Learn Visual Basic 6.0

7. Graphics Techniques with Visual Basic

Review and Preview

- In past classes, we've used some graphics tools: line tools, shape tools, image boxes, and picture boxes. In this class, we extend our graphics programming skills to learn how to draw lines and circles, do drag and drop, perform simple animation, and study some basic plotting routines.

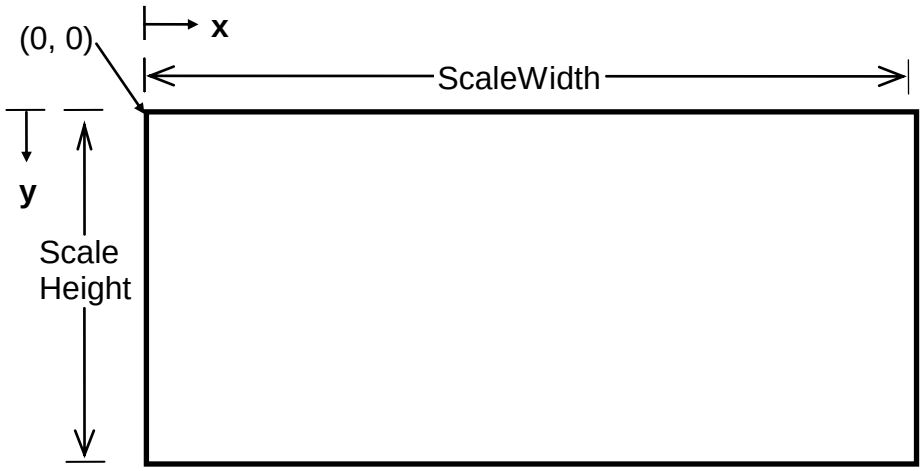
Graphics Methods

- **Graphics methods** apply to forms and picture boxes (remember a picture box is like a form within a form). With these methods, we can draw lines, boxes, and circles. Before discussing the commands that actually perform the graphics drawing, though, we need to look at two other topics: **screen management** and **screen coordinates**.

- In single program environments (DOS, for example), when something is drawn on the screen, it stays there. Windows is a multi-tasking environment. If you switch from a Visual Basic application to some other application, your Visual Basic form may become partially obscured. When you return to your Visual Basic application, you would like the form to appear like it did before being covered. All controls are automatically restored to the screen. Graphics methods drawings may or may not be restored - we need them to be, though. To accomplish this, we must use proper **screen management**.

- The simplest way to maintain graphics is to set the form or picture box's **AutoRedraw** property to True. In this case, Visual Basic always maintains a copy of graphics output in memory (creates **persistent graphics**). Another way to maintain drawn graphics is (with AutoRedraw set to False) to put all graphics commands in the form or picture box's **Paint** event. This event is called whenever an obscured object becomes unobscured. There are advantages and disadvantages to both approaches (beyond the scope of discussion here). For now, we will assume our forms won't get obscured and, hence, beg off the question of persistent graphics and using the AutoRedraw property and/or Paint event.

- All graphics methods described here will use the **default coordinate system**:



Note the **x** (horizontal) coordinate runs from left to right, starting at **0** and extending to **ScaleWidth - 1**. The **y** (vertical) coordinate goes from top to bottom, starting at **0** and ending at **ScaleHeight - 1**. Points in this coordinate system will always be referred to by a Cartesian pair, **(x, y)**. Later, we will see how we can use any coordinate system we want.

ScaleWidth and ScaleHeight are object properties representing the “graphics” dimensions of an object. Due to border space, they are not the same as the Width and Height properties. For all measurements in twips (default coordinates), ScaleWidth is less than Width and ScaleHeight is less than Height. That is, we can’t draw to all points on the form.

- PSet Method:

To set a single point in a graphic object (form or picture box) to a particular color, use the **PSet** method. We usually do this to designate a starting point for other graphics methods. The syntax is:

```
ObjectName.PSet (x, y), Color
```

