controls and have their own set of events, properties, and methods. Scroll bar controls are not the same as the built-in scroll bars that are attached to text boxes, list boxes, combo boxes, or MDI forms (text boxes and MDI forms have a ScrollBars property to add or remove scroll bars that are attached to the control).

Windows interface guidelines now suggest using slider controls as input devices instead of scroll bars. Examples of slider controls can be seen in the Windows 95/98 control panel. A slider control of this type is included in the Professional and Enterprise editions of Visual Basic.

For More Information For additional information on scroll bar controls, see "Using Visual Basic's Standard Controls."

Controls That Display Pictures and Graphics

Because Windows is a graphical user interface, it's important to have a way to display graphical images in your application's interface. Visual Basic includes four controls that make it easy to work with graphics: the picture box control, the image control, the shape control, and the line control.

The image, shape and line controls are sometimes referred to as "lightweight" graphical controls. They require less system resources and consequently display somewhat faster than the picture box control; they contain a subset of the properties, methods and events available in the picture box. Each is best suited for a particular purpose.

To provide this feature	Use this control
A container for other controls.	Picture box
Printing or graphics methods.	Picture box
Displaying a picture.	Image control or picture box
Displaying a simple graphical element	Shape or line control

Picture boxes, image controls, shape controls, and line controls are discussed in the following sections:

- Working with the Picture Box Control   The basics of using the picture box control.
- Lightweight Graphical Controls   The basics of using the image, shape, and line controls.
- The Images Application   An example of using the graphical controls.

Working With the Picture Box Control

The primary use for the picture box control is to display a picture to the user. The actual picture that is displayed is determined by the Picture property. The Picture property contains the file name (and optional path) for the picture file that you wish to display.

> **Note**   Form objects also have a Picture property that can be set to display a picture directly on the form's background.

To display or replace a picture at run time, you can use the LoadPicture function to set the Picture property. You supply the name (and optional path) for the picture and the LoadPicture function handles the details of loading and displaying it:

```
picMain.Picture = LoadPicture("VANGOGH.gif")
```

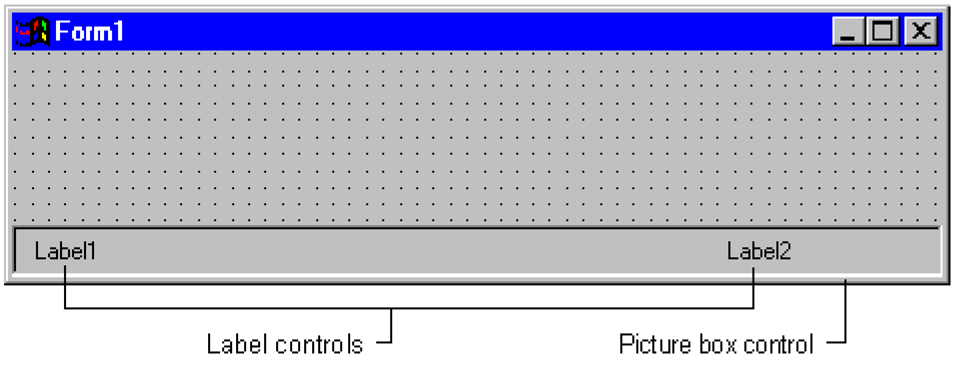
The picture box control has an AutoSize property that, when set to True, causes the picture box to resize automatically to match the dimensions of its contents. Take extra care in designing your form if you plan on using a picture box with the AutoSize enabled. The picture will resize without regard to other controls on the form, possibly causing unexpected results, such as covering up other controls. It's a good idea to test this by loading each of the pictures at design time.

Using the Picture Box as a Container

The picture box control can also be used as a container for other controls. Like the frame control, you can draw other controls on top of the picture box. The contained controls move with the picture box and their Top and Left properties will be relative to the picture box rather than the form.

A common use for the picture box container is as a toolbar or status bar. You can place image controls on it to act as buttons, or add labels to display status messages. By setting the Align property to Top, Bottom, Left, or Right, the picture box will "stick" to the edge of the form. Figure 3.16 shows a picture box with its Align property set to Bottom. It contains two label controls which could be used to display status messages.

