Message Box, Forms and Menu

UNIT III

Dr. A. Malathi M.Sc.,M.Phil.,Ph.D.,M.Sc(Psych)
Assistant Professor
PG and Research Department of Computer Science
Government Arts College
Coimbatore - 641018

UNIT III

UNIT III: Timer Control, Scroll Bar, Message Box (), Input Box (), Functions, MDI Forms, Menus and Dialog Boxes: Building Drop – Down Menus, Sub Menus - Pop - Up Menus –Dialog Boxes – Debugging and Executing A Projects –Error Handling – Convert –VB Project To Exe File – Procedures –Scope- Optional Arguments.

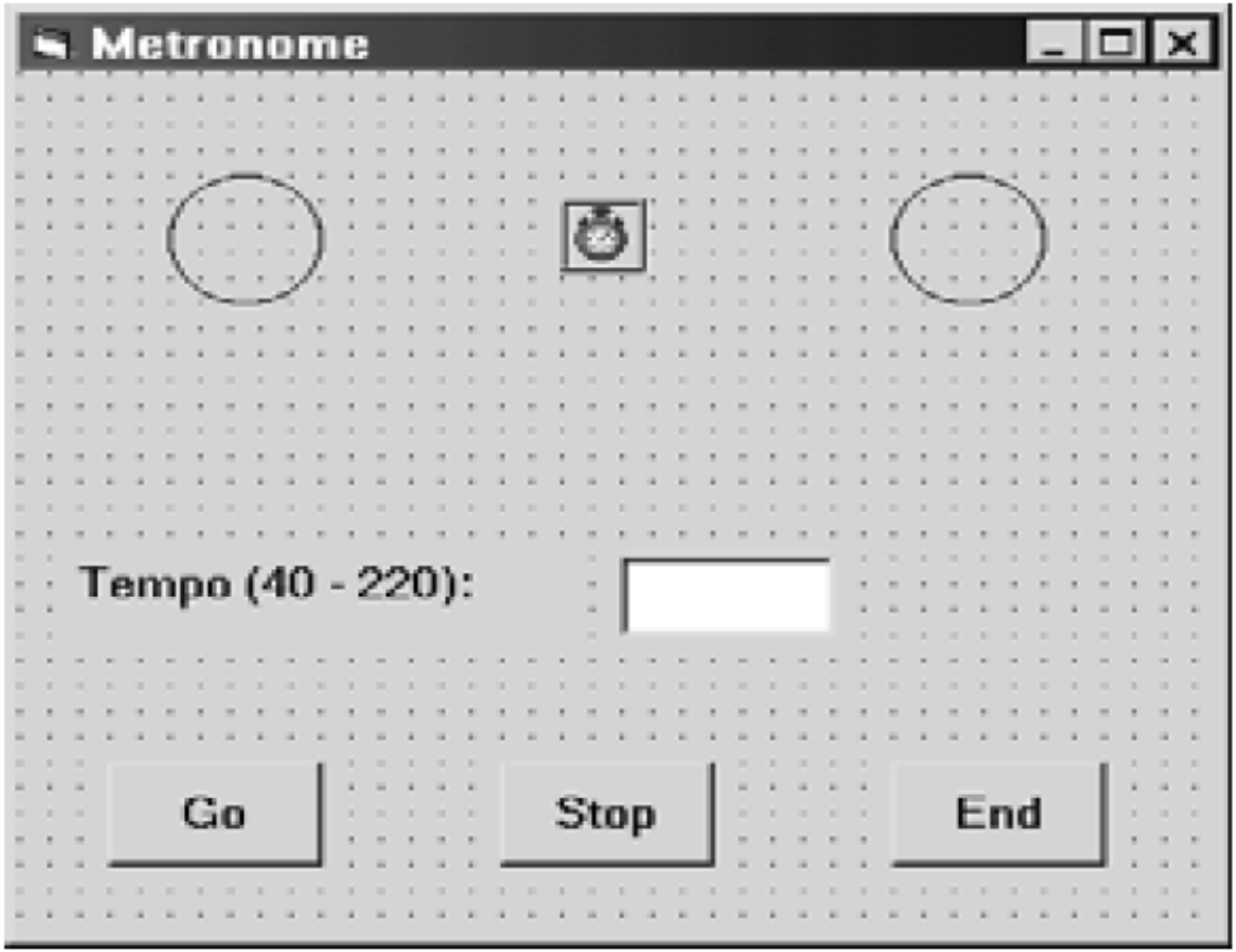
TIMER CONTROL

- ✓ Timed events, such as a digital clock or a stopwatch, make use of the timer control.
- ✓ The timer is placed in the Form Design Window at design time
- ✓ Its location and appearance are unimportant.
- ✓ primary importance is the Interval property. This property can be assigned an integer value ranging from 0 to 65,535. A zero value disables the timer. Positive values represent the number of milliseconds between timed events. Thus, a value of 1 represents an interval of one millisecond (one thousandth of a second); 1000 represents a one-second interval; and 60,000 represents one-minute interval.
- ✓ The Enabled property must be assigned a value of True in order to activate the timer.

Example

Object	*Property*	*Value*
Form1	Caption	"Metronome"
Shape1	Shape	3 (Circle)
	FillColor	Red
	FillStyle	1 (Transparent – default value)
Shape2	Shape	3 (Circle)
	FillColor	Red
	FillStyle	1 (Transparent – default value)
Timer	Enabled	False
Label1	Caption	"Tempo (40-220):"
	Font	MS Sans Serif, 10-point
Text1	Caption	(none)
	Font	MS Sans Serif, 10-point
Command1	Caption	"Go"
	Font	MS Sans Serif, 10-point
Command2	Caption	"Stop"
	Font	MS Sans Serif, 10-point
Command3	Caption	"End"
	Font	MS Sans Serif, 10-point