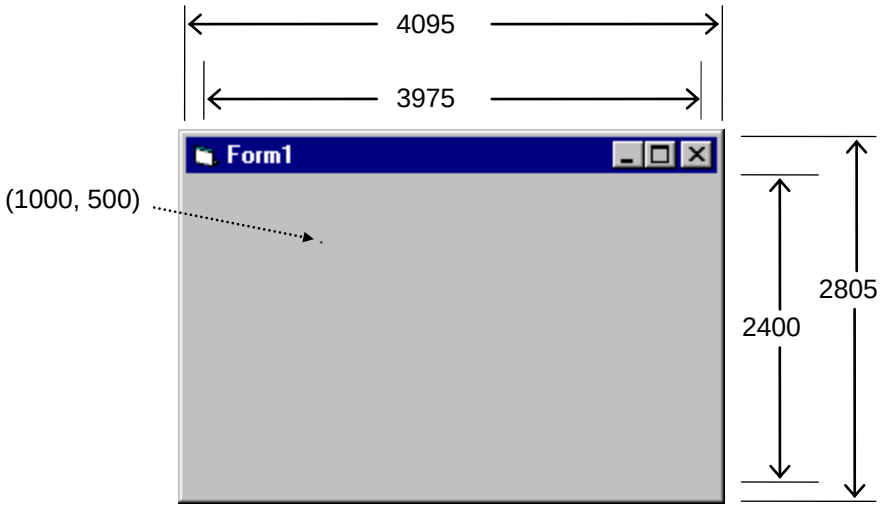
where **ObjectName** is the object name, **(x, y)** is the selected point, and **Color** is the point color (discussed in the next section). If the ObjectName is omitted, the current form is assumed to be the object. If Color is omitted, the object's **ForeColor** property establishes the color. PSet is usually used to initialize some further drawing process.

- Pset Method Example:

  This form has a ScaleWidth of 3975 (Width 4095) and a ScaleHeight of 2400 (Height 2805). The command:

  ```
  PSet (1000, 500)
  ```

  will have the result:



  The marked point (in color **ForeColor,** black in this case) is pointed to by the Cartesian coordinate (1000, 500) - this marking, of course, does not appear on the form. If you want to try this example, and the other graphic methods, put the code in the Form_Click event. Run the project and click on the form to see the results (necessary because of the AutoRedraw problem).

- CurrentX and CurrentY:

  After each drawing operation, the coordinate of the last point drawn to is maintained in two Visual Basic system variables, **CurrentX** and **CurrentY**. This way we always know where the next drawing operation will begin. We can also change the values of these variables to move this last point. For example, the code:

  ```
  CurrentX = 1000
  CurrentY = 500
  ```

  is equivalent to:

  ```
  PSet(1000, 500)
  ```

- Line Method:

  The **Line** method is very versatile. We can use it to draw line segments, boxes, and filled boxes. To draw a line, the syntax is:

  ObjectName.Line (x1, y1) - (x2, y2), Color

  where **ObjectName** is the object name, **(x1, y1)** the starting coordinate, **(x2, y2)** the ending coordinate, and **Color** the line color. Like PSet, if ObjectName is omitted, drawing is done to the current form and, if Color is omitted, the object's **ForeColor** property is used.

  To draw a line from (CurrentX, CurrentY) to (x2, y2), use:

  ObjectName.Line - (x2, y2), Color

  There is no need to specify the start point since CurrentX and CurrentY are known.

  To draw a box bounded by opposite corners (x1, y1) and (x2, y2), use:

  ObjectName.Line (x1, y1) - (x2, y2), Color, B

  and to fill that box (using the current **FillPattern**), use:

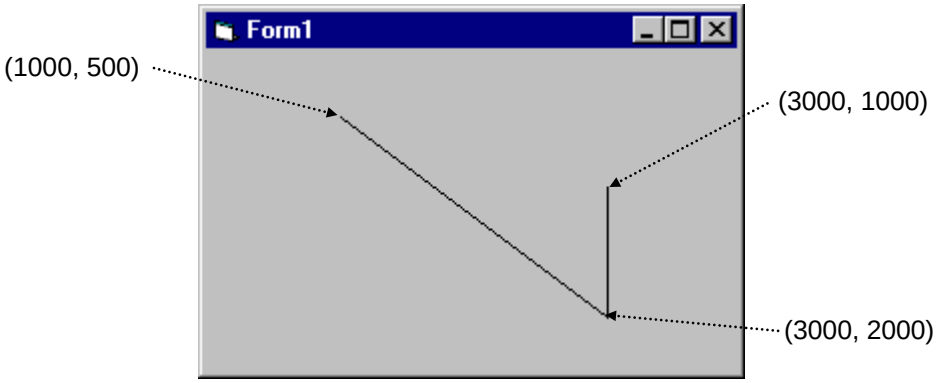
  ObjectName.Line (x1, y1) - (x2, y2), Color, BF

- Line Method Examples:

  Using the previous example form, the commands:

  ```
  Line (1000, 500) - (3000, 2000)
  Line - (3000, 1000)
  ```