Figure 3.16   Picture box used as a status bar



Other Uses for the Picture Box

The picture box control has several methods that make it useful for other purposes. Think of the picture box as a blank canvas upon which you can paint, draw or print. A single control can be used to display text, graphics or even simple animation.

The Print method allows you to output text to the picture box control just as you would to a printer. Several font properties are available to control the characteristics of text output by the Print method; the Cls method can be used to erase the output.

Circle, Line, Point and Pset methods may be used to draw graphics on the picture box. Properties such as DrawWidth, FillColor, and FillStyle allow you to customize the appearance of the graphics.

Animation can be created using the PaintPicture method by moving images within the picture control and rapidly changing between several different images.

For More Information   For additional information on the picture box control, see "Using Visual Basic's Standard Controls."

Lightweight Graphical Controls

The image, shape and line controls are considered to be lightweight controls; that is, they support only a subset of the properties, methods, and events found in the picture box. Because of this, they typically require less system resources and load faster than the picture box control.

Using Image Controls Instead of Picture Boxes

The image control is similar to the picture box control but is used only for displaying pictures. It doesn't have the ability to act as a container for other controls, and it doesn't support the advanced methods of the picture box.

Pictures are loaded into the image control just as they are in the picture box: at design time, set the Picture property to a file name and path; at run time, use the LoadPicture function.

The sizing behavior of the image control differs from that of the picture box. It has a Stretch property while the picture box has an AutoSize property. Setting the AutoSize property to True causes a picture box to resize to the dimensions of the picture; setting it to False causes the picture to be cropped (only a portion of the picture is visible). When set to False (the default), the Stretch property of the image control causes it to resize to the dimensions of the picture. Setting the Stretch property to True causes the picture to resize to the size of the image control, which may cause the picture to appear distorted.

For More Information For additional information on the image control, see "Using Visual Basic's Standard Controls."

Using an Image Control to Create Your Own Buttons

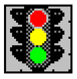
An image control also recognizes the Click event, so you can use this control anywhere you'd use a command button. This is a convenient way to create a button with a picture instead of a caption. Grouping several image controls together horizontally across the top of the screen — usually within a picture box — allows you to create a toolbar in your application.

For instance, the Test Buttons example shows an image control that users can choose like they choose a command button. When the form is first displayed, the control displays one of three traffic icons from the Icon Library included with Visual Basic. Each time the image control is clicked, a different icon is displayed. (For a working version of this example, see Button.frm in the Controls.vbp sample application.)

If you inspect the form at design time, you will see that it actually contains all three icons "stacked" on top of each other. By changing the Visible property of the top image control to False, you allow the next image (with its Visible property set to True) to appear on top.

Figure 3.17 shows the image control with one of the traffic icons (Trffc10a.ico).

Figure 3.17 Image control with a traffic icon



To create a border around the image control, set the BorderStyle property to 1-Fixed Single.

Note Unlike command buttons, image controls do not appear pushed in when clicked. This means that unless you change the bitmap in the MouseDown event, there is no visual cue to the user that the "button" is being pushed.

For More Information For information on displaying a graphic image in an image control, see "Using Visual Basic's Standard Controls."

Using Shape and Line Controls

Shape and line controls are useful for drawing graphical elements on the surface of a form. These controls don't support any events; they are strictly for decorative purposes.

Several properties are provided to control the appearance of the shape control. By setting the Shape property, it can be displayed as a rectangle, square, oval, circle, rounded rectangle, or rounded square. The BorderColor and FillColor properties can be set to change the color; the BorderStyle, BorderWidth, FillStyle, and DrawMode properties control how the shape is drawn.

The line control is similar to the shape control but can only be used to draw straight lines.

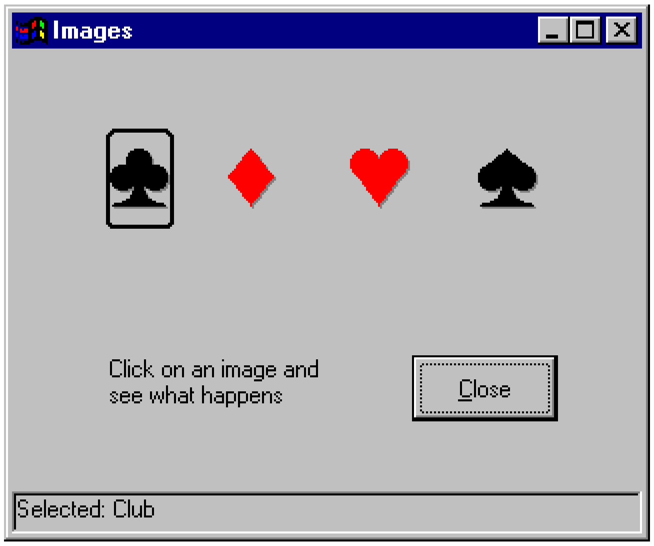
For More Information For additional information on the shape and line controls, see "Using Visual Basic's Standard Controls."