Form design



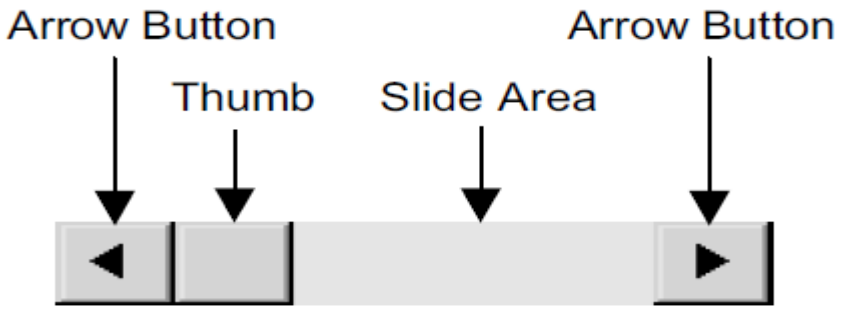
coding

```
Private Sub Timer1_Timer()
    'Beep
    If (Shape1.FillStyle = 0) Then   'left circle is gray – change to red
        Shape1.FillStyle = 1
        Shape2.FillStyle = 0
    Else                             'left circle is red – change to gray
        Shape1.FillStyle = 0
        Shape2.FillStyle = 1
    End If
End Sub

Private Sub Command1_Click()
    Dim Tempo As Single

    Tempo = Val(Text1.Text)
    If (Tempo < 40 Or Tempo > 220) Then    'Tempo out of range
        Beep
        Text1.Text = ""
        MsgBox ("Tempo out of Range - Please Try Again")
        Exit Sub
    End If

Private Sub Command2_Click()
    Text1.Text = ""
    Timer1.Enabled = False
End Sub

Private Sub Command3_Click()
    End
End Sub
```

HORIZONTAL SCROLL BAR

Scroll Bar

- ✓ Scroll bars can be used to view a large document by moving the visible window (scrolling) vertically or horizontally. They can also be used to select a particular value within a specified range, or to select a specific item from a list.



- ✓ important properties associated with scroll bars are Min, Max, SmallChange and LargeChange. Min and Max represent integer values corresponding to the minimum and maximum thumb locations within the slide area. The defaults are Min = 0 and Max = 32767.
- ✓ SmallChange and LargeChange indicate the size of the incremental movements when you click on the arrow buttons or the empty slide area, respectively.

BUILDING DROP-DOWN MENUS

- *Drop-down menus* represent another important class of components in the user interface. To create a drop-down menu, click on the Menu Editor button in the Toolbar or select Menu Editor from the Tools menu. Note

The check boxes labeled Enabled and Visible should be selected, as shown in the figure.


Menu Editor
Caption:
OK
Name:
Cancel
Index:
Shortcut:
(None)
HelpContextID:
0
NegotiatePosition:
0 - None
Checked
Enabled
Visible
WindowList
Next
Insert
Delete

Enter identifiers for the Caption and Name for each menu item. (The Caption is actually the screen name of the item, as it appears in the Menu Bar or within the drop-down menu. The Name is used only in Visual Basic code – it is not displayed when the application is running.) The Caption will appear in the large area at the bottom of the Menu Editor as well as within the Caption field. You may either press the Enter key or click on the Next button after the information has been entered for each menu item.

All of the menu components must be entered, in the following order:

1. The first menu heading (i.e., the screen name for the first menu, which appears in the menu bar).
2. The corresponding menu items for the first menu.
3. The second menu heading.
4. The corresponding menu items for the second menu.

The menu headings must be flush left within each line. Items that appear *within* each menu must be indented one level, as indicated by four ellipses preceding each item. The indentation is accomplished using the right-arrow button. Click once to indent one level (four ellipses). The opposite action, i.e., moving an indented item to the left, is accomplished with the left-arrow button.

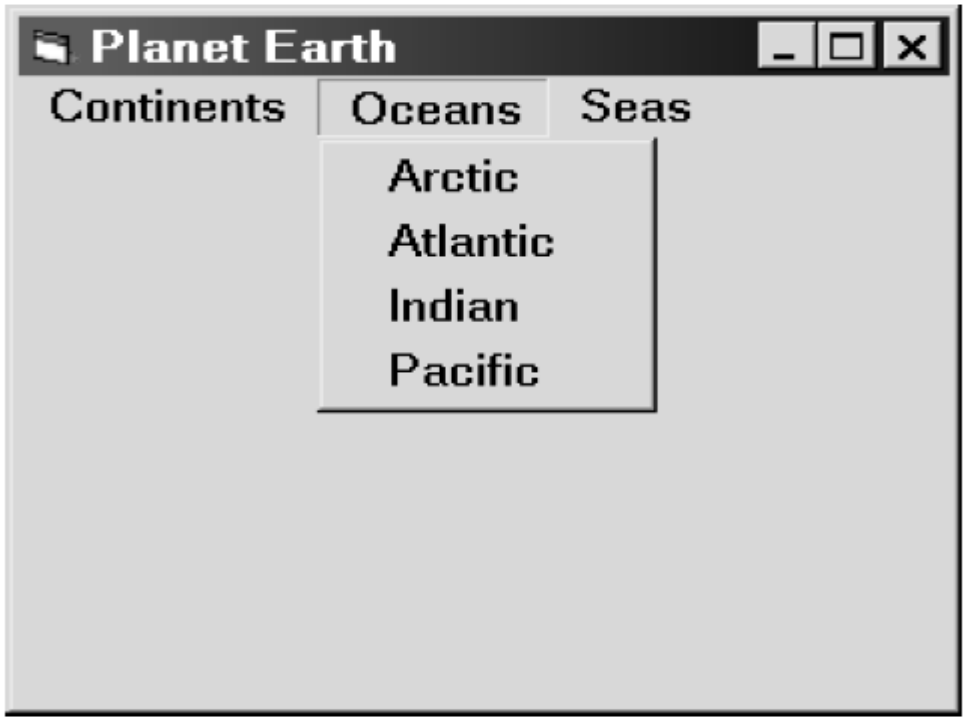
s

The relative ordering of each menu component can be altered using the up- and down-arrow buttons.

Example

The complete list of menu items (captions and names) is shown below

Caption	*Name*
Continents	mnuContinents
....Africa	mnuAfrica
....Antarctica	mnuAntarctica
....Asia	mnuAsia
....Australia	mnuAustralia
....Europe	mnuEurope
....North America	mnuNorthAmerica
....South America	mnuSouthAmerica
Oceans	mnuOceans
....Arctic	mnuArctic
....Atlantic	mnuAtlantic
....Indian	mnuIndian
....Pacific	mnuPacific



ACCESSING A MENU FROM THE KEYBOARD

A keyboard *access character* can be defined for each menu item. This allows the user to view a drop-down menu by pressing Alt and the access key for the menu heading, rather than clicking on the menu heading. In addition, once the drop-down menu is shown, the user may select a menu item by pressing its access key (without Alt) rather than clicking on the menu item. To define an access character, use the Menu Editor to place an ampersand (&) in front of the desired character within each menu item caption (i.e., within each screen name). The access character will then be underlined when the associated menu item is shown. Note that *a drop-down menu must actually be visible on the screen for its access characters to be active*.