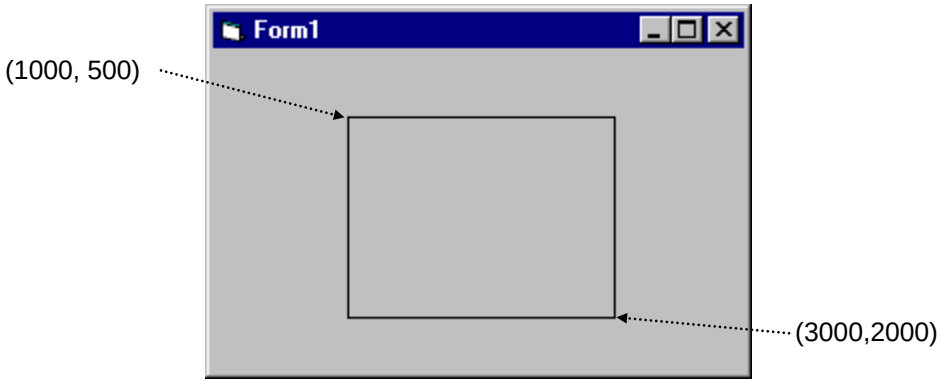
  draws these line segments:



  The command:

  ```
  Line (1000, 500) - (3000, 2000), , B
  ```

  draws this box (note two commas after the second coordinate - no color is specified):



- Circle Method:

  The **Circle** method can be used to draw circles, ellipses, arcs, and pie slices. We'll only look at drawing circles - look at on-line help for other drawing modes. The syntax is:

  ObjectName.Circle (x, y), r, Color

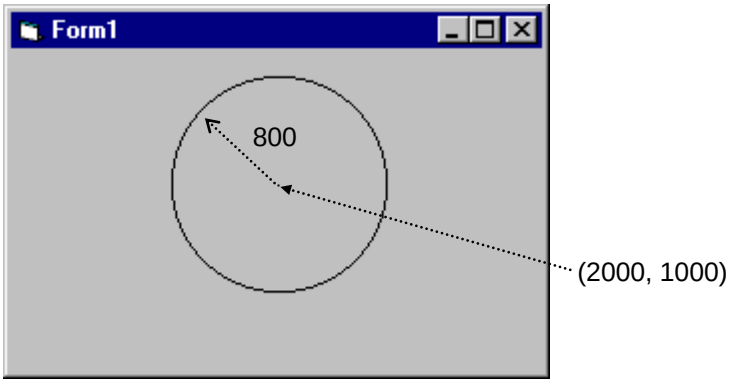
  This command will draw a circle with center **(x, y)** and radius **r**, using **Color**.

- Circle Example:

  With the same example form, the command:

  Circle (2000, 1000), 800

  produces the result:



- Print Method:

  Another method used to 'draw' to a form or picture box is the **Print** method. Yes, for these objects, printed text is drawn to the form. The syntax is:

  ObjectName.Print [information to print]

  Here the printed information can be variables, text, or some combination. If no object name is provided, printing is to the current form.

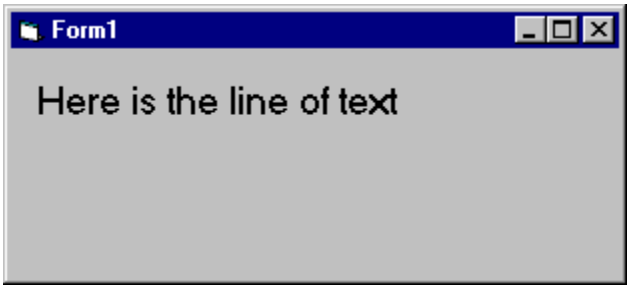
  Information will print beginning at the object's **CurrentX** and **CurrentY** value. The color used is specified by the object's **ForeColor** property and the font is specified by the object's **Font** characteristics.

- Print Method Example:

  The code (can't be in the Form_Load procedure because of that pesky AutoRedraw property):

  ```
  CurrentX=200
  CurrentY=200
  Print "Here is the line of text"
  ```

  will produce this result (I've used a large font):



- Cls Method:

  To clear the graphics drawn to an object, use the **Cls** method. The syntax is:

  ```
  ObjectName.Cls
  ```

  If no object name is given, the current form is cleared. Recall Cls only clears the lowest of the three display layers. This is where graphics methods draw.

- For each graphic method, line widths, fill patterns, and other graphics features can be controlled via other object properties. Consult on-line help for further information.