Send feedback to MSDN. Look here for MSDN Online resources.

The Images Application

The form shown in Figure 3.18 uses four image controls, a shape control, a picture box, and a command button. When the user selects a playing card symbol, the shape control highlights the symbol and a description is displayed in the picture box. For a working version of this example, see Images.frm in the Controls.vbp sample application.

Figure 3.18 Image and shape control example



The following table lists the property settings for the objects in the application.

Object	Property	Setting
Picture box	Name Align	picStatus Bottom
First image control	Name Picture	imgClub Spade.ico
Second image control	Name Picture	imgDiamond Diamond.ico
Third image control	Name Picture	imgHeart Heart.ico
Fourth image control	Name Caption	imgSpade Spade.ico
Shape control	Name Shape BorderWidth Height Width	shpCard 4 - Rounded Rectangle 2 735 495
Command button	Name Caption	cmdClose &Close

Events in the Images Application

The Images application responds to events as follows:

- The Click event in each of the image controls sets the Left property of the shape control equal to its own Left property, moving the shape on top of the image.
- The Cls method of the picture box is invoked, clearing the current caption from the status bar.
- The Print method of the picture box is invoked, printing the new caption on the status bar.

The code in the image control Click event looks like this:

```
Private Sub imgHeart_Click()
   shpCard.Left = imgClub.Left
   picStatus.Cls
   picStatus.Print "Selected: Club"
   shpCard.Visible = True
End Sub
```

Note that the first line in the Click event code assigns a value (the Left property of the image control) to the Left property of the shape control using the = operator. The next two lines invoke methods, so no operator is needed. In the third line, the value ("Selected: Club") is an argument to the Print method.

There is one more line of code in the application that is of interest; it is in the Form Load event.

```
shpCard.Visible = False
```

By setting the Visible property of the shape control to False, the shape control is hidden until the first image is clicked. The Visible property is set to True as the last step in the image control Click event.

For More Information For additional information on properties, methods, and events see "Programming Fundamentals."

Additional Controls

Several other standard controls are included in the Visual Basic toolbox. Some controls are useful for working with large amounts of data contained in an external database. Other controls can be used to access the Windows file system. Still other controls defy categorization, but are useful nonetheless.

You can also use ActiveX controls, previously called custom or OLE controls, in a Visual Basic application in the same way that you use the standard controls. The Professional and Enterprise editions of Visual Basic include several ActiveX controls as well as the capability to build your own controls. Additional ActiveX controls for just about any purpose imaginable are available for purchase from numerous vendors.

For More Information For additional information on using ActiveX controls, see "Managing Projects."

Data Access Controls

In today's business, most information is stored in one or more central databases. Visual Basic includes several data access controls for accessing most popular databases, including Microsoft Access and SQL Server.

- The ADO Data control is used to connect to a database. Think of it as a pipeline between the database and the other controls on your form. Its properties, methods, and events allow you to navigate and manipulate external data from within your own application.

- The DataList control is similar to the list box control. When used in conjunction with an ADO Data control, it can be automatically filled with a list of data from a field in an external database.

- The DataCombo control is like a combination of the DataList control and a text box. The selected text in the text box portion can be edited, with the changes appearing in the underlying database.

- The DataGrid control displays data in a grid or table. When used in conjunction with an ADO Data control, it presents fully editable data from multiple fields in an external database.
- The Microsoft Hierarchical FlexGrid control is a unique control for presenting multiple views of data. Think of it as a combination of a grid and a tree or outline control. At run time, the user can rearrange columns and rows to provide different views of the data.

For More Information For additional information on data access controls, see "Using Visual Basic's Standard Controls." For more information on working with external data, see the *Visual Basic Data Access Guide*.

File System Controls

Visual Basic includes three controls for adding file handling capabilities to your application. These controls are normally used together to provide a view of drives, directories and files; they have special properties and events that tie them together.

- The DriveListBox control looks like a combo box. It provides a drop-down list of drives from which the user can select.
- The DirListBox is similar to a list box control, but with the built-in capability of displaying a list of directories in the currently selected drive.
- The FileListBox control also looks like a list box with a list of file names in a selected directory.

Note These controls are provided primarily for backward compatibility with applications created in earlier versions of Visual Basic. The common dialog control provides an easier method of working with file access. For more information on common dialog control, see "Miscellaneous Controls" later in this chapter.

Miscellaneous Controls

Several other standard controls are included in Visual Basic. Each serves a unique purpose.

- The timer control can be used to create an event in your application at a recurring interval. This is useful for executing code without the need for user interaction.
- The OLE container control is an easy way to add capabilities like linking and embedding to your application. Through the OLE container control, you can provide access to the functionality of any OLE-enabled application such as Microsoft Excel, Word and many others.
- The common dialog control adds built-in dialog boxes to your application for the selection of files, colors, fonts, and printing functions.

For More Information For additional information on any of the standard controls, see "Using Visual Basic's Standard Controls."

Understanding Focus

Focus is the ability to receive user input through the mouse or keyboard. When an object has the focus, it can receive input from a user. In the Microsoft Windows interface, several applications can be running at any time, but only the application with the focus will have an active title bar and can receive user input. On a Visual Basic form with several text boxes, only the text box with the focus will display text entered by means of the keyboard.

The GotFocus and LostFocus events occur when an object receives or loses focus. Forms and most controls support these events.

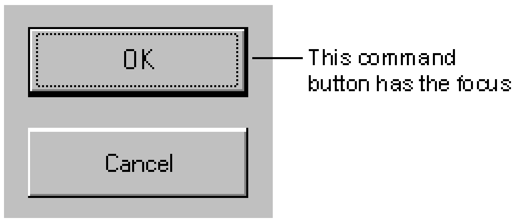
Event	Description
GotFocus	Occurs when an object receives focus.
LostFocus	Occurs when an object loses focus. A LostFocus event procedure is primarily used for verification and validation updates, or for reversing or changing conditions you set up in the object's GotFocus procedure.

You can give focus to an object by:

- Selecting the object at run time.
- Using an access key to select the object at run time.
- Using the SetFocus method in code.

You can see when some objects have the focus. For example, when command buttons have the focus, they appear with a highlighted border around the caption (see Figure 3.19).

Figure 3.19 A command button showing focus



An object can receive focus only if its Enabled and Visible properties are set to True. The Enabled property allows the object to respond to user-generated events such as keyboard and mouse events. The Visible property determines whether an object is visible on the screen.

Note A form can receive focus only if it doesn't contain any controls that can receive the focus.

Validate Event of Controls

Controls also have a Validate event, which occurs before a control loses focus. However, this event occurs only when the CausesValidation property of the control that is about to receive the focus is set to True. In many cases, because the Validate event occurs before the focus is lost, it is more appropriate than the LostFocus event for data validation. For more information, see "Validating Control Data By Restricting Focus" in "Using Visual Basic's Standard

Controls That Can't Receive Focus

Some controls, such as the lightweight controls, cannot receive focus. Lightweight controls include the following:

- Frame control
- Image control
- Label control
- Line control
- Shape control

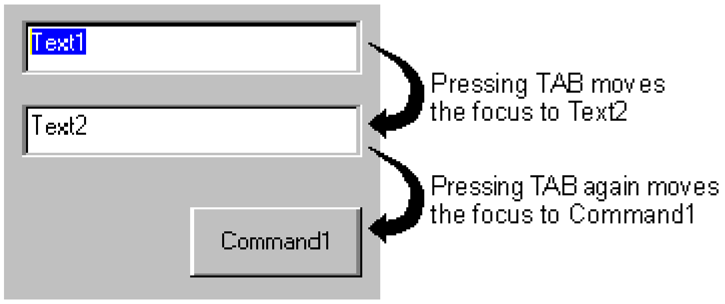
Additionally, controls that are invisible at run time, such as the Timer control, cannot receive focus.

Setting the Tab Order

The *tab order* is the order in which a user moves from one control to another by pressing the TAB key. Each form has its own tab order. Usually, the tab order is the same as the order in which you created the controls.

For example, assume you create two text boxes, Text1 and Text2, and then a command button, Command1. When the application starts, Text1 has the focus. Pressing TAB moves the focus between controls in the order they were created, as shown in Figure 3.20.

Figure 3.20   Tab example


Text1
Text2
Command1
Pressing TAB moves the focus to Text2
Pressing TAB again moves the focus to Command1

To change the tab order for a control, set the TabIndex property. The TabIndex property of a control determines where it is positioned in the tab order. By default, the first control drawn has a TabIndex value of 0, the second has a TabIndex of 1, and so on. When you change a control's tab order position, Visual Basic automatically renumbers the tab order positions of the other controls to reflect insertions and deletions. For example, if you make Command1 first in the tab order, the TabIndex values for the other controls are automatically adjusted upward, as shown in the following table.

Control	TabIndex before it is changed	TabIndex after it is changed
Text1	0	1
Text2	1	2
Command1	2	0

The highest TabIndex setting is always one less than the number of controls in the tab order (because numbering starts at 0). Even if you set the TabIndex property to a number higher than the number of controls, Visual Basic converts the value back to the number of controls minus 1.

Note Controls that cannot get the focus, as well as disabled and invisible controls, don't have a TabIndex property and are not included in the tab order. As a user presses the TAB key, these controls are skipped.

Removing a Control from the Tab Order

Usually, pressing TAB at run time selects each control in the tab order. You can remove a control from the tab order by setting its TabStop property to False (0).

A control whose TabStop property has been set to False still maintains its position in the actual tab order, even though the control is skipped when you cycle through the controls with the TAB key.

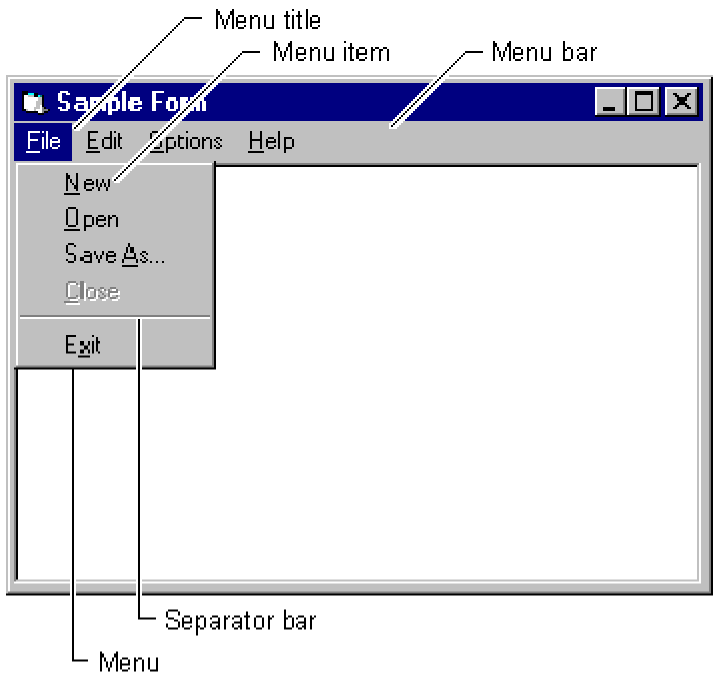
Note An option button group has a single tab stop. The selected button (that is, the button with its Value set to True) has its TabStop property automatically set to True, while the other buttons have their TabStop property set to False.

Menu Basics

If you want your application to provide a set of commands to users, menus offer a convenient and consistent way to group commands and an easy way for users to access them.

Figure 3.21 Illustrates the elements of a menu interface on an untitled form.

Figure 3.21 The elements of a menu interface on a Visual Basic form



The *menu bar* appears immediately below the *title bar* on the form and contains one or more *menu titles*. When you click a menu title (such as File), a menu containing a list of menu items drops down. Menu items can include commands (such as New and Exit), separator bars, and submenu titles. Each menu item the user sees corresponds to a menu control you define in the Menu Editor (described later in this chapter).

To make your application easier to use, you should group menu items according to their function. In Figure 3.21, for example, the file-related commands New, Open, and Save As… are all found on the File menu.

Some menu items perform an action directly; for example, the Exit menu item on the File menu closes the application. Other menu items display a *dialog box* — a window that requires the user to supply information needed by the application to perform the action. These menu items should be followed by an ellipsis (…). For example, when you choose Save As… from the File menu, the Save File As dialog box appears.

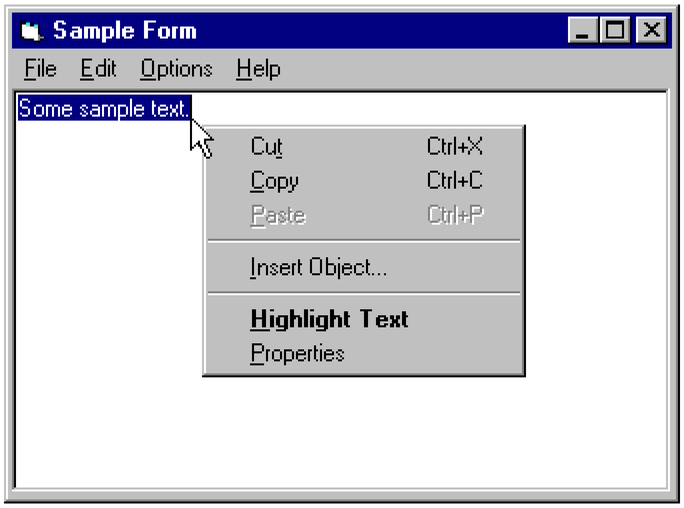
A menu control is an object; like other objects it has properties that can be used to define its appearance and behavior. You can set the Caption property, the Enabled and Visible properties, the Checked property, and others at design time or at run time. Menu controls contain only one event, the Click event, which is invoked when the menu control is selected with the mouse or using the keyboard.

For More Information For additional information on menu controls, see "Creating Menus with the Menu Editor" in "Creating a User Interface."

Pop-up Menus

A *pop-up menu* is a floating menu that is displayed over a form, independent of the menu bar, as shown in Figure 3.22. The items displayed on the pop-up menu depend on the location of the pointer when the right mouse button is pressed; therefore, pop-up menus are also called *context menus*. (In Windows 95 or later, you activate context menus by clicking the right mouse button.) You should use pop-up menus to provide an efficient method for accessing common, contextual commands. For example, if you click a text box with the right mouse button, a contextual menu would appear, as shown in Figure 3.22.

Figure 3.22 A pop-up menu



Any menu that has at least one menu item can be displayed at run time as a pop-up menu. To display a pop-up menu, use the PopupMenu method.

For More Information For additional information on creating pop-up menus, see "Creating Menus" in "Creating a User Interface."

Using the Menu Editor

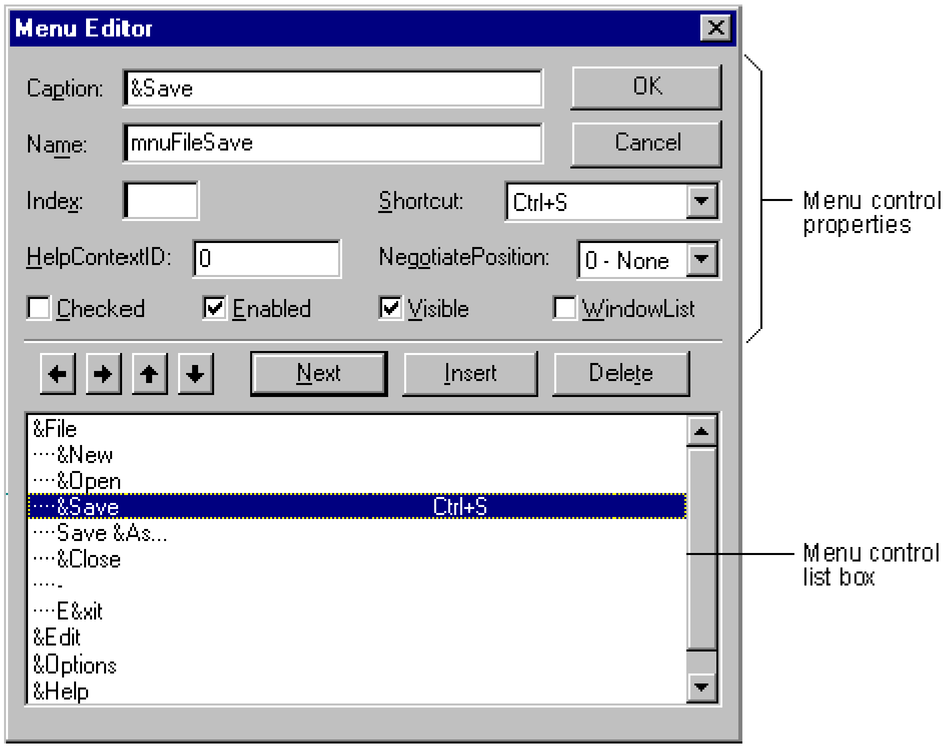
With the Menu Editor, you can add new commands to existing menus, replace existing menu commands with your own commands, create new menus and menu bars, and change and delete existing menus and menu bars. The main advantage of the Menu Editor is its ease of use. You can customize menus in a completely interactive manner that involves very little programming.

To display the Menu Editor

- From the **Tools** menu, choose **Menu Editor**.

This opens the Menu Editor, shown in Figure 3.23

Figure 3.23 The Menu Editor



Caption: &Save
Name: mnuFileSave
Index:
Shortcut: Ctrl+S
HelpContextID: 0
NegotiatePosition: 0 - None
☐ Checked ☑ Enabled ☑ Visible ☐ WindowList

```
&File
····&New
····&Open
····&Save                  Ctrl+S
····Save &As...
····&Close
····-
····E&xit
&Edit
&Options
&Help
```

While most menu control properties can be set using the Menu Editor; all menu properties are also available in the Properties window. You would normally create a menu in the Menu Editor; however, to quickly change a single property, you could use the Properties window.

For More Information For additional information on creating menus and using the Menu Editor, see "Creating Menus" in "Creating a User Interface."

Send feedback to MSDN. Look here for MSDN Online resources.

Visual Basic Concepts

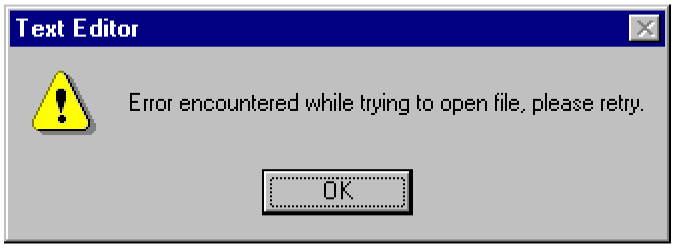
Prompting the User with Dialog Boxes

In Windows-based applications, dialog boxes are used to prompt the user for data needed by the application to continue or to display information to the user. Dialog boxes are a specialized type of form object that can be created in one of three ways:

- *Predefined* dialog boxes can be created from code using the MsgBox or InputBox functions.
- *Customized* dialog boxes can be created using a standard form or by customizing an existing dialog box.
- *Standard* dialog boxes, such as Print and File Open, can be created using the common dialog control.

Figure 3.24 shows an example of a predefined dialog box created using the MsgBox function.

Figure 3.24 A predefined dialog box



Error encountered while trying to open file, please retry.

This dialog is displayed when you invoke the MsgBox function in code. The code for displaying the dialog box shown in Figure 3.24 looks like this:

```
MsgBox "Error encountered while trying to open file," & vbCrLf & "please retry.", vbExclam
```

You supply three pieces of information, or arguments, to the MsgBox function: the message text, a constant (numeric value) to determine the style of the dialog box, and a title. Styles are available with various combinations of buttons and icons to make creating dialog boxes easy.

Because most dialog boxes require user interaction, they are usually displayed as modal dialog boxes. A *modal* dialog box must be closed (hidden or unloaded) before you can continue working with the rest of the application. For example, a dialog box is modal if it requires you to click OK or Cancel before you can switch to another form or dialog box.

Modeless dialog boxes let you shift the focus between the dialog box and another form without having to close the dialog box. You can continue to work elsewhere in the current application while the dialog box is displayed. Modeless dialog boxes are rare; you will usually display a dialog because a response is needed before the application can continue. From the Edit menu, the Find dialog box in Visual Basic is an example of a modeless dialog box. Use modeless dialog boxes to display frequently used commands or information.

For More Information For additional information on creating dialog boxes, see "Creating a User Interface."