In addition to access characters, we can also define *keyboard shortcuts* for some or all of the menu items within a drop-down menu. A keyboard shortcut is typically a function key, or a Ctrl-key combination or a Shift key combination. Keyboard shortcuts are selected directly from the Shortcut field within the Menu Editor.

The menu items have been modified to add access characters, as shown below (note the added ampersands).

&Continents

....&Africa

....An&tarctica

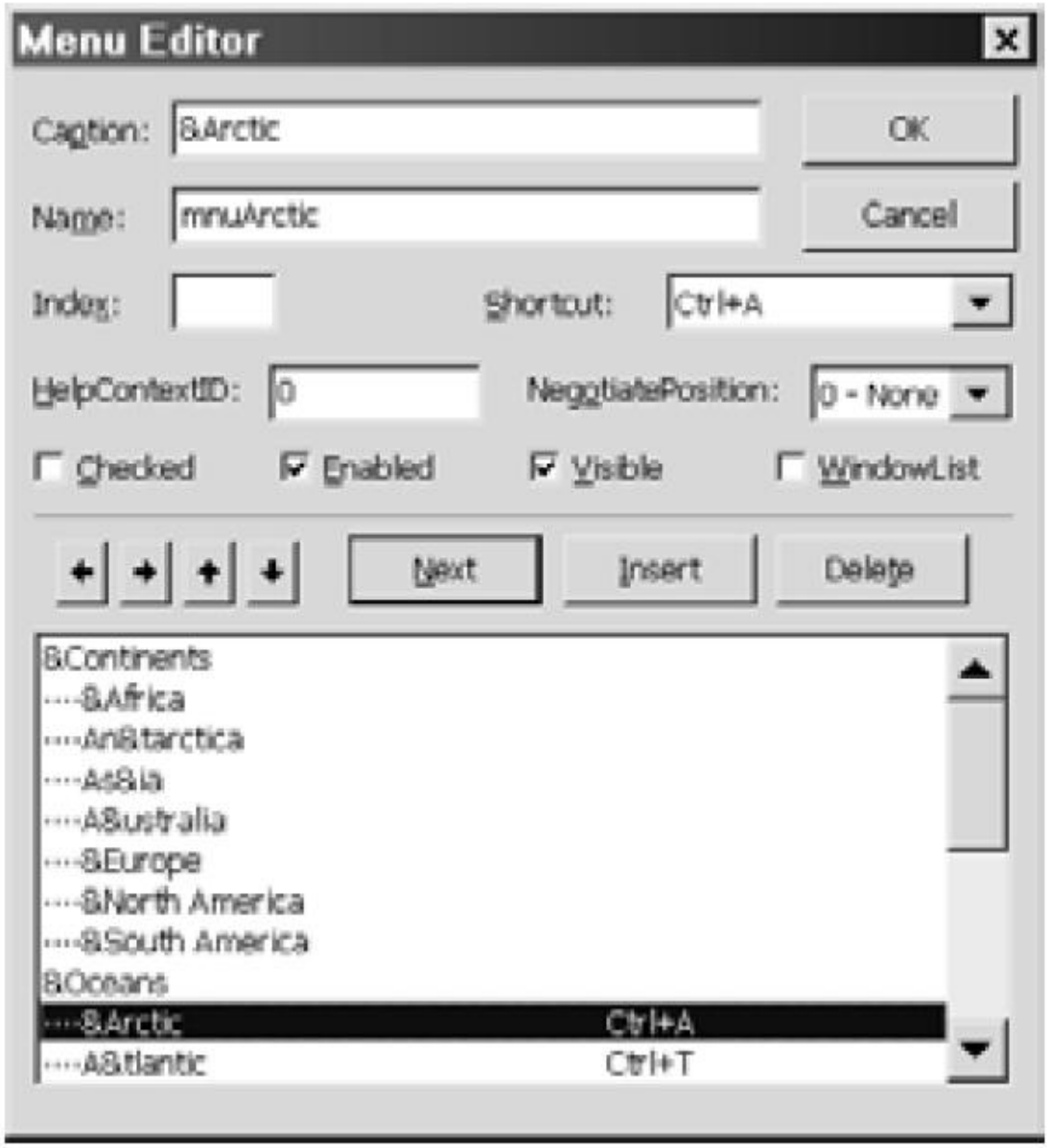
....As&ia

....A&ustralia

....&Europe

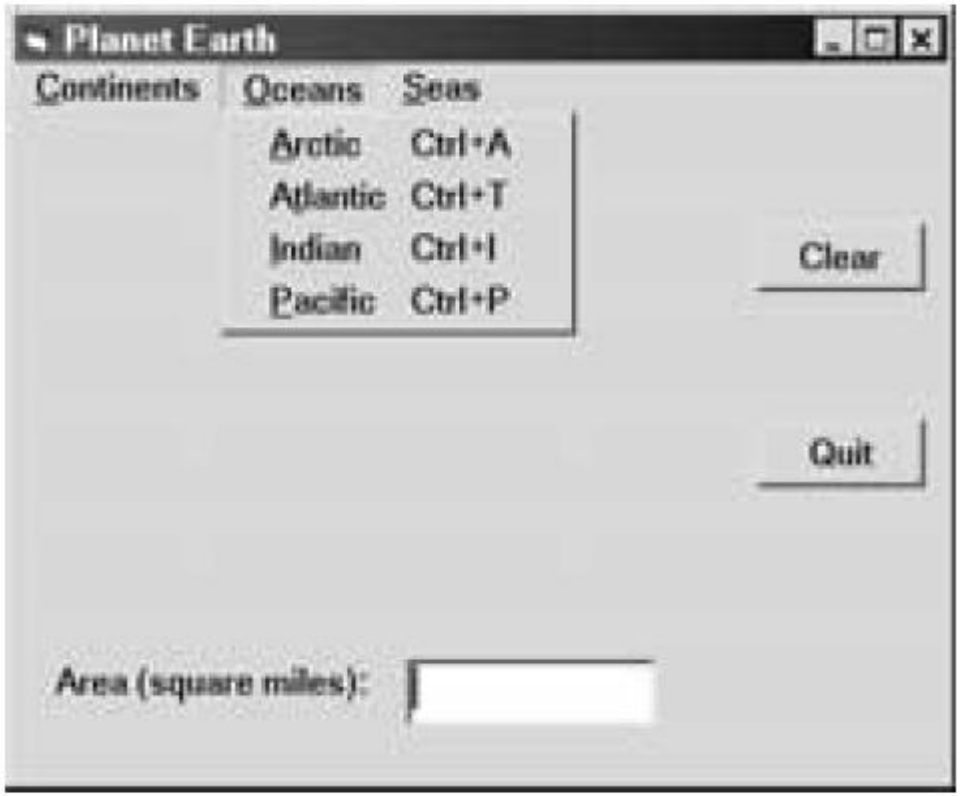
....&North America

....&South America



the menu editor, with Arctic as the active menu item listed under Oceans. Note that the key combination Ctrl+A has been selected as the keyboard shortcut for this menu item.

Thus, the area of the Arctic ocean can be displayed by clicking on Oceans and then Arctic, by pressing Alt-O followed by A, or by pressing Ctrl-A directly from the main window.



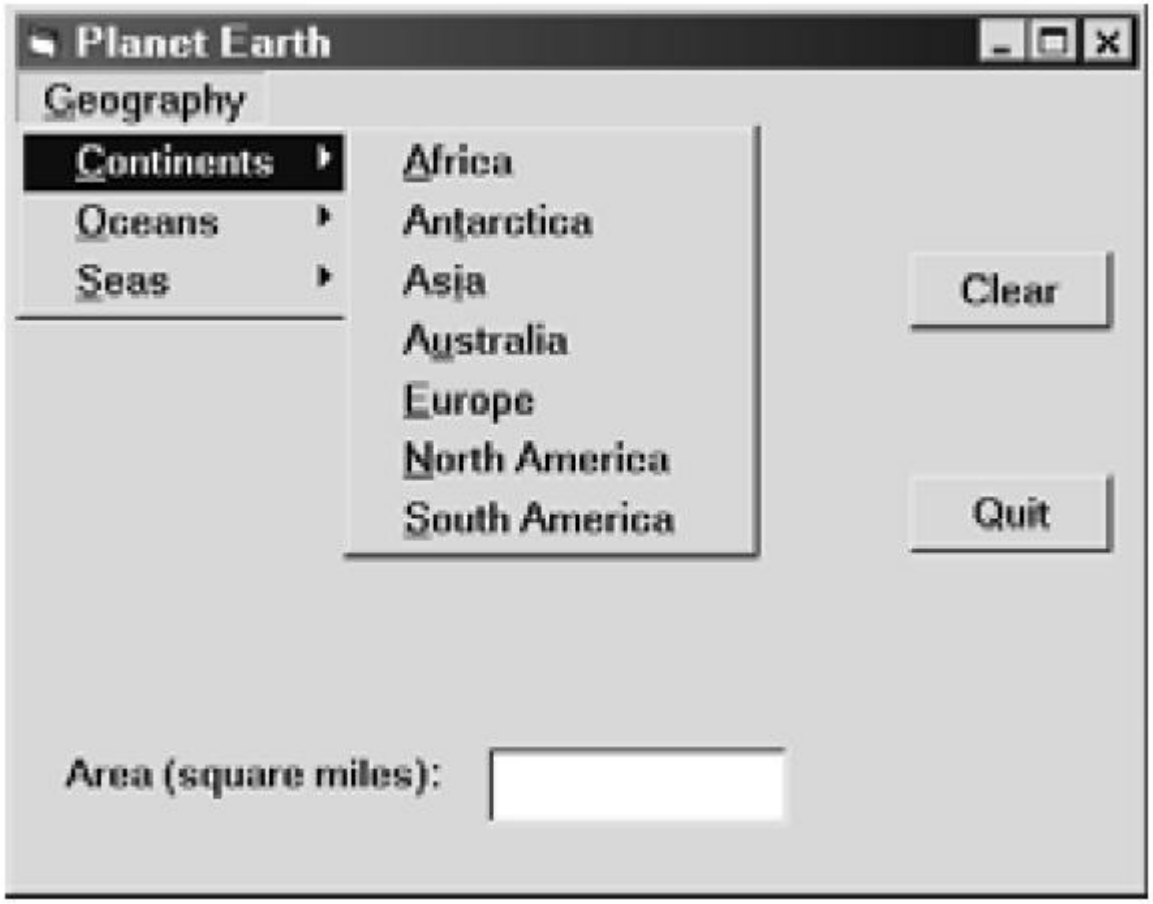
MENU ENHANCEMENTS

- The menu editor includes other features that permit various menu item enhancements. For example, a checkmark can be assigned to a menu item, indicating the on-off status of the menu item. Selecting the box labeled Checked will cause the menu item to be checked initially.
- Its status can then be changed (i.e., the check mark can be removed and later displayed) under program control when the program is executing.

SUBMENUS

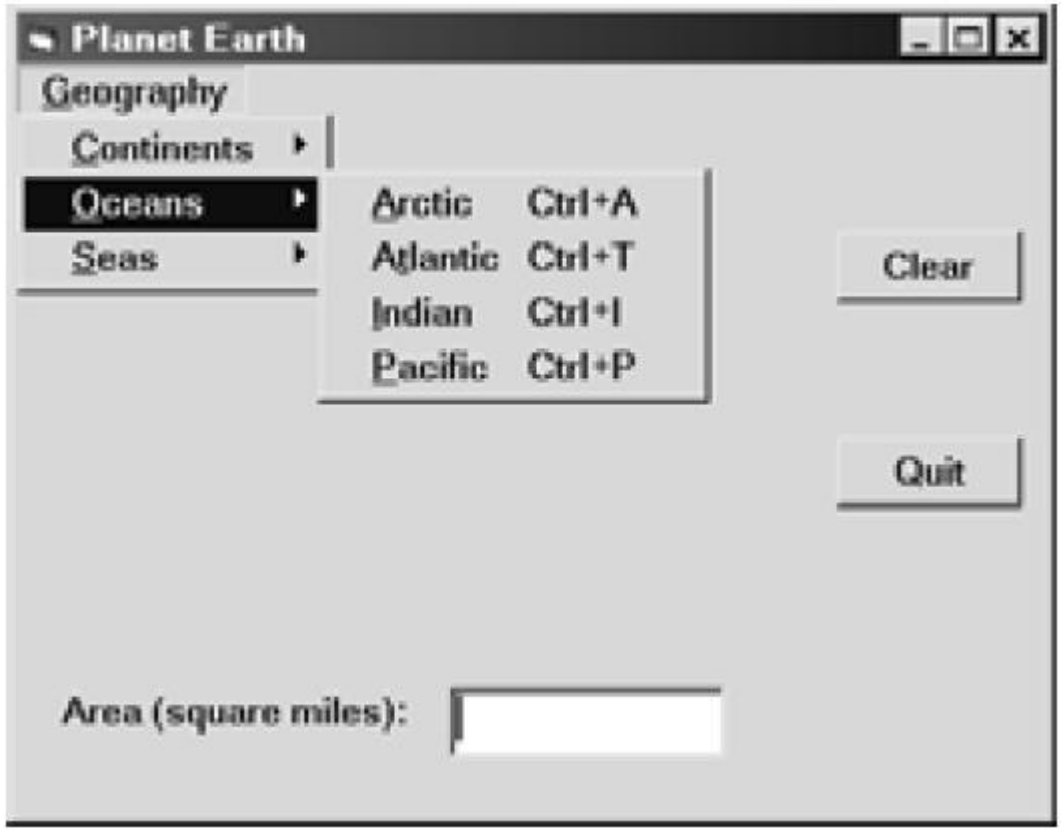
- ✓ A menu item may have a *submenu* associated with it. Placing the mouse over the menu item (or pressing the access character, keyboard shortcut, etc.) will cause the corresponding submenu to be displayed adjacent to the parent menu item.
- ✓ The use of submenus allows menu selections to be arranged in a logical, hierarchical manner.

- The below given menu will contain three menu items, Continents, Oceans and Seas. Each of these menu items will have its own submenu.



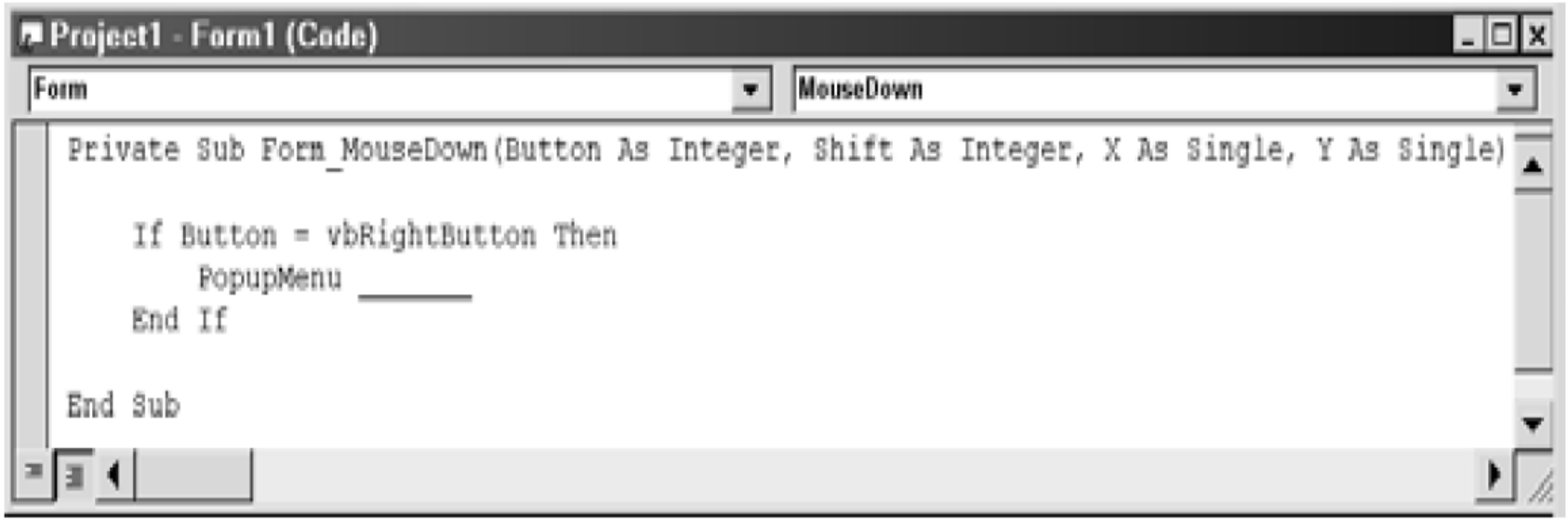
The list of menu items within the Menu Editor, adding the overall heading Geography at the top of the list, and then indenting all of the remaining menu items by one level. The modified list will appear as follows:

&Geography
....&Continents
.....&Africa
.....An&tarctica
.....As&ia
.....A&ustralia
.....&Europe
.....&North America
.....&South America
....&Oceans
.....&Arctic
.....A&tlantic
.....&Indian
.....&Pacific



POP-UP MENUS

- ✓ A pop-up menu can appear anywhere within a form, usually in response to clicking the right mouse button.
- ✓ the upper left corner of the pop-up menu appears at the location of the mouse click, though the position of the pop-up menu can be altered by specifying some additional parameters.
- ✓ A pop-up menu is created via the Menu Editor in the same manner as a drop-down menu, except that the main menu item is not visible (i.e., the Visible feature is unchecked).
- ✓ An event procedure must then be entered into the Code Editor so that the pop-up menu appears in response to the mouse click.
- ✓ All of the components of this event procedure have a predefined meaning and must be entered as shown. (The undefined underscore, which represents the caption for the first pop-down menu item, is supplied by the programmer.) Note that the first and last lines are generated automatically by the Code Editor, provided the correct object name (Form) is selected in the upper left portion of the Code Editor, and the correct action (MouseDown) is selected in the upper right.



- ✓ The action specified by each pop-up menu item must be entered into the Code Editor as a separate event procedure, as before. Thus, one event procedure is required to display the pop-up menu, and an additional event procedure is required for each of the various actions taken in response to the pop-up menu selections.

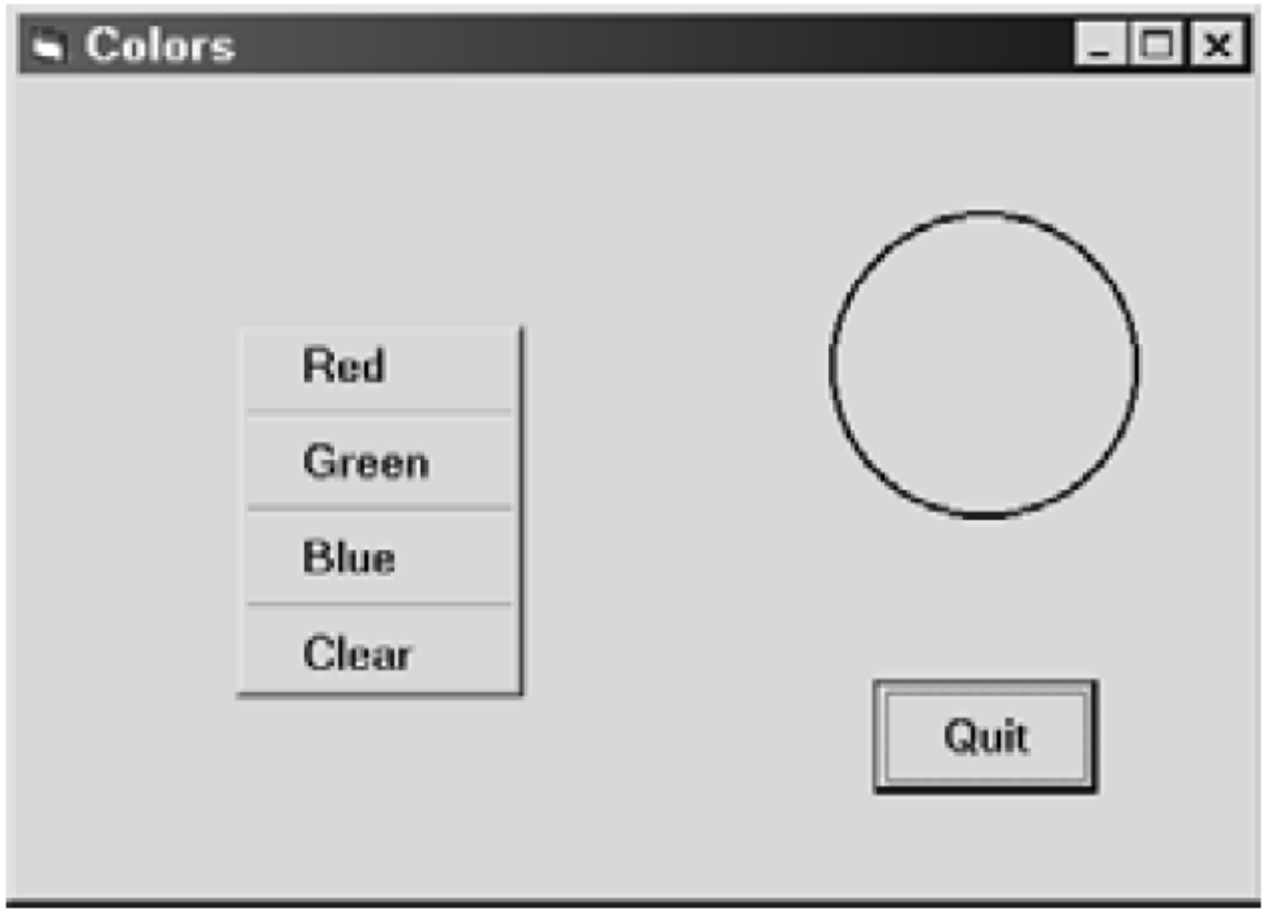
Figure shows the Menu Editor, with the entries required to change the color within the circle. Notice the caption (Colors) and the name (mnuColor) assigned to the first menu item. Also, note the use of separators between the menu items.

In order to display the menu and bring about the desired color changes in response to the menu selections, we must add the following event procedures via the Code Editor Window.

Example coding

```
Private Sub Form_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
 If Button = vbRightButton Then
            PopupMenu mnuColor
 End If
 End Sub
 Private Sub RedColor_Click()
 Shape1.FillColor = vbRed
 End Sub
 Private Sub GreenColor_Click()
 Shape1.FillColor = vbGreen
 End Sub
 Private Sub BlueColor_Click()
 Shape1.FillColor = vbBlue
 End Sub
```

Clicking the right mouse button then causes the pop-up menu to appear,



DIALOG BOXES

A *dialog box* is used to exchange information between the program and the user. Dialog boxes typically contain common controls (e.g., labels, text boxes, option buttons, check boxes, and command buttons) to enter or display information.

A "secondary" form (e.g., a dialog box) can be *added* to an active project via the Load command; i.e.,

Load *form*

For example, the command

Load Form2

will cause the form named Form2 to be loaded into the currently active project.

Similarly, a form can be *removed* from an active project, thus freeing up memory, via the Unload command;

i.e.,

Unload *form*

For example,

Unload Form2

Thus, the form named Form2 will be unloaded (removed) from the currently active project. As a result, references to the object named Form2 will no longer be recognized within the currently active project.

Loading a form into an active project does not in itself cause the form to be visible. To make the form visible, we use the Show method; i.e.,

form.Show

(Recall that a *method* is similar to a property. Whereas properties represent values associated with objects,however, methods carry out actions on objects.) For example,

Form2.Show

This causes the form named Form2 to become visible within the currently active project. Moreover, Form2 will be the currently active form, and it will be displayed on top of any other visible forms. If the form.Show method is followed by a 1; e.g.,

Form2.Show 1

the new form will be displayed as a *modal* form. That is, the form will remain in place, preventing the activation of any other forms, until the user disposes of the form by accepting it (e.g., by clicking OK), or rejecting it (e.g., by clicking Cancel).

The Hide method is directly analogous but opposite to the Show method. Thus, the command

Form2.Hide

causes Form2 to no longer be visible within the currently active project. This command does *not* cause Form2 to be *unloaded* from the project. Recall that we refer to a property (or method) associated with an object in a single-form project as

object name.property

For example,

Text1.Text

When working with multiform projects, however, it is often necessary to refer to a property (or method) of an object in a different form. To do so, we precede the object name with the form name; i.e.,

form name.object name.property

For example,

Form2.Text1.Text

Of course, the placement of these references is determined by the program logic.

THE MsgBox FUNCTION

- MsgBox function is actually a type of dialog box which displays a given output string and one or more command buttons (e.g., OK), and returns a positive integer whose value depends on the action taken by the user.

the function reference may be written as

integer variable = MsgBox(*string, integer, title*)

The value of the *integer* argument (default 0) defines the command buttons that appear within the dialog box. Also, *title* represents a string that will appear in the message box's title bar. It's default value (if not included as an explicit argument) will be the project name.

Integer Argument	*Resulting Command Buttons*
0	OK
1	OK, Cancel
2	Abort, Retry, Ignore
3	Yes, No, Cancel
4	Yes, No
5	Retry, Cancel

The value returned by the MsgBox function will depend upon the particular command button selected by the user during program execution. The possible values are summarized below.

Command Button	*Return Value*
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

THE InputBox FUNCTION

- The InputBox function is will automatically include a string prompting the user for input, and a text box where the user can enter an input string. It will also include two command buttons – OK and Cancel. Figure shows a typical input box with a prompt and a blank text box, awaiting user input.

In general terms, the function reference may be written as

string variable = Input Box(*prompt, title, default*)

- ✓ The first argument (*prompt*) represents a string that appears within the dialog box as a prompt for input.
- ✓ The second argument (*title*) represents a string that will appear in the title bar. It's default value (if not included as an explicit argument) will be the project name.
- ✓ The last argument (*default*) represents a string appearing appearing initially in the input box's text box.

Debugging and executing a project

syntactic errors

syntactic errors (also called *compilation errors*) occur when Visual Basic commands are written improperly.

```
Text6.Text = 0.01 * Val(Text1.Text) + 0.05 * Val(Text2.Text) + 0.1 * Val(Text3.Text) +
                                    0.25 * Val(Text4.Text) + 0.5 * Val(Text5.Text)
```

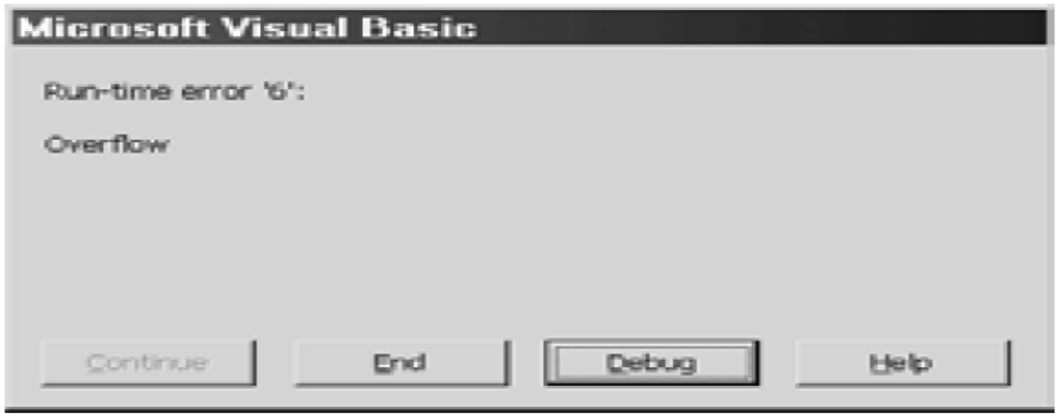
Now suppose that the right parenthesis at the end of the command had inadvertently been omitted; i.e.,

```
Text6.Text = 0.01 * Val(Text1.Text) + 0.05 * Val(Text2.Text) + 0.1 * Val(Text3.Text) +
                                    0.25 * Val(Text4.Text) + 0.5 * Val(Text5.Text
```

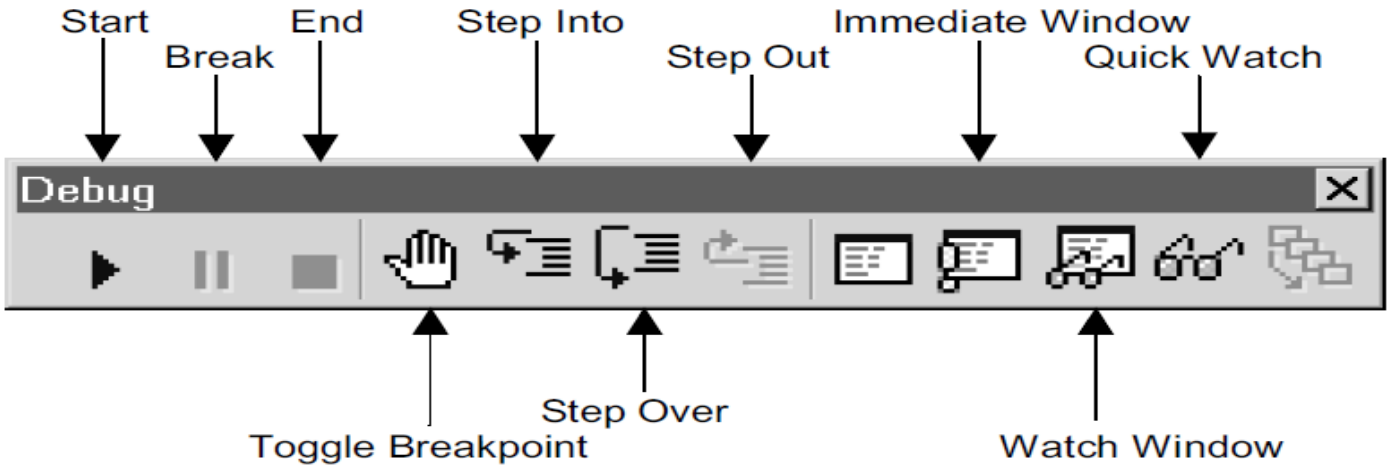
An attempt to run this program will result in a syntactic error message, as shown in figure:

LOGICAL ERRORS

Many execution errors are caused by faulty program logic (e.g., dividing by zero or attempting to take the square root of a negative number). A logical error results in a system crash, a message is produced indicating the reason for the crash.



Visual Basic allows you to access its debugging features three different ways: via the Debug menu on the main menu bar, through certain function keys, or through the Debug toolbar, as illustrated in Figure (TheDebug toolbar can be displayed by selecting Toolbars/Debug from the View menu.)



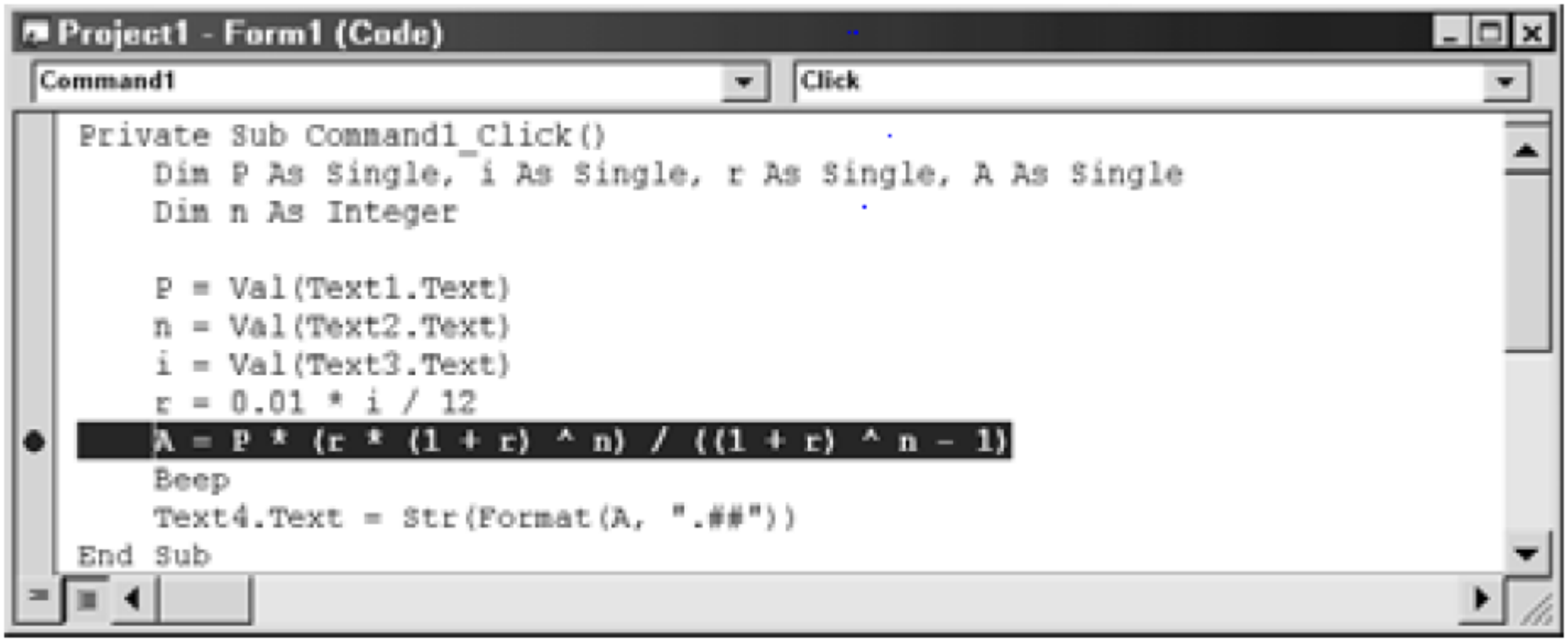
SETTING BREAKPOINTS

There are several different ways to set a breakpoint. The first step is to examine the program listing within the Code Editor Window and identify the statement where the break point will be located. Then select the statement, or simply click anywhere within the statement, and set the breakpoint in any of the following ways:

1. Select Toggle Breakpoint from the Debug menu.
2. Click on the Toggle Breakpoint button within the Debug toolbar.
3. On an Intel-based computer, press function key F9.

Once the breakpoint has been set, the statement will be clearly highlighted, as shown in Figure. Observe the dark circle to the left of the selected statement, in addition to the heavy highlighting.

Note that the breakpoint is set *ahead* of the selected statement. That is, the break in the program execution will occur just *before* the selected statement is executed. Also, note that the breakpoint is removed the same way it is set; i.e., by selecting Toggle Breakpoint from the Debug menu, by clicking on the Toggle Breakpoint button on the Debug toolbar, or by pressing function key F9. Thus, the breakpoint feature is referred to as a *toggle.*



If a program contains several different breakpoints, it may be convenient to remove all of them at once. To do so, simply select Clear All Breakpoints from the Debug menu, of press function keys Ctrl-Shift-F9 simultaneously.

USER-INDUCED ERRORS

User-induced errors are the result of mistakes made by the user when the program is executing (e.g., entering numbers that are out of range, or entering non numerical characters when a numerical value is expected). Errors of this type can usually be anticipated and "trapped" by one or more If-Then-Else blocks. However, it may be more convenient to use an *error handler* routine to trap the error and then take appropriate remedial action.

Procedures

- ✓ A *procedure* (including an event procedure) is a self-contained group of Visual Basic commands that can be accessed from a remote location within a Visual Basic program.
- ✓ Visual Basic supports three types of procedures – *Sub* procedures (sometimes referred to simply as *subroutines*), *Function* procedures (also called *functions*), and *Property* procedures.

SUB PROCEDURES (SUBROUTINES)

In its simplest form, a sub procedure is written as

Sub *procedure name* (*arguments*)

. . . . .

statements

. . . . .

End Sub

The *procedure name* must follow the same naming convention used with variables.

The list of *arguments* is optional. Arguments represent information that is transferred into the procedure from the calling statement. Each argument is written as a variable declaration; i.e.,

- *argument name* As *data type*

The data type can be omitted if the argument is a variant. Multiple arguments must be separated by commas. If arguments are not present, an empty pair of parentheses must appear in the Sub statement.

- A sub procedure can be accessed from elsewhere within the module via the Call statement. The Call statement is written
- Call *procedure name* (*arguments*)

The list of arguments in the Call statement must agree with the argument list in the procedure definition. The arguments must agree in *number*, in *order*, and in *data type*. However, the respective names may be different.

The required procedures (a sub procedure and two event procedures) are shown below.

```
Sub Smallest(a, b)
    Dim Min
    If (a < b) Then
        Min = a
        MsgBox "a is smaller (a = " & Str(Min) & ")"
   ElseIf (a > b) Then
        Min = b
         MsgBox "b is smaller (b = " & Str(Min) & ")"
   Else
        Min = a
        MsgBox "Both values are equal (a, b = " & Str(Min) & ")"
    End If
End Sub
```

```
Private Sub Command1_Click()
    Dim x As Variant, y As Variant
    x = Val(Text1.Text)
    y = Val(Text2.Text)
   Call Smallest(x, y)
End Sub
Private Sub Command2_Click()
    End
End Sub
```

When passing an argument by reference, the argument name may be preceded by the reserved word ByRef within the procedure definition; i.e.,

- ByRef *argument name* As *data type*

In order to pass an argument by value, the argument name within the procedure must be preceded by the reserved word ByVal; i.e.,

- ByVal *argument name* As *data type*

Here are the corresponding procedures.

```
Sub Smallest(ByVal a, ByVal b, ByRef c)
     If (a < b) Then
          c = a
    Else
         c = b
    End If
End Sub
Private Sub Command1_Click()
Dim x, y, z, min
    x = Val(Text1.Text)
    y = Val(Text2.Text)
    z = Val(Text3.Text)
   Call Smallest(x, y, min)
   Call Smallest(z, min, min)
   Text4.Text = Str(min)
End Sub
```

```
Private Sub Command2_Click()
    Text1.Text = ""
    Text2.Text = ""
    Text3.Text = ""
    Text4.Text = ""
End Sub
Private Sub Command3_Click()
   End
End Sub
```

EVENT PROCEDURES

Event procedures should be quite familiar by now, as we have been using them throughout this book. An event procedure is a special type of sub procedure. It is accessed by some specific action, such as clicking on an object, rather than by the Call statement or by referring to the procedure name.