Using Colors

- Notice that all the graphics methods can use a **Color** argument. If that argument is omitted, the **ForeColor** property is used. Color is actually a hexadecimal (long integer) representation of color - look in the Properties Window at some of the values of color for various object properties. So, one way to get color values is to cut and paste values from the Properties Window. There are other ways, though.

- Symbolic Constants:

  Visual Basic offers eight **symbolic constants** (see Appendix I) to represent some basic colors. Any of these constants can be used as a **Color** argument.

Constant	Value	Color
vbBlack	0x0	Black
vbRed	0xFF	Red
vbGreen	0xFF00	Green
vbYellow	0xFFFF	Yellow
vbBlue	0xFF0000	Blue
vbMagenta	0xFF00FF	Magenta
vbCyan	0xFFFF00	Cyan
vbWhite	0xFFFFFF	White

- QBColor Function:

  For Microsoft QBasic, GW-Basic and QuickBasic programmers, Visual Basic replicates the sixteen most used colors with the **QBColor** function. The color is specified by QBColor(Index), where the colors corresponding to the Index are:

Index	Color	Index	Color
0	Black	8	Gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	White	15	Light (bright) white

- RGB Function:

  The **RGB** function can be used to produce one of 2^{24} (over 16 million) colors! The syntax for using RGB to specify the color property is:

  ```
  RGB(Red, Green, Blue)
  ```

  where **Red**, **Green**, and **Blue** are integer measures of intensity of the corresponding primary colors. These measures can range from 0 (least intensity) to 255 (greatest intensity). For example, RGB(255, 255, 0) will produce yellow.

- Any of these four representations of color can be used anytime your Visual Basic code requires a color value.

- Color Examples:

  ```
  frmExample.BackColor = vbGreen
  picExample.FillColor = QBColor(3)
  lblExample.ForeColor = RGB(100, 100, 100)
  ```

Mouse Events

- Related to graphics methods are **mouse events**. The mouse is a primary interface to performing graphics in Visual Basic. We've already used the mouse to **Click** and **DblClick** on objects. Here, we see how to recognize other mouse events to allow drawing in forms and picture boxes.

- MouseDown Event:

  The **MouseDown** event procedure is triggered whenever a mouse button is pressed while the mouse cursor is over an object. The form of this procedure is:

```
Sub ObjectName_MouseDown(Button As Integer, Shift As Integer, X As
Single, Y As Single)
        .
        .
End Sub
```

  The arguments are:

Button	Specifies which mouse button was pressed.
Shift	Specifies state of Shift, Ctrl, and Alt keys.
X, Y	Coordinate of mouse cursor when button was pressed.

  Values for the Button argument are:

Symbolic Constant	Value	Description
vbLeftButton	1	Left button is pressed.
vbRightButton	2	Right button is pressed.
vbMiddleButton	4	Middle button is pressed.

  Only one button press can be detected by the MouseDown event. Values for the Shift argument are:

Symbolic Constant	Value	Description
vbShiftMask	1	Shift key is pressed.
vbCtrlMask	2	Ctrl key is pressed.
vbAltMask	4	Alt key is pressed.

  The Shift argument can represent multiple key presses. For example, if Shift = 5 (vbShiftMask + vbAltMask), both the Shift and Alt keys are being pressed when the MouseDown event occurs.

- MouseUp Event:

  The **MouseUp** event is the opposite of the MouseDown event. It is triggered whenever a previously pressed mouse button is released. The procedure outline is:

```
Sub ObjectName_MouseUp(Button As Integer, Shift As Integer, X As
Single, Y As Single)
        .
        .
End Sub
```

  The arguments are:

Button	Specifies which mouse button was released.
Shift	Specifies state of Shift, Ctrl, and Alt keys.
X, Y	Coordinate of mouse cursor when button was released.

  The **Button** and **Shift** constants are the same as those for the MouseDown event.

- MouseMove Event:

  The **MouseMove** event is continuously triggered whenever the mouse is being moved. The procedure outline is:

```
Sub ObjectName_MouseMove(Button As Integer, Shift As Integer, X As
Single, Y As Single)
        .
        .
End Sub
```

  The arguments are:

Button	Specifies which mouse button(s), if any, are pressed.
Shift	Specifies state of Shift, Ctrl, and Alt keys
X, Y	Current coordinate of mouse cursor

The **Button** and **Shift** constants are the same as those for the MouseDown event. A difference here is that the Button argument can also represent multiple button presses or no press at all. For example, if Button = 0, no button is pressed as the mouse is moved. If Button = 3 (vbLeftButton + vbRightButton), both the left and right buttons are pressed while the mouse is being moved.

Example 7-1

Blackboard

1. Start a new application. Here, we will build a blackboard we can scribble on with the mouse (using colored 'chalk').

2. Set up a simple menu structure for your application using the Menu Editor. The menu should be:

   File
   - New
   - ———
   - Exit

   Properties for these menu items should be:

Caption	**Name**
&File	mnuFile
&New	mnuFileNew
-	mnuFileSep
E&xit	mnuFileExit

3. Put a picture box and a single label box (will be used to set color) on the form. Set the following properties:

   Form1:

BorderStyle	1-Fixed Single
Caption	Blackboard
Name	frmDraw

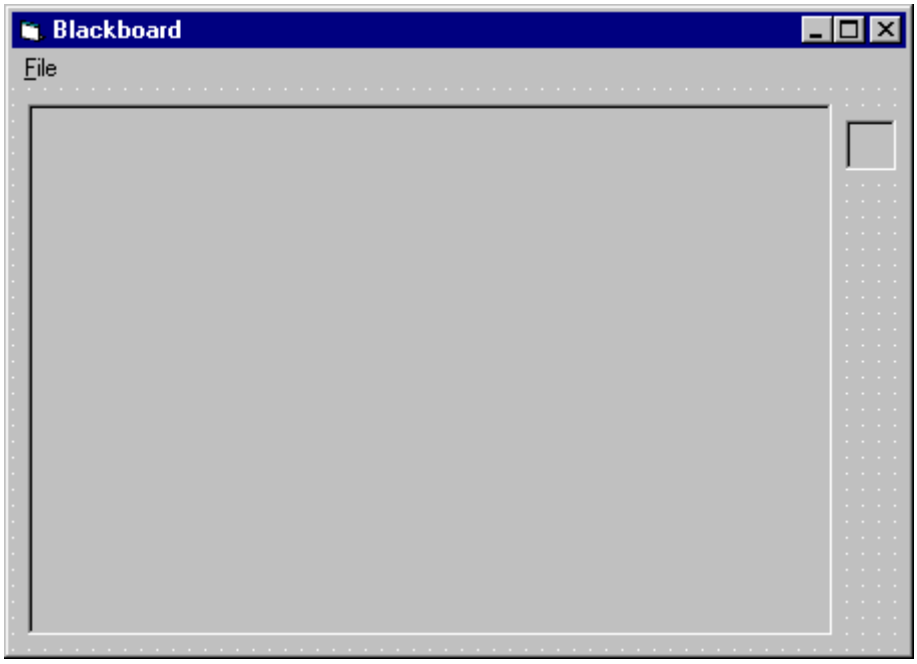
   Picture1:

Name	picDraw

   Label1:

BorderStyle	1-Fixed Single
Caption	[Blank]
Name	lblColor

The form should look something like this:



4. Now, copy and paste the label box (create a control array named **lblColor**) until there are eight boxes on the form, lined up vertically under the original box. When done, the form will look just as above, except there will be eight label boxes.

5. Type these lines in the **general declarations** area. **DrawOn** will be used to indicate whether you are drawing or not.

```
Option Explicit
Dim DrawOn As Boolean
```

6. Attach code to each procedure.

   The **Form_Load** procedure loads colors into each of the label boxes to allow choice of drawing color. It also sets the **BackColor** to black and the **ForeColor** to Bright White.

```
Private Sub Form_Load()
'Load drawing colors into control array
Dim I As Integer
For I = 0 To 7
  lblColor(I).BackColor = QBColor(I + 8)
Next I
picDraw.ForeColor = QBColor(15) ' Bright White
picDraw.BackColor = QBColor(0) ' Black
End Sub
```

   In the **mnuFileNew_Click** procedure, we check to see if the user really wants to start over. If so, the picture box is cleared with the **Cls** method.

```
Private Sub mnuFileNew_Click()
'Make sure user wants to start over
Dim Response As Integer
Response = MsgBox("Are you sure you want to start a new
drawing?", vbYesNo + vbQuestion, "New Drawing")
If Response = vbYes Then picDraw.Cls
End Sub
```

   In the **mnuFileExit_Click** procedure, make sure the user really wants to stop the application.

```
Private Sub mnuFileExit_Click()
'Make sure user wants to quit
Dim Response As Integer
Response = MsgBox("Are you sure you want to exit the
Blackboard?", vbYesNo + vbCritical + vbDefