SYBEX Sample Chapter

Mastering AutoCAD© VBA

Marion Cottingham

Chapter 7: Macro-izing Solid Areas

ISBN: 0-7821-2871-8

Macro-izing Solid Areas

AUTOCAD VBA

Chapter 7

This chapter introduces VBA techniques for working with solid areas. You'll learn how to add a circle to your drawing and fill it with color. Also covered are freeform shapes: how to draw them and calculate the area they cover from a VBA macro. The ComboBox control is introduced, and you'll see how to format points displayed in the drawing space. Island Detection styles are discussed. Input boxes are used to communicate and retrieve input from the user. The final project in this chapter creates a drawing of a range top by prompting the user for its outside dimensions and number of burners. Then the project calculates the size and positions for the burners and draws them, all from a macro.

This chapter covers the following topics:

- Drawing circles
- Drawing freeform shapes
- Calculating areas of shapes
- Filling objects using inner and outer loops
- Working with input boxes
- Drawing a range top from a macro

Drawing Circles

Drawing a circle requires calling the AddCircle method with two arguments—an array containing the coordinates defining its center, and a radius to define the circle's size. The AddCircle method creates a Circle object and adds it to the specified ModelSpace, PaperSpace, or Blocks collection. The Circle object is accessed by a variable that's set up to contain a reference to it.

You can specify the values required for the center and radius in the following three ways:

- Assign values in code, which has the disadvantage of producing identical circles each time it is run (although sometimes this is exactly what is required). This is achieved in the same way that the endpoints of a line were assigned in the CreateLine macro from Listing 5.3 in Chapter 5.
- Create a UserForm and ask the user to enter values into text box controls. This requires the user to enter the three coordinates defining the center into text boxes in much the same way as for the Line Input UserForm from Chapter 6, except the three textboxes for the second endpoint are replaced by one text box to contain the radius.
- Use methods from the Utility object to prompt the user to enter values into the AutoCAD command line, or to mouse-click points in the Model Space. This technique is demonstrated in Exercise 7.1.

You can also use combinations of these methods.

Exercise 7.1: The DrawCircle Macro

Listing 7.1 uses the third technique mentioned above: putting the Utility object to work. It shows how a circle can be generated from a macro using the Utility object's methods to interact with the user. We prompt them to provide the center and radius in much the same way as for adding an Arc object in Chapter 6.

For this exercise, start a new project and place the macro into the ThisDrawing module so that it will appear in the Macros dialog box on your computer.

Listing 7.1: DrawCircle Macro

```
Public Sub DrawCircle()
    Dim CircleObject As AcadCircle
    Dim Center As Variant
    Dim Radius As Double
    'give instructions to user in the command line
```

```
6       With ThisDrawing.Utility
7           Center = .GetPoint(, "Click the position ↵
               for the center.")
8           Radius = .GetDistance(Center, "Enter the radius.")
9       End With
10   'create a Circle object and draw it in red
11       Set CircleObject = ThisDrawing.ModelSpace.AddCircle↵
            (Center, Radius)
12       CircleObject.Color = acRed
13       CircleObject.Update
14   End Sub
```

Analysis

- Line 1 starts the `DrawCircle` macro that prompts the user to input the center and radius of a circle, and then draws a red circle to the user specifications.
- Line 2 declares `CircleObject` as a variable that can be set up to refer to a `Circle` object.
- Line 3 declares `Center` as a Variant type. This is an excellent way of getting around the fact that normally the values of an array's elements must be assigned individually. The `Center` variable is assigned the value returned from the `GetPoint` method (line 7), which is a Variant type containing a three-element array of doubles that contain the x-, y-, and z-coordinates of the point clicked by the user. After this assignment, you can retrieve each coordinate's value separately from the `Center` variable just as if you had declared it as an array; for example, `X = Center(0)`.
- Line 4 declares the `Radius` variable, which is assigned the value returned by the `GetDistance` method of the Utility object (returned by the Utility property of ThisDrawing).
- Line 6 starts the `With` statement block and uses the Utility property to retrieve the Utility object from the ThisDrawing collection. The Utility object is used to set up the automatic qualifier for the items starting with a period character (`.`) inside the `With` statement block.
- Line 7 calls the `GetPoint` method of the Utility object to prompt the user to click on the position for the circle's center. The text passed as the second argument appears in the AutoCAD command line. This method returns the three coordinates of the point clicked, which are assigned to the `Center` variable.

- Line 8 calls the `GetDistance` method to prompt the user to enter the circle's radius. The `Center` is passed as the first argument to act as the base point. A rubber-band line anchored at this base point appears to follow the mouse cursor around as the user moves it.
- Line 9 ends the `With` statement block.
- Line 11 uses the `AddCircle` method to create a Circle object with the user's specifications and adds it to the ModelSpace collection. A reference to this new Circle object is assigned to the variable `CircleObject` that was declared in line 2 as an `AcadCircle` type.
- Line 12 assigns the value of the AutoCAD constant `acRed` to the `Color` property, which will be the color of the boundary of the Circle object.
- Line 13 uses the `Update` method to refresh the drawing space to draw the circle.

Figure 7.1 shows a circle generated using this macro. Notice how the Circle object's color property refers only to the color of its circumference and that no color fill has been performed.

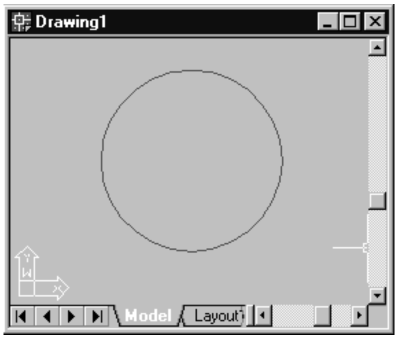


Figure 7.1 *Circle produced by the DrawCircle macro in Listing 7.1*

Drawing Filled Circles

In this section you'll learn how to fill a circle with color by employing a Hatch object that can fill any area with a solid fill or one of the industry-standard hatch patterns available. AutoCAD gives you a choice of over 50 such patterns. You can also use patterns from external libraries or define your own patterns based on the current linetype.

The AddHatch Method

A Hatch object is created and added to the ModelSpace collection using the AddHatch method. This method has three parameters:

PatternType The PatternType parameter specifies one of three AutoCAD constants:

- acHatchPatternTypePredefined, which uses the PatternName string to search for the pattern in AutoCAD's default pattern file `acad.pat`
- acHatchPatternTypeUserDefined, which uses the current linetype to define a new pattern
- acHatchPatternTypeCustomDefined, which uses the PatternName string to search for the pattern in `.pat` library files rather than the default `acad.pat` file

PatternName The PatternName argument is a string containing the name of the hatch pattern to be used. AutoCAD uses the PatternType setting to determine whether to look in the default `acad.pat` file for the named pattern, or to look in one of the other `.pat` files that contain custom-defined patterns.

Associativity The Associativity argument is set to True if the hatch is to be associated with the boundary that contains it, and False otherwise. When set to True, an associative hatch will change when its boundary is modified. On the other hand, a nonassociative hatch is independent of its boundary, so the hatch pattern will remain in place even after the original boundary is modified or removed.

Like all objects in VBA, the AddHatch method should always be in an assignment statement, using the `Set` statement to set to a variable the reference to the Hatch object being created.

Remember, the first statement after creating your Hatch object should be a call to the AppendOuterLoop method, to ensure that the outer boundary for the hatch pattern is closed. If any other operation is attempted, then AutoCAD cannot predict what will happen next.

EXERCISE 7.2: DRAWING A FILLED CIRCLE

For this exercise, start a new project and place the macro into ThisDrawing's Code window. That way it will appear in the Macros dialog box on your PC so that it can be run from the AutoCAD window.

Listing 7.2: DrawFilledCircle Macro

```
1   Sub DrawFilledCircle()
2       'HatchObject will reference the new Hatch object
3       Dim HatchObject As AcadHatch
4       'OuterCircle becomes the circle defining the outer ↵
           loop boundary
5       Dim OuterCircle(0) As AcadCircle
6       Dim Center As Variant
7       Dim Radius As Double
8       'get Center position and radius from the user
9       With ThisDrawing.Utility
10          Center = .GetPoint(, "Click the position ↵
               for the center.")
11          Radius = .GetDistance(Center, "Enter the radius.")
12      End With
13      'create Circle object and assign reference ↵
           to OuterCircle
14      Set OuterCircle(0) = ThisDrawing.ModelSpace.AddCircle↵
           (Center, Radius)
15      OuterCircle(0).Color = acYellow
16      OuterCircle(0).Update
17      'create Hatch object and assign reference to HatchObject
18      Set HatchObject = ThisDrawing.ModelSpace.AddHatch↵
           (acHatchPatternTypePredefined, "SOLID", True)
19      'add the outer loop boundary to the Hatch object
20      HatchObject.AppendOuterLoop (OuterCircle)
21      HatchObject.Evaluate
22      HatchObject.Update
23  End Sub
```

Analysis

- Line 1 starts the `DrawFilledCircle` macro, which prompts the user in the command line to input the center and radius of a circle. These values are used to create the circle that forms the outer loop boundary when the circle is filled with solid color.
- Line 3 declares `HatchObject` as a variable capable of referencing a Hatch object.
- Line 5 declares `OuterCircle` as being a single-element array capable of containing a reference to a Circle object. Notice how it is declared with a dimension of

zero; this can also be written as `0 to 0`, and is interpreted as containing one element. `OuterCircle` needs to be an array because it is passed as an argument to the `AppendOuterLoop` method (line 20) that expects an array of one or more objects collectively defining the closed boundary that can be used for doing the filling.

- Line 6 declares `Center` as a Variant type so that it can be assigned the x-, y-, and z-coordinates returned by the `GetPoint` method in a single assignment statement, and thereafter can be treated like an array with an index value to access each coordinate separately.
- Lines 9 through 12 contain the `With` statement block that uses methods from the Utility object to prompt the user to enter the center and radius of the circle to be filled. The user can respond by entering values into the command-line window, or by checking points in the Model tab.
- Line 14 uses the `AddCircle` method to add a Circle object to the user's specifications to the ModelSpace collection, and assigns a reference to this new object to the first and only element in the `OuterCircle` array.
- Line 15 changes the color of the circle to yellow.
- Line 16 updates the screen to show the yellow circle.
- Line 18 calls the `AddHatch` method to create a Hatch object, and adds it to the ModelSpace collection. The first argument passed to this method is the AutoCAD constant `acHatchPatternTypePredefined`, to instruct the interpreter to search the default `acad.pat` file for the definition of the pattern named in the second argument as `SOLID`. The third argument passes the value `True` to the `Associativity` parameter so that the pattern is associated with the circular boundary.
- Line 20 uses the `AppendOuterLoop` method to set the outer-loop boundary for the fill. The one-element array `OuterCircle` that references the Circle object is passed as the argument to provide the boundary. Lines 18 and 20 should always be treated as an inseparable pair, because if line 20 were omitted, AutoCAD would enter an unpredictable state.
- Line 21 calls the `Evaluate` method that finds the point where the pattern definition lines intersect with the hatch boundary. When solid-fill hatch patterns have been requested, as is the case here, the `Evaluate` method divides the hatch area into triangles and fills each one with the fill color. If the boundary passed

to the `AppendOuterLoop` method (line 20) is not a single closed loop, this fact will be picked up by `Evaluate`, and the application will simply stop and give an error message.

- Line 22 calls the `Update` method of the Hatch object to draw the filled circle in the Model tab.
- Line 23 ends the `DrawFilledCircle` macro.

Figure 7.2 shows the result of running the DrawFilledCircle macro.

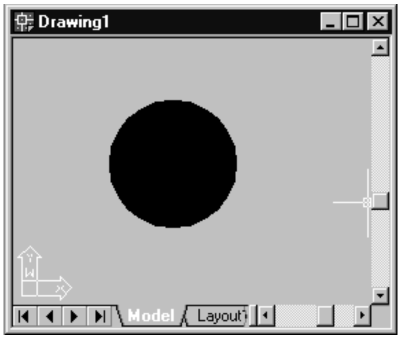


Figure 7.2 *Circle created and filled by the DrawFilledCircle macro (Listing 7.2)*

Circle of Bricks Application

When you have a specific effect in mind, filling a circle with a Hatch pattern may not give you the look you want. Suppose you want a circle of brick pavers, for example. Figure 7.3 shows the result of changing `SOLID` to `BRICK` at Line 18 of Listing 7.2.

EXERCISE 7.3: DRAW CIRCULAR PAVERS APPLICATION

Let's create an application that provides a much better visual solution than the circle filled with a BRICK hatch pattern. The following steps show you how it's done:

1. Start a new project and add a UserForm. Place a TextBox and accompanying Label, a Frame containing two OptionButtons, and two command buttons, as shown in Figure 7.4.
2. Change the object values in the Properties window to those shown in Table 7.1.

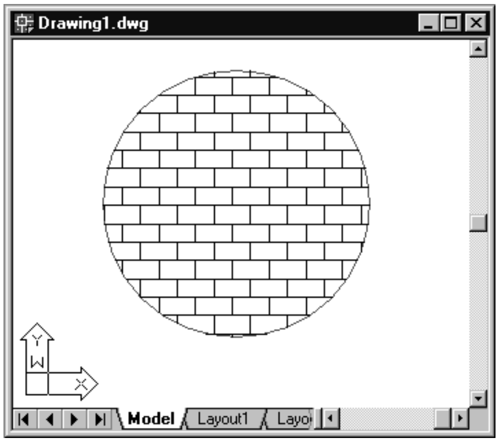
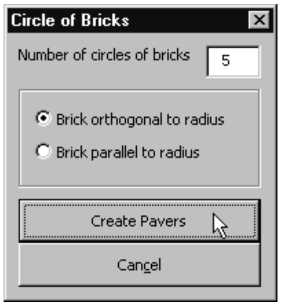


Figure 7.3 *Circle filled with BRICK hatch pattern instead of SOLID.*


Circle of Bricks

Number of circles of bricks 5

[✓ Brick orthogonal to radius]

Brick parallel to radius

Figure 7.4 *GUI for the Circle of Bricks application*

3. Enter the code shown in Listing 7.3 into the UserForm's Code window.

   The DrawCircularPavers procedure draws all the red circles and calls the DrawMortar procedure to draw the lines between bricks. Both procedures are private by default, so they won't be listed in the AutoCAD Macros dialog box.

Table 7.1 Property Values for Controls in Circle of Bricks Application

OLD NAME	NEW NAME	CAPTION
UserForm1	frmCircleOfBricks	Circle of Bricks
Label1		Number of circles of bricks
TextBox1	txtNumberOfCircles	
Frame1	—	Select the middle for drawing points
OptionButton1	optBrickOrthogonal	Brick orthogonal to radius
OptionButton2	optBrickParallel	Brick parallel to radius
CommandButton1	cmdCreatePavers	Create Pavers
CommandButton2	cmdCancel	Cancel

LISTING 7.3: DRAWCIRCULARPAVERS PROCEDURE

```
Sub DrawCircularPavers()
Dim BrickCircles() As AcadCircle
Dim Center As Variant, Radius As Double
Dim Counter As Integer
ReDim BrickCircles(txtNumberOfCircles)
'get Center position and radius from the user
With ThisDrawing.Utility
    Center = .GetPoint( ↵
       , "Click the position for the center.")
    Radius = .GetDistance(Center, "Enter the radius.")
End With
    'create Brick circle objects
For Counter = 0 To txtNumberOfCircles - 1
    Set BrickCircles(Counter) = ↵
       ThisDrawing.ModelSpace.AddCircle(Center, ↵
       Radius - Counter * Radius / txtNumberOfCircles)
    BrickCircles(Counter).Color = acRed
    BrickCircles(Counter).Update
    DrawMortar Center, Counter, Radius
Next
End Sub

Sub DrawMortar(Center As Variant, Counter As Integer, ↵
   Radius As Double)
    'declare variables for end points for line
```

```
22  Dim StartPoint(0 To 2) As Double, EndPoint(0 To 2) As Double
23  Dim Theta As Double, StepSize As Double
24  Static Adjust As Double
25  If frmCircleOfBricks.optBrickParallel = True Then
26      StepSize = 15 * Pi / 180
27  Else 'place brick orthogonal to radius
28      StepSize = 30 * Pi / 180
29      If Adjust = 0# Then
30          Adjust = 15 * Pi / 180
31      Else
32          Adjust = 0#
33      End If
34  End If
35  For Theta = 0 To 360 * Pi / 180 Step StepSize
36      StartPoint(0) = (Radius - Counter * Radius / ↵
           txtNumberOfCircles) * Cos(Theta + Adjust) + Center(0)
37      StartPoint(1) = (Radius - Counter * Radius / ↵
           txtNumberOfCircles) * Sin(Theta + Adjust) + Center(1)
38      EndPoint(0) = (Radius - (Counter + 1) * Radius / ↵
           txtNumberOfCircles) * Cos(Theta + Adjust) + Center(0)
39      EndPoint(1) = (Radius - (Counter + 1) * Radius / ↵
           txtNumberOfCircles) * Sin(Theta + Adjust) + Center(1)
40      StartPoint(2) = 0#: EndPoint(2) = 0#
41      With ThisDrawing.ModelSpace
42          .AddLine StartPoint, EndPoint
43          .Item(ModelSpace.Count - 1).Update
44      End With
45  Next
46  End Sub
```

Analysis

- Line 1 starts the `DrawCircularPavers()` procedure from the frmCircleOf-Bricks UserForm.
- Line 2 declares the `BrickCircles` as a dynamic array—you can tell from the empty parentheses. This array will contain the references to the circle objects.
- Line 3 declares the `Center` and `Radius` variables that are assigned values entered by the user in the AutoCAD window.
- Line 4 declares the `Counter` variable that is used as the loop counter in the `For` loop.

- Line 5 redimensions the BrickCircles dynamic array to make it large enough to contain the number of circles requested by the user.
- Lines 7 through 10 get the central point and radius of the circle from the user.
- Line 12 starts the For loop that uses Counter to access each circle on the BrickCircle array.
- Line 13 creates a Circle object. The Counter variable is used both as an index to the BrickCircles array and also as an ordinary variable to calculate the radius for the current circle. The circles are generated from largest to smallest, and references to them are stored in the array.
- Lines 14 and 15 set the color of the new circle and draw it on the screen.
- Line 16 calls the DrawMortar procedure (Line 20) to draw the lines between the bricks forming the current circle.
- Line 17 ends the For loop.
- Line 18 ends the DrawCircularPavers procedure.
- Line 20 starts the DrawMortar procedure.
- Line 22 declares the StartPoint and EndPoint arrays that are used to store the coordinates of the current line-between-bricks being drawn.
- Line 23 declares Theta, which is the angle at which the line-between-bricks will be drawn, and StepSize, which determines the number of bricks that make up the circle.
- Line 24 declares the Adjust variable to rotate the bricks by half a brick every second row. This variable is declared as Static so that it keeps its value between circles.
- Line 25 starts the outer If statement by testing to see if the "Brick parallel to radius" option button is selected.
- Line 26 sets the StepSize to 15 degrees multiplied by pi / 180, to ensure that StepSize is assigned radians. This accommodates bricks that are placed so that their length is parallel to the radius at that point.
- Line 27 starts the Else part of the If statement.
- Line 28 sets the StepSize variable to 30 degrees converted to radians. This accommodates bricks that are placed at right angles to the radius at that point.

- Lines 29 through 33 contain the inner `If` statement block that toggles the value of the `Adjust` variable between zero and half a brick length, to give the staggered effect shown in Figure 7.5.
- Line 34 ends the outer `If` statement block.
- Line 35 starts the `For` loop that uses `Theta` as the angle in radians between successive lines-between-bricks.
- Lines 36 through 40 assign values of coordinates to the start and endpoints of the current brick line. They use the `Sin` and `Cos` functions to calculate the x- and y-coordinates.
- Lines 41 through 44 contain the `With` statement block that creates a line and displays it on the screen.
- Lines 45 and 46 end the `For` loop and `DrawMortar` procedure, respectively.

In these next two steps, you will see how to write a macro to start your Circle of Bricks application, and how to code the Click event procedures for the two command buttons.

1. Choose Insert ➜ Module, and enter the following global constant declaration into Module1 with the DrawCircleOfBricks macro:

   ```
   Global Const Pi = 3.14159

   Sub DrawCircleOfBricks()
       frmCircleOfBricks.Show
   End Sub
   ```

   The DrawCircleOfBricks macro opens the Circle of Bricks UserForm and is the only procedure from the project listed in the AutoCAD Macros dialog box.

2. Now all you have left to do is add the code to the event procedures for the two command buttons. Enter the code shown in Listing 7.4 into the Circle of Bricks UserForm's Code window. There will be two skeleton Click event procedures for the two command buttons already waiting.

LISTING 7.4: THE COMMAND BUTTON EVENT PROCEDURES

```
1   Private Sub cmdCancel_Click()
2       Unload Me
3   End Sub
4
5   Private Sub cmdCreatePavers_Click()
```

```
6       Unload Me
7       DrawCircularPavers
8   End Sub
```

ANALYSIS

- Line 1 opens the `cmdCancel` Click event procedure.
- Line 2 unloads the Circle of Bricks UserForm and return to the AutoCAD window without making any changes.
- Line 5 starts the event procedure that runs when the user clicks the Create Pavers command button.
- Line 6 unloads the Circle of Bricks UserForm.
- Line 7 calls the `DrawCircularPavers` procedure to draw the lines between the bricks in the current circle.
- Line 8 ends the event procedure.

Now run your Circle of Bricks application from the AutoCAD Macros dialog box. Figure 7.5 shows the result of running the application with the default settings shown in Figure 7.4. Figure 7.6 shows the result of running the application with "7" entered as the number of circles of bricks, and with the "Brick parallel to radius" option selected.

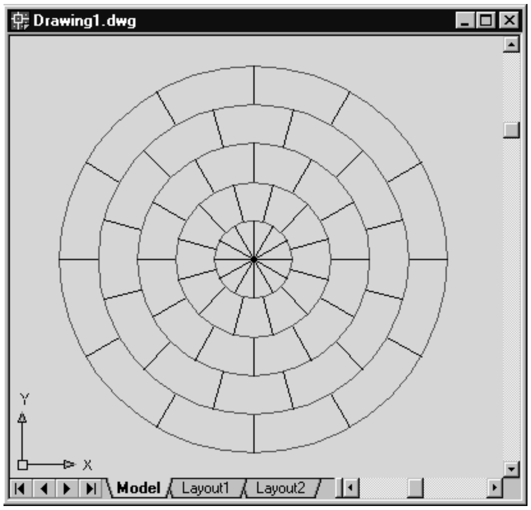


Figure 7.5 *Result of Circle of Bricks application with the settings from Figure 7.4*

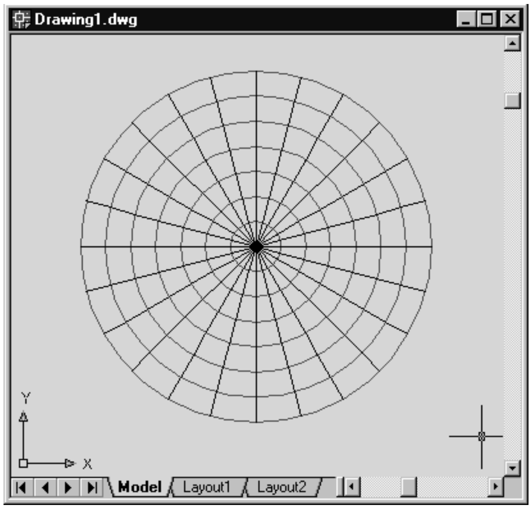


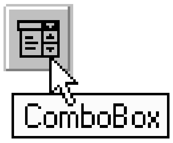
Figure 7.6 Result of Circle of Bricks application with 7 circles and the "Brick parallel to radius" option

Drawing Freeform Shapes

In this section you'll see how to develop an application that allows the user to select points defining a shape from the Model Space and fill it with a fill pattern. The user will be allowed to choose the fill pattern, as well as the representation of the selected points in the drawing.

EXERCISE 7.4: DRAWING FREEFORM SHAPES

1. Start a new project, and choose Insert ➜ UserForm to add a new UserForm. Place two ComboBox controls



dragged from the Toolbox, with accompanying Labels. Make the ComboBoxes roughly the same height as a text box control. Figure 7.7 gives a good arrangement for these controls. The drop-down lists of the Combo Box controls will display the available point middles and fill patterns for the user to select, to be used in the Hatch object.

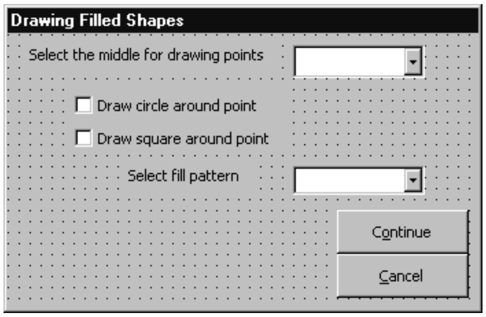


Figure 7.7 *GUI for specifying how points will be drawn, and the fill pattern*

2. Add two CheckBox controls that will be used to designate whether a circle, a square, or both surround the middle that is drawn to represent a point.
3. Add two command buttons, one for continuing the drawing and one for canceling the whole thing and returning to AutoCAD.
4. Change the properties of the UserForm and its controls to those listed in Table 7.2.

Table 7.2 Properties for the GUI Controls for Drawing Filled Shapes

OLD NAME	NEW NAME	CAPTION
UserForm1	frmDrawFilledShape	Drawing Filled Shapes
ComboBox1	cboPointMiddle	—
ComboBox2	cboFillPattern	—
CheckBox1	chkCircleAroundPoint	Draw circle around point
CheckBox2	chkSquareAroundPoint	Draw square around point
CommandButton1	cmdContinue	Continue
CommandButton2	cmdCancel	Cancel
Label1	—	Select the middle for drawing points
Label2	—	Select fill pattern

5. Listing 7.5 adds the list of middles and fill patterns to the combo boxes in the UserForm's Initialize event procedure. It also sets their Text properties to the default middle and fill values. Enter this code into the UserForm_Initialize skeleton.

The CheckBox control has the capability to display text and so does not require an accompanying Label control.

Listing 7.5: Initialize Procedure

```
1   Private Sub UserForm_Initialize()
2       'add items to the middle combo box
3       With cboPointMiddle
4           .AddItem "dot"
5           .AddItem "empty"
6           .AddItem "plus"
7           .AddItem "cross"
8           .AddItem "line"
9           'set the default item
10          .Text = "empty"
11      End With
12      'add items to the fill combo box
13      With cboFillPattern
14          .AddItem "SOLID"
15          .AddItem "BRICK"
16          .AddItem "DOLMIT"
17          .AddItem "ZIGZAG"
18          .AddItem "LINE"
19          .AddItem "BOX"
20          .AddItem "DOTS"
21          .AddItem "EARTH"
22          .AddItem "HONEY"
23          'set the default item
24          .Text = "SOLID"
25      End With
26  End Sub
```

Analysis

- Line 1 starts the `Initialize` event procedure that automatically runs when the UserForm is first loaded.

- Line 3 opens the `With` block so that items can be added to the `cboPointMiddle` combo box without the `AddItem` method being fully qualified.
- Lines 4 through 8 use the `AddItem` method to add to the first combo box all the items available for the point middle. These items will be included in the drop-down list that appears when the user clicks on the down-arrow button to the right of the combo box.
- Line 10 sets the `Text` property of the combo box to `"empty"`, which becomes the default and will appear in the combo box when the UserForm opens.
- Line 11 ends the `With` statement block.
- Line 13 opens the `With` statement block so that items can be added to the drop-down list in the `cboFillPattern` combo box without the `AddItem` method being fully qualified.
- Lines 14 through 22 add fill patterns to the `cboFillPattern` combo box.
- Line 24 sets the `Text` property of the Fill Pattern combo box to `"SOLID"`, which will appear in the combo box as the default.
- Line 25 ends the `With` statement block.
- Line 26 ends the `Initialize` event procedure.

Now you need a macro that first obtains the user's requirements for drawing points and filling shapes, then prompts the user to click on the required points in the Model tab, and finally makes sure the shape is closed before filling it with the pattern requested. The macro shown in Listing 7.6 does all these things. Enter this code into a standard module (choose Insert ➜ Module).

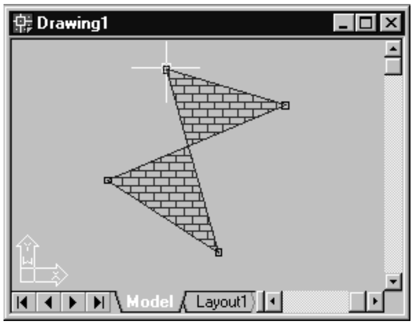
Listing 7.6: DrawGenericShape Macro

```
1   Sub DrawGenericShape()
2       frmDrawFilledShape.Show
3       GetShape 'get points defining shape from user
4       CloseShape
5       DrawFilledShape
6       Unload frmDrawFilledShape
7   End Sub
```

Analysis

- Line 1 starts the `DrawGenericShape` procedure, which is the main macro for this application.

- Line 2 starts the GUI by loading and displaying the UserForm, to allow the user to designate the requirements for visually representing the points clicked and the pattern for filling polygons. When the UserForm is loaded, its `UserForm_Initialize` event procedure is executed (Listing 7.5). When the user has finished selecting options for the points and the fill for the shape, the UserForm is closed and execution returns to line 3 in this macro.
- Line 3 calls the `GetShape` procedure (Listing 7.7) that prompts the user to enter the points specifying the required shape into the AutoCAD window.
- Line 4 calls the `CloseShape` procedure that creates a last point for the shape and assigns the same coordinates as the first point, so that the shape is guaranteed to be closed when the `DrawFilledShape` procedure is called in the next statement. The shape will be correctly filled even if it is convex, as shown here:



- Line 5 calls the `DrawFilledShape` procedure that fills the shape with the fill pattern selected by the user.
- Line 6 unloads the `frmDrawFilledShape` UserForm.
- Line 7 ends the `DrawGenericShape` procedure.

Listing 7.7 shows the `GetShape` procedure that prompts the user to enter the points defining the shape. Your next step is to enter this code into Module1.

LISTING 7.7: GETSHAPE PROCEDURE

```
1   Public Sub GetShape()
2   Dim OuterCircle(0) As AcadCircle
3   Dim NewPoint As AcadPoint 'reference to current Point object
4   Dim Point3D As Variant 'last point input
5   Dim Center(0 To 2) As Double, CircleRadius As Double
6   Dim Finished As Boolean
```

```
7   Finished = False
8   NumberOfElements = -1
9   'draw a red circle for user to click when finished
10  CircleRadius = 0.25
11  Set OuterCircle(0) = ThisDrawing.ModelSpace.AddCircle↵
       (Center, CircleRadius)
12  OuterCircle(0).Color = acRed
13  OuterCircle(0).Update
14  ZoomAll
15  While Not Finished
16      'continue receiving user input until the red circle ↵
           is clicked
17      Point3D = ThisDrawing.Utility.GetPoint(, "Click on ↵
           next point or red circle to finish!")
18      If Sqr(Point3D(0) ^ 2 + Point3D(1) ^ ↵
           2 + Point3D(2) ^ 2) < CircleRadius Then
19          'red circle has been clicked
20          Finished = True
21          OuterCircle(0).Delete 'removes red circle
22          ThisDrawing.Regen acActiveViewport
23      Else 'update number of array elements and ↵
           redimension the array
24          NumberOfElements = NumberOfElements + 3
25          ReDim Preserve PolyArray(0 To NumberOfElements)
26          'add the last three coordinates input by the ↵
               user to the array
27          PolyArray(NumberOfElements - 2) = Point3D(0)
28          PolyArray(NumberOfElements - 1) = Point3D(1)
29          PolyArray(NumberOfElements) = Point3D(2)
30          Set NewPoint = ↵
               ThisDrawing.ModelSpace.AddPoint(Point3D)
31          'draw line if required
32          If NumberOfElements > 2 Then RedrawPolyline
33        End If
34  Wend
35  End Sub
```

Analysis

- Line 1 starts the `GetShape` procedure that stores the points entered by the user in a dynamically expanding array that is extended as each point is selected, to ensure that it contains all the points clicked so far. A polyline based on this array is redrawn each time it's updated, so that the user always sees the points they have already entered.

- Line 2 declares the `OuterCircle` variable that is capable of referencing a circle object. The user will click inside this circle when they have finished entering their shape points.
- Line 3 declares `NewPoint` as a variable capable of holding a reference to a point object.
- Line 4 declares `Point3D` as a Variant, so that it can be assigned a three-dimensional array by the GetPoint method of the Utility object. As previously stated, the only way an array can be assigned as a whole rather than element by element is by declaring it as a Variant.
- Line 5 declares the array holding the x-, y-, and z-coordinates of the circle's center, and the circle's radius.
- Line 6 declares the `Finished` variable as Boolean. This variable is set to False until the user clicks inside the red circle. The `While` loop block (Lines 15 through 34) continues to be executed until `Finished` becomes False.
- Line 7 sets `Finished` to False so that the `While` loop block of statements will be run at least once.
- Line 8 sets the `NumberOfElements` variable to `–1` so that it reflects the highest index value of the `PolyArray` array—this value will become zero after the first point has been entered. For example, at line 24 you'll see that this variable is incremented by 3 so that the first point's coordinates will be stored at positions 0 through 2 in the array (lines 27 through 29), and the second point's coordinates will be stored at 3 through 5, and so on.
- Line 10 assigns 0.25 to the `CircleRadius` variable. This can be set to any value you choose, but it should be made large enough that the user can easily click it.
- Line 11 calls the `AddCircle` method to add a circle to the ModelSpace collection, and assigns a reference to it to the `OuterCircle` variable.
- Line 12 sets the color of the circle to red.
- Line 13 calls the `Update` method of the circle object to ensure that it appears in the Model Space.
- Line 14 calls the `ZoomAll` method to ensure that the entire drawing (including the red circle) appears on the screen.

- Line 15 starts the `While` loop block.
- Line 17 calls the `GetPoint` method to prompt the user to click a point in the Model tab and assigns the coordinates of the point to the `Point3D` variable.
- Line 18 tests to see if the last point entered falls inside the circle, indicating that the user has finished.
- Line 20 runs if the user clicked inside the circle; it sets the `Finished` variable to True to stop the `While` loop looping.
- Line 21 deletes the red circle object from the ModelSpace collection.
- Line 22 calls the `Regen` method to update the active viewport, which makes the circle disappear from the screen.
- Line 23 starts the `Else` part of the `While` statement block, which runs when the user has clicked a point in the Model tab.
- Line 24 adds 3 to the `NumberOfElements` variable; this is because storing the x-, y-, and z-coordinates of the point clicked requires that three more elements are added to the array size.
- Line 25 redimensions the array to accommodate the new point. The `Preserve` word is used here to instruct the interpreter not to throw away the existing contents, but rather to extend the array by appending the three new elements.
- Lines 27 through 29 assign the coordinates of the point clicked to the `Point3D` array.
- Line 30 creates a new `Point` object and assigns a reference to it to the `New-Point` variable.
- Line 32 tests if the user is clicking their second point and, if so, calls the `RedrawPolyline` procedure (Listing 7.8). This procedure deletes the last polyline from the screen and generates a new one containing the point just clicked by the user.
- Lines 33, 34, and 35 end the `If` statement, `While` loop, and `GetShape` macro, respectively.

Now, with the Drawing Filled Freeform Shapes application in place, you need to add the global procedures that draw the filled shape. Redraw the polyline, defining the freeform shape after an additional boundary point has been added, and close the application.

1. Enter the CloseShape, DrawFilledShape, and RedrawPolyline procedures (Listing 7.8) into Module1.
2. Place the `Dim` statements in Lines 1 through 6 into the General Declarations section of Module1's Code window.

Listing 7.8: Entering Global Variables and Procedures into Module1

```
1        'PolygonObject will reference the Polyline object
2        Dim PolygonObject(0) As AcadPolyline
3        'PolyArray will contain all points clicked by user
4        Dim PolyArray() As Double
5        'NumberOfElements set to number of elements in the array
6        Dim NumberOfElements As Integer
7
8    Public Sub CloseShape()
9        'add the last point to the array and make it
10       'the same coordinates as the first one to ensure closure
11       NumberOfElements = NumberOfElements + 3
12       ReDim Preserve PolyArray(0 To NumberOfElements)
13       PolyArray(NumberOfElements - 2) = PolyArray(0)
14       PolyArray(NumberOfElements - 1) = PolyArray(1)
15       PolyArray(NumberOfElements) = PolyArray(2)
16       RedrawPolyline
17   End Sub
18
19   Public Sub DrawFilledShape()
20       'HatchObject will reference the Hatch object
21       Dim HatchObject As AcadHatch
22       'create Hatch object and assign reference to HatchObject
23       Set HatchObject = ThisDrawing.ModelSpace.AddHatch↲
            (acHatchPatternTypePreDefined, ↲
            frmDrawFilledShape.cboFillPattern.Text, True)
24       'add the outer loop boundary to the Hatch object
25       HatchObject.AppendOuterLoop (PolygonObject)
26       HatchObject.Color = 180
27       HatchObject.Evaluate
28       HatchObject.Update
29   End Sub
30
31   Public Sub RedrawPolyline()
32       'delete last Polyline, create nextand draw it
```

```
33      With ThisDrawing.ModelSpace
34          If NumberOfElements > 5 Then PolygonObject(0).Delete
35          Set PolygonObject(0) = .AddPolyline(PolyArray)
36          PolygonObject(0).Color = acBlue
37          PolygonObject(0).Update
38      End With
39  End Sub
```

Analysis

- Lines 1 through 6 declare the variables that will be used in various places throughout the code in this module.
- Lines 8 through 17 contain the `CloseShape` procedure that is called after the user presses the Enter key. This procedure places one last point identical to the first point into the `PolyArray` array, and calls the `RedrawPolyLine` procedure to redraw the shape.

The first and last points of a shape to be filled must be equivalent to ensure that the polyline is closed; otherwise the color will bleed through any gaps in the bounding polyline and cover the drawing area. You cannot put the onus on the user, because the mouse isn't accurate enough to select a specific screen pixel.

- Line 19 starts the `DrawFilledShape` procedure that uses a Hatch object to fill the shape entered by the user.
- Line 21 declares the `HatchObject` variable as being capable of referencing a Hatch object.
- Line 23 calls the `AddHatch` method to create a Hatch object and add it to the ModelSpace collection. The `AddHatch` method is called with the AutoCAD constant `acHatchPatternTypePredefined` so that the interpreter searches the `acad.pat` file for the definition of the pattern. The second argument, `frmDrawFilledShape.cboFillPattern.Text`, uses the Text property setting from the Fill Pattern combo box containing the pattern's name. Lastly, the `True` value is passed to the `Associativity` parameter so that the pattern changes with the boundary.
- Line 25 calls the `AppendOuterLoop` method with the single-element `PolygonObject` array that is set up to reference a `PolyLine` object in the assignment statement in line 35.

- Lines 26 through 28 all use the reference to the Hatch object to change its color, to calculate how it will appear inside the boundary, and to redraw it in the Model Space. Line 26 sets the `Color` property to 180—up until now I've always used one of AutoCAD's color constants, but you can also use a number in the range 0 through 256.

The Color property of an object can be assigned AutoCAD color constants or a number in the range 0 to 256.

- Line 31 starts the `RedrawPolyline` procedure, which deletes the old Polyline object and creates a new one that includes the point just entered by the user. It draws the new polyline to give the user feedback on how their shape is progressing.
- Line 34 tests to see how many coordinates have been added to the array so far, because the Polyline object isn't created until the user has entered the second point. If `NumberOfElements` is greater than 5, then the Polyline object exists and needs to be deleted using the `Delete` method as shown. This test is necessary because any attempt to delete an object before it has been created will cause your application to terminate abnormally.
- Line 35 creates a new Polyline object containing all the points entered so far.
- Lines 36 and 37 assign blue to the `Color` property and redraw the polyline in the ModelSpace, so that the user can view all the points they have entered to date.
- Line 38 ends the `With` statement block.
- Line 39 ends the `RedrawPolyline` procedure.

Listing 7.9 shows the code that responds to the user's click of the Continue button. This button's Click event procedure calculates a number that determines how a point will appear when it's passed with the `PDMODE` system variable that determines how points are drawn. When this is done, the UserForm is closed and returns the user to the AutoCAD window. Enter the code given in Listing 7.9 into the Cancel command button's Click event procedure.

LISTING 7.9: THE CANCEL COMMAND BUTTON'S CLICK EVENT PROCEDURE

```
Private Sub cmdCancel_Click()
    frmDrawFilledShape.Hide
End Sub

```

```
5   Private Sub cmdContinue_Click()
6       'calculate PointDisplay based on user selections
7       Dim PointDisplay As Integer
8       Select Case cboPointMiddle.Text
9           Case "dot"
10              PointDisplay = 0
11          Case "empty"
12              PointDisplay = 1
13          Case "plus"
14              PointDisplay = 2
15          Case "cross"
16              PointDisplay = 3
17          Case "line"
18              PointDisplay = 4
19      End Select
20      If chkCircleAroundPoint.Value = True ↵
           Then PointDisplay = PointDisplay + 32
21      If chkSquareAroundPoint.Value = True ↵
           Then PointDisplay = PointDisplay + 64
22      'use the system variable PDMODE to draw points to ↵
           the user's specification
23      ThisDrawing.SetVariable "PDMODE", PointDisplay
24      Unload Me
25  End Sub
```

Analysis

- Lines 1 through 3 contain the `cmdCancel_Click` event procedure that closes the UserForm and returns the user to the AutoCAD window.
- Line 5 starts the `cmdContinue_Click` event procedure that calculates the setting required by the system variable to draw the points in the format required by the user, and to fill the shape with the pattern selected.
- Lines 7 through 23 calculate the `PointDisplay` variable and pass it as the argument to the system variable PDMODE. This variable is given a value in the range 0 to 4 according to the middle required. If a circle is required around the point, then 32 is added; if a square is required, then 64 is added.
- Line 24 unloads the Draw Filled Shape UserForm.
- Line 25 ends the `cmdCancel_Click` event procedure.

To run your application, open the Macros dialog box from the AutoCAD window and select DrawGenericShape. Figure 7.8 shows the Model tab with the following settings: Cross has been selected as the middle; the Draw Circle Around Point check box has been selected; and the ZigZag fill pattern has been chosen from the Select Fill Pattern combo box. Figure 7.9 shows the Model tab after the red circle was clicked, which caused the CloseShape procedure to take the last point entered by the user and join it up with the first one entered.

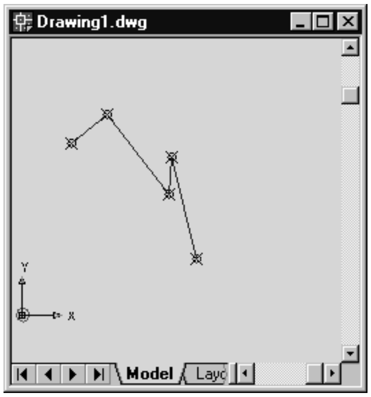


Figure 7.8 *The Model tab after the user has entered points defining a shape*

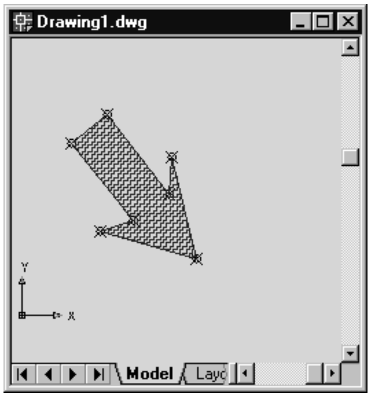
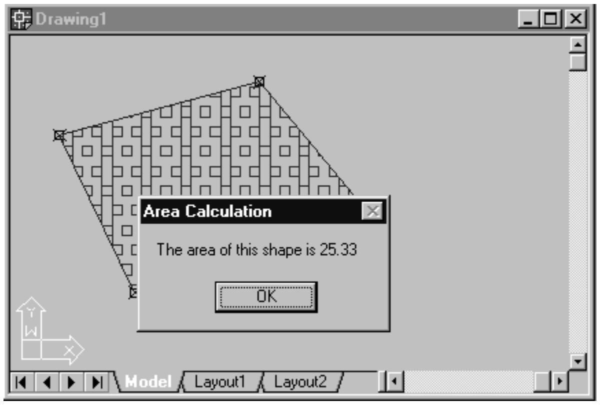


Figure 7.9 *Shape filled with ZIGZAG pattern*

Calculating Areas of Shapes

AutoCAD has an Area property that holds the calculated area of any closed shape such as an arc, circle, ellipse, or polyline. Let's extend the Drawing Filled Shapes project to incorporate a display of the shape's area after the user has pressed Enter to draw it. The results are displayed in a message box (Figure 7.10).

The CalculateAreaOfShape procedure shown in Listing 7.10 shows how simple this is to do. This procedure is not stand-alone, in that it must be called from within the DrawGenericShape macro, so that the PolygonObject variable refers to a Polyline object and is available so that we can access its Area method. Call CalculateAreaOfShape from the DrawGenericShape macro given in Listing 7.6, just after the call to the DrawFilled-Shape macro.



```
The area of this shape is 25.33
```

Figure 7.10 *Message box displaying the area calculated for the shape just entered*

LISTING 7.10: CALCULATEAREAOFSHAPE MACRO

```
1   Public Sub CalculateAreaOfShape()
2       MsgBox "The area of this shape is " ↵
           & Format(PolygonObject(0).Area, "###0.00"), , ↵
           "Area Calculation"
3   End Sub
```

ANALYSIS

- Line 1 starts the `CalculateAreaOfShape` procedure that displays the value of the Polyline object's Area property in a message box.

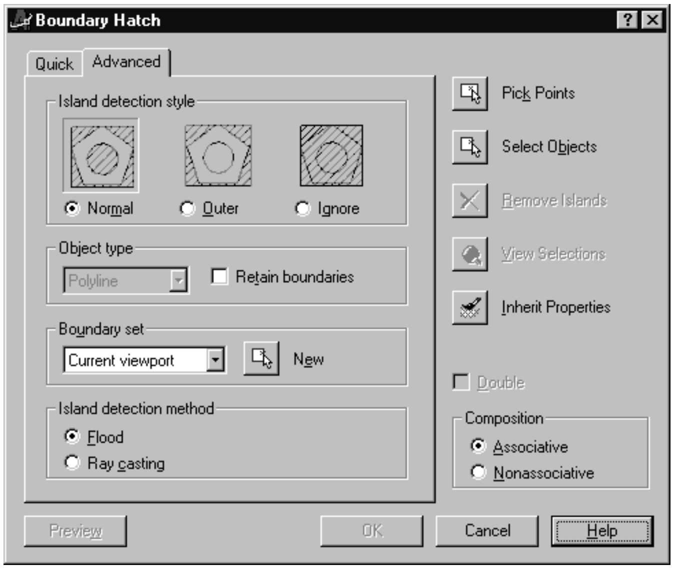
- Line 2 uses the `MsgBox` function to display the area to the user. (This function is discussed in the section "Communicating with Message Boxes" in Chapter 5.) The message argument uses the `Format` function, which returns a variant containing a string formatted as specified by a format string. In this statement, the function is passed the Area property's value and returns a number with two decimal places after the point.

You'll find out more about formatting text in Chapter 9.

- Line 3 ends the `CalculateAreaOfShape` procedure.

Filling with Inner and Outer Loops

In this project you'll see how to fill areas that are nested inside other areas, and the various results that can be achieved using the Island Detection styles. To see the styles available from the AutoCAD window, choose Draw ➜ Hatch to open the Boundary Hatch dialog box, and select the Advanced tab (see Figure 7.11).


Boundary Hatch

Quick | Advanced

Island detection style: [✓ Normal] Outer Ignore

Object type: Polyline ☐ Retain boundaries

Boundary set: Current viewport New

Island detection method: [✓ Flood] Ray casting

☐ Double

Composition: [✓ Associative] Nonassociative

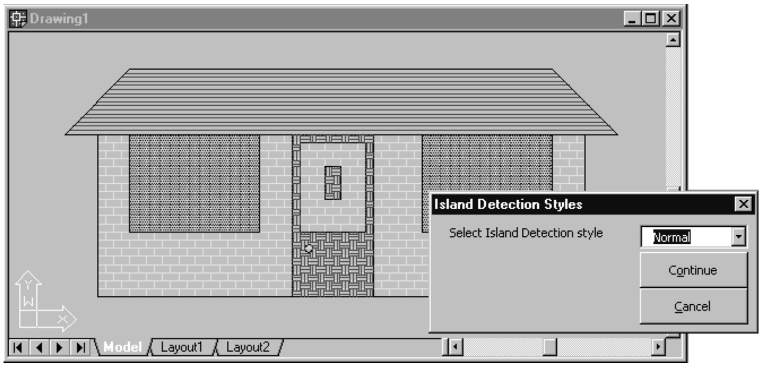
Figure 7.11 *Advanced tab of the Boundary Hatch dialog box*

The Island Detection style defines how objects that are nested inside the outermost hatch boundary are filled. There are three settings available:

Normal Hatching starts at the outermost boundary and moves inward. When the next nested boundary is encountered, the hatching is toggled on or off. The result is that inner objects are alternately shaded and unshaded. Figure 7.12 shows a door drawn using the Normal Island Detection style.

Outer Hatching starts from the outermost boundary and moves inward, stopping as soon as the first nested boundary is encountered.

Ignore Everything inside the outermost boundary is hatched; any nested boundaries are ignored.



Select Island Detection style: Normal

Figure 7.12 *Door drawn using the Normal Island Detection hatch style*

If there are no nested objects, the Island Detection setting has no effect on the drawing.

Let's develop a small GUI that allows the user to select one of the three Island Detection styles. Each time they click on a new style, the drawing in the Model tab is updated to show the effect of the selected style.

Exercise 7.5: Island Detection Styles Application

1. Start a new project. Add a Label, a ComboBox control, and two command buttons to the UserForm (see the dialog box shown in Figure 7.12).
2. Change the Name and Caption properties to those shown in Table 7.3.

Table 7.3 Name and Caption Properties for Island Detection Styles UserForm

OLD NAME	NEW NAME	CAPTION
UserForm1	frmIslandStyles	Island Detection Styles
ComboBox1	cboIslandStyle	—
Label1	—-	Select Island Detection style
CommandButton1	cmdContinue	Continue
CommandButton2	cmdCancel	Cancel

3. Change the Accelerator property of the Continue command button to **o,** and the Cancel button to **C.**
4. Enter the code from Listing 7.11 into the skeleton event procedures provided in the UserForm's Code window.

The Initialize event procedure is long and repetitive, and I apologize for this. The code is available to copy/paste from the CD. Normally when a procedure needs lots of data, it reads it from a data file; you'll see how this is done when you get to Chapter 12.

5. Run your project from the VBA IDE, and watch the drawing in the Model tab change as you make new selections from the combo box.

LISTING 7.11: EVENT PROCEDURES FOR THE ISLAND DETECTION STYLES USERFORM

```
Dim HatchObject As AcadHatch

Private Sub cboIslandStyle_Change()
    Select Case cboIslandStyle.Text
        Case "Normal"
        HatchObject.HatchStyle = acHatchStyleNormal
        Case "Outer"
        HatchObject.HatchStyle = acHatchStyleOuter
        Case "Ignore"
        HatchObject.HatchStyle = acHatchStyleIgnore
    End Select
    HatchObject.Evaluate
    HatchObject.Update
```

```
End Sub

Private Sub cmdCancel_Click()
    HatchObject.HatchStyle = acHatchStyleNormal
    HatchObject.Evaluate
    HatchObject.Update
    Unload Me
End Sub

Private Sub cmdContinue_Click()
    Unload Me
End Sub

Private Sub UserForm_Initialize()
    'declare all AcadEntity reference variables ↲
       for hatch loops
    Dim RoofLoop(0) As AcadEntity
    Dim WallLoop(0) As AcadEntity
    Dim Window1Loop(0) As AcadEntity
    Dim Window2Loop(0) As AcadEntity
    Dim DoorLoop(0) As AcadEntity
    Dim DoorWindow1Loop(0) As AcadEntity
    Dim DoorWindow2Loop(0) As AcadEntity
    Dim DoorHandleLoop(0) As AcadEntity
    'declare arrays to hold the points defining the house
    Dim Roof(0 To 14) As Double
    Dim Wall(0 To 14) As Double
    Dim Window1(0 To 14) As Double
    Dim Window2(0 To 14) As Double
    Dim Door(0 To 14) As Double
    Dim DoorWindow1(0 To 14) As Double
    Dim DoorWindow2(0 To 14) As Double
    Dim CircleCenter(0 To 2) As Double
    'define the roof
    Roof(0) = 2#: Roof(1) = 8#: Roof(2) = 0#
    Roof(3) = 15#: Roof(4) = 8#: Roof(5) = 0#
    Roof(6) = 17#: Roof(7) = 6#: Roof(8) = 0#
    Roof(9) = 0#: Roof(10) = 6#: Roof(11) = 0#
   Roof(12) = 2#: Roof(13) = 8#: Roof(14) = 0#
    Set HatchObject = ThisDrawing.ModelSpace.AddHatch ↲
       (acHatchPatternTypePreDefined, "LINE", True)
```

```
Set RoofLoop(0) = ↵
   ThisDrawing.ModelSpace.AddPolyline(Roof)
HatchObject.AppendOuterLoop (RoofLoop)
HatchObject.Color = acRed
HatchObject.Evaluate
HatchObject.Update
'define the wall
Wall(0) = 1#: Wall(1) = 6#: Wall(2) = 0#
Wall(3) = 16#: Wall(4) = 6#: Wall(5) = 0#
Wall(6) = 16#: Wall(7) = 1#: Wall(8) = 0#
Wall(9) = 1#: Wall(10) = 1#: Wall(11) = 0#
Wall(12) = 1#: Wall(13) = 6#: Wall(14) = 0#
Set HatchObject = ThisDrawing.ModelSpace.AddHatch ↵
   (acHatchPatternTypePreDefined, "BRICK", True)
Set WallLoop(0) = ↵
   ThisDrawing.ModelSpace.AddPolyline(Wall)
HatchObject.AppendOuterLoop (WallLoop)
HatchObject.Color = acYellow
HatchObject.Evaluate
HatchObject.Update
'define the left window
Window1(0) = 2#: Window1(1) = 6#: Window1(2) = 0#
Window1(3) = 6#: Window1(4) = 6#: Window1(5) = 0#
Window1(6) = 6#: Window1(7) = 3#: Window1(8) = 0#
Window1(9) = 2#: Window1(10) = 3#: Window1(11) = 0#
Window1(12) = 2#: Window1(13) = 6#: Window1(14) = 0#
Set HatchObject = ThisDrawing.ModelSpace.AddHatch ↵
   (acHatchPatternTypePreDefined, "dots", True)
Set Window1Loop(0) = ↵
   ThisDrawing.ModelSpace.AddPolyline(Window1)
HatchObject.AppendOuterLoop (Window1Loop)
HatchObject.Color = acBlue
HatchObject.Evaluate
HatchObject.Update
'define the right window
Window2(0) = 11#: Window2(1) = 6#: Window2(2) = 0#
Window2(3) = 15#: Window2(4) = 6#: Window2(5) = 0#
Window2(6) = 15#: Window2(7) = 3#: Window2(8) = 0#
Window2(9) = 11#: Window2(10) = 3#: Window2(11) = 0#
Window2(12) = 11#: Window2(13) = 6#: Window2(14) = 0#
Set HatchObject = ThisDrawing.ModelSpace.AddHatch ↵
```

```
   (acHatchPatternTypePreDefined, "dots", True)
Set Window2Loop(0) = ↲
   ThisDrawing.ModelSpace.AddPolyline(Window2)
HatchObject.AppendOuterLoop (Window2Loop)
HatchObject.Color = acBlue
HatchObject.Evaluate
HatchObject.Update
'define the door
Door(0) = 7#: Door(1) = 6#: Door(2) = 0#
Door(3) = 9.5: Door(4) = 6#: Door(5) = 0#
Door(6) = 9.5: Door(7) = 1#: Door(8) = 0#
Door(9) = 7#: Door(10) = 1#: Door(11) = 0#
Door(12) = 7#: Door(13) = 6#: Door(14) = 0#
Set HatchObject = ThisDrawing.ModelSpace.AddHatch ↲
   (acHatchPatternTypePreDefined, "earth", True)
Set DoorLoop(0) = ↲
   ThisDrawing.ModelSpace.AddPolyline(Door)
HatchObject.AppendOuterLoop (DoorLoop)
HatchObject.Color = acGreen
HatchObject.Evaluate
HatchObject.Update
'define the window in the door
DoorWindow1(0) = 7.25: DoorWindow1(1) = 5.75: ↲
   DoorWindow1(2) = 0#
DoorWindow1(3) = 9.25: DoorWindow1(4) = 5.75: ↲
   DoorWindow1(5) = 0#
DoorWindow1(6) = 9.25: DoorWindow1(7) = 3#: ↲
   DoorWindow1(8) = 0#
DoorWindow1(9) = 7.25: DoorWindow1(10) = 3#: ↲
   DoorWindow1(11) = 0#
DoorWindow1(12) = 7.25: DoorWindow1(13) = 5.75: ↲
   DoorWindow1(14) = 0#
Set DoorWindow1Loop(0) = ↲
   ThisDrawing.ModelSpace.AddPolyline(DoorWindow1)
HatchObject.AppendInnerLoop (DoorWindow1Loop)
HatchObject.Color = acBlue
HatchObject.Evaluate
HatchObject.Update

DoorWindow2(0) = 8#: DoorWindow2(1) = 5#: ↲
   DoorWindow2(2) = 0#
```

```
119     DoorWindow2(3) = 8.5: DoorWindow2(4) = 5#: ↵
           DoorWindow2(5) = 0#
120     DoorWindow2(6) = 8.5: DoorWindow2(7) = 4#: ↵
           DoorWindow2(8) = 0#
121     DoorWindow2(9) = 8#: DoorWindow2(10) = 4#: ↵
           DoorWindow2(11) = 0#
122     DoorWindow2(12) = 8#: DoorWindow2(13) = 5#: ↵
           DoorWindow2(14) = 0#
123     Set DoorWindow2Loop(0) = ↵
           ThisDrawing.ModelSpace.AddPolyline(DoorWindow2)
124     HatchObject.AppendInnerLoop (DoorWindow2Loop)
125     HatchObject.Color = acBlue
126     HatchObject.Evaluate
127     HatchObject.Update
128     'define the door handle
129     CircleCenter(0) = 7.5: CircleCenter(1) = 2.5: ↵
           CircleCenter(2) = 0#
130     Set DoorHandleLoop(0) = ThisDrawing.ModelSpace.AddCircle ↵
           (CircleCenter, 0.125)
131     HatchObject.AppendInnerLoop (DoorHandleLoop)
132     HatchObject.Color = acRed
133     HatchObject.Evaluate
134     HatchObject.Update
135     'add items to the combo box
136     cboIslandStyle.AddItem "Normal"
137     cboIslandStyle.AddItem "Outer"
138     cboIslandStyle.AddItem "Ignore"
139     'set default combo box island style
140     cboIslandStyle.Text = "Normal"
141 End Sub
```

Analysis

- Line 1 declares `HatchObject` as a variable capable of referencing a Hatch object. This declaration is made in the General Declarations section of the UserForm's Code window so that it can be accessed by any procedure from this UserForm.
- Lines 3 through 14 contain the `Change` event procedure of the `Island` combo box. These statements run in the event that the user selects a different style; the `HatchStyle` property is set to one of the AutoCAD constants, depending on the user's latest choice. The `Evaluate` method is called to recompute where the

hatch pattern intersects with the boundary using the new style, before the Update method redraws it on the Model Space.

Notice how `cbo` *is used as the prefix for the combo box, as per Windows object-naming conventions.*

- Lines 16 through 21 contain the Click event procedure for the Cancel command button. This reinitializes the `HatchStyle` to Normal before evaluating the pattern-boundary intersections and redrawing the shape. The last statement unloads the UserForm and returns the user to the AutoCAD window.
- Lines 23 through 25 unload the UserForm, leaving the `HatchStyle` set at the last one selected by the user, and then return to the AutoCAD window.
- Lines 27 through 141 contain all the statements from the `Initialize` event procedure that runs when the UserForm is first loaded.
- Lines 29 through 36 declare one-element arrays that are all capable of referencing an AutoCAD object. An array is required as the argument by the `AppendOuterLoop` method, rather than a single value variable.
- Lines 38 through 44 declare arrays that will contain the doors and windows definitions.
- Lines 47 through 51 assign the coordinates defining the roof to the `Roof` array.
- Line 52 calls the `AddHatch` method to create a Hatch object and sets up the `HatchObject` variable to refer to it.
- Line 53 calls the `AddPolyLine` method to create a PolyLine object that defines the roof. The `RoofTop` array is set up as a reference to the PolyLine object.
- Line 54 calls the `AppendOuterLoop` method of the Hatch object to specify the boundary of the outer loop. This method must be called before any inner loops can be defined.
- Line 55 assigns the color red to the Hatch object.
- Line 56 calls the `Evaluate` method to evaluate the lines or fill color to be used for the Hatch pattern.
- Line 57 updates the Hatch object.
- Lines 58 through 69 perform the same operations on the wall definition as Lines 46 through 57 did on the roof.

- Lines 70 through 105 perform the same operations for the windows and door as Lines 46 through 57 did for the roof.
- Line 113 calls the `AppendInnerLoop` method to add the `DoorWindowLoop` array to the current `HatchObject` that already contains an outer loop.

The AppendOuterLoop method must be called as soon as the AddHatch method has been executed to create a new Hatch object. The AppendInnerLoop method should not be called before the AppendOuter-Loop has set up the outer boundary.

Working with Input Boxes

Input boxes are the special dialog boxes that pop up and ask the user to enter some piece of information that the application needs to continue. This function shares some similarities with the MsgBox function, but the dialog box displayed to the user in this case has a TextBox control in which the user enters their input. Figure 7.13 shows the list of parameters for the InputBox function.

The Prompt parameter is the only one that VBA requires be entered, and it expects a string that gives the user instructions about what data should be entered into the text box.

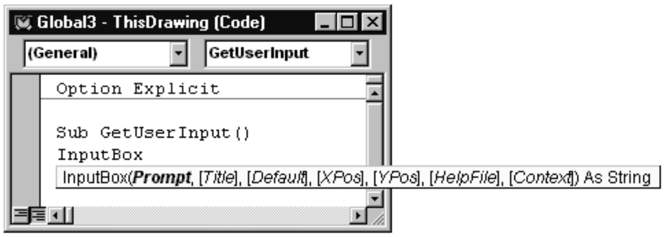
Listing 7.12 is the GetUserInput macro that calls the InputBox function with the two arguments listed to display the Input Box shown in Figure 7.14.

Listing 7.12: GetUserInput Macro

```
1   Sub GetUserInput()
2       Dim MyName As String
3       MyName = InputBox↵
           ("Please enter your name", "Login Details")
4   End Sub
```

Analysis

Line 3 calls the `InputBox` function, with the prompt and title parameters being passed the string values shown. The `InputBox` function then puts everything into modal mode while it waits for the user to enter their name or click one of the command buttons.



```
Option Explicit

Sub GetUserInput()
InputBox
```

InputBox(*Prompt*, [*Title*], [*Default*], [*XPos*], [*YPos*], [*HelpFile*], [*Context*]) As String

Figure 7.13 *Parameter list for the InputBox function*

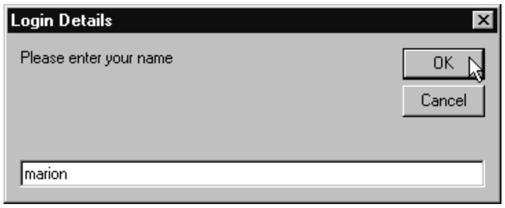


Figure 7.14 *Input box receiving name data*

Multiple-Line Prompts

If the instructions in the Prompt argument for an input box require more than one line, you can insert carriage returns using the Visual Basic constant vbCr at the point where you want to start the next line. For example, the statement

```
MyName = InputBox("Please enter your name" & vbCr ↵
   & "followed by your telephone number", "Login Details")
```

creates the prompt shown in Figure 7.15.

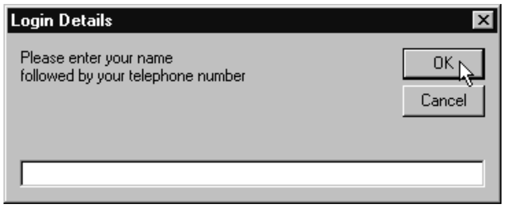


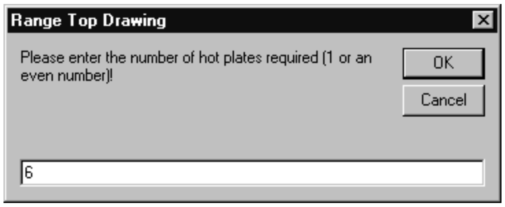
Figure 7.15 *Input box with two-line prompt*

Drawing a Range Top from a Macro

This next project combines several of the topics covered in this chapter. It uses the Utility object to prompt the user to click the points defining the position and dimensions of the range top, and then draws the range top. Then the InputBox function is used to prompt the user to enter the number of burners required—this is the only GUI employed by the project. It takes yet another approach to drawing circles—the ones that represent the range-top's burners—calculating their size and position in the macro itself before drawing them on the range top.

EXERCISE 7.6: THE DRAW RANGE TOP APPLICATION

1. Start a new project and enter the code shown in Listing 7.13 into the ThisDrawing Code window.
2. Run your project. A prompt appears in the command line, asking you to click the position for the bottom-left corner of the range top.
3. Click at the point required for the corner. The command line now asks you to specify the opposite corner.
4. Click the opposite corner. The input box appears, prompting for the number of hot plates required and telling you what sort of input is expected.


Range Top Drawing

Please enter the number of hot plates required (1 or an even number)!

6

5. Enter a number and click OK. The range top appears in the Model Space, with the number of burners you specified.

Figure 7.16 shows a selection of range tops created using this macro.

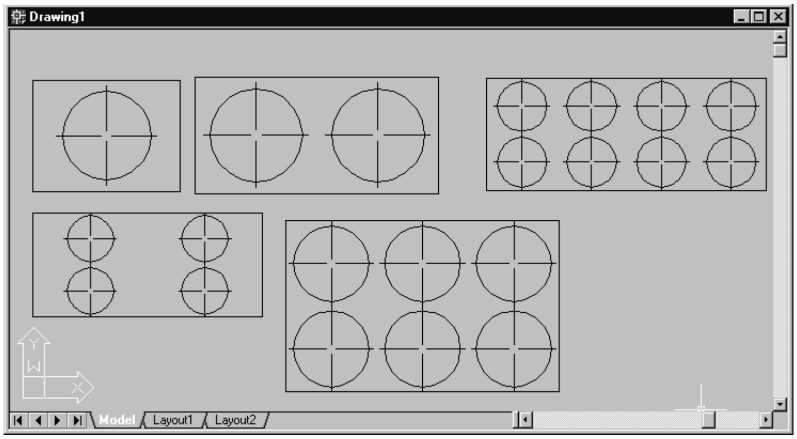


Figure 7.16 *Variety of range tops produced by the DrawRangeTop macro*

LISTING 7.13: DRAWRANGETOP MACRO

```
Sub DrawRangeTop()
    'range top related declarations
    Dim RangeObject As AcadPolyline
    Dim Depth As Variant, Width As Variant
    Dim Corner1 As Variant, Corner2 As Variant
    Dim Range(0 To 14) As Double
    Dim LineObject As AcadLine
    'circle related declarations
    Dim HotPlateObject As AcadCircle
    Dim Center(0 To 2) As Double
    Dim Radius As Double
    'Burner related declarations
    Dim NumberOfHotPlates As Integer
    Dim PlatesInARow As Integer, PlatesInACol As Integer
    Dim Count1 As Integer, Count2 As Integer
    Dim StartPoint(0 To 2) As Double, ↵
       EndPoint(0 To 2) As Double
    'prompt user for dimensions
    With ThisDrawing.Utility
```

```
        Corner1 = .GetPoint(, ↵
           "Click position for ↵
           bottom-left corner of range top.")
        Corner2 = .GetPoint(, "Specify opposite corner.")
    End With
    Range(0) = Corner1(0): Range(1) = Corner1(1): ↵
       Range(2) = Corner1(2)
    Range(3) = Corner1(0): ↵
       Range(4) = Corner2(1) + Depth: Range(5) = Corner1(2)
    Range(6) = Corner2(0) + Width: ↵
       Range(7) = Corner2(1) + Depth: Range(8) = Corner2(2)
    Range(9) = Corner2(0) + Width: ↵
       Range(10) = Corner1(1): Range(11) = Corner1(2)
    Range(12) = Corner1(0): Range(13) = Corner1(1): ↵
       Range(14) = Corner1(2)
    Set RangeObject = ↵
       ThisDrawing.ModelSpace.AddPolyline(Range)
    RangeObject.Update
    Width = Range(9) - Range(0)
    Depth = Range(4) - Range(1)
    'prompt user for number of hot plates
    NumberOfHotPlates = InputBox("Please enter the ↵
       number of hot plates required (1 or an even ↵
       number)!", "Range Top Drawing")
    'calculate layout of hot plates
    If NumberOfHotPlates > 2 Then
        PlatesInARow = NumberOfHotPlates / 2#
        PlatesInACol = 2
    Else
        PlatesInARow = NumberOfHotPlates
        PlatesInACol = 1
    End If
    'calculate position for each hot plate
    For Count2 = 1 To PlatesInACol
        Center(1) = ↵
           Range(1) + (Depth / (2# * PlatesInACol)) ↵
           + (Count2 - 1) * (Depth / 2#)
        For Count1 = 1 To PlatesInARow
```

```
Center(0) = ↵
   Range(0) + (Width / (2# * PlatesInARow)) ↵
   + (Count1 - 1) * (Width / PlatesInARow)
If (Width / PlatesInARow) < ↵
   (Depth / PlatesInACol) Then
    Radius = Width / (2 * PlatesInARow + 2#)
Else
    Radius = Depth / (2 * PlatesInACol + 0.5)
End If
Dim Adjustment As Double
Adjustment = Radius / 8#
'create and draw circle representing hot plate
With ThisDrawing.ModelSpace
    Set HotPlateObject = ↵
       .AddCircle(Center, Radius)
    HotPlateObject.Update
    'draw four lines on top of hot plate for gas
    StartPoint(0) = Center(0): StartPoint(1) = ↵
       Center(1) + Adjustment: StartPoint(2) = ↵
       Center(2)
    EndPoint(0) = Center(0): EndPoint(1) = ↵
       Center(1) + Radius + Adjustment: ↵
       EndPoint(2) = Center(2)
    Set LineObject = ↵
       .AddLine(StartPoint, EndPoint)
    LineObject.Update
    StartPoint(0) = ↵
       Center(0) + Adjustment: StartPoint(1) = ↵
       Center(1): StartPoint(2) = Center(2)
    EndPoint(0) = ↵
       Center(0) + Radius + Adjustment: ↵
       EndPoint(1) = Center(1): EndPoint(2) = ↵
       Center(2)
    Set LineObject = ↵
       .AddLine(StartPoint, EndPoint)
    LineObject.Update
    StartPoint(0) = Center(0): StartPoint(1) = ↵
       Center(1) - Adjustment: StartPoint(2) = ↵
       Center(2)
```

```
67                  EndPoint(0) = Center(0): EndPoint(1) = ↵
                       Center(1) - Radius - Adjustment: ↵
                       EndPoint(2) = Center(2)
68                  Set LineObject = ↵
                       .AddLine(StartPoint, EndPoint)
69                  LineObject.Update
70                  StartPoint(0) = ↵
                       Center(0) - Adjustment: StartPoint(1) = ↵
                       Center(1): StartPoint(2) = Center(2)
71                  EndPoint(0) = ↵
                       Center(0) - Radius - Adjustment: ↵
                       EndPoint(1) = Center(1): EndPoint(2) = ↵
                       Center(2)
72                  Set LineObject = ↵
                       .AddLine(StartPoint, EndPoint)
73                  LineObject.Update
74              End With
75          Next
76      Next
77  End Sub
```

Analysis

- Line 1 starts the `DrawRangeTop` macro, which interacts with the user to obtain the information needed to draw the outline of the range top, and then calculates the size and position of each of the burners.
- Lines 19 and 20 prompt the user for the position of two diagonally opposite corners for drawing the range top.
- Lines 22 through 26 assign the values of the x-, y-, and z-coordinates that make up the four corners of the range top, plus a fifth corner that is a duplicate of the first so that all four edges are drawn.
- Lines 29 and 30 calculate the width based on the x-coordinates from the `Range` array, and the depth based on the y-coordinates.
- Line 32 calls the `InputBox` function to get input from the user. The first argument is the prompt; the second argument is the caption for the title bar of the input box.

- Lines 33 through 40 determine how many rows and columns will be used when the hot plates are placed on the range top.
- Line 42 starts the outer `For` loop that executes once for each column of hot plates.
- Line 43 calculates the y-coordinate for the center.
- Line 44 starts the inner `For` loop that is repeated for each row of hot plates.
- Line 45 calculates the x-coordinate for the center.
- Lines 46 through 50 calculate the largest radius for the hot plates, based on the range top's dimensions and the number of burners required.
- Line 51 declares the `Adjustment` variable that determines how far the four radial lines will be translated from the hot plates' centers.
- Line 52 assigns a value to the `Adjustment` variable. This makes it easy to update the adjustment value, since it only needs to be done in this statement.
- Line 55 draws the circle representing the hot plate.
- Lines 57 through 73 create four Line objects and add them to the drawing.
- Line 74 ends the `With` statement block.
- Lines 75 and 76 end the two `For` loops.
- Line 77 ends the `DrawRangeTop` macro.

Summary

After reading this chapter, you'll know how to do the following tasks and understand the related concepts in VBA code:

- Draw circles.
- Access points and distances specified in the AutoCAD window from a macro.
- Fill a circle with color.
- Draw a circle of bricks from an application.
- Set the boundaries for hatching outer loops and inner loops.

- Fill nested shapes using the Island Detection style.
- Use the Associativity argument of the AddHatch method to determine whether or not the hatch pattern gets updated with the boundary.
- Draw any shape, fill it with a hatch pattern, and find out its area.
- Add Hatch objects to the ModelSpace collection.
- Use the Variant type as a means of passing a three-element array in one assignment statement.
- Add items to a combo box and determine the one selected by the user at run time.
- Use the Input Box function to receive input from the user.
- Place an array of objects into a rectangular area in the Model Space, using a macro to calculate the size and positions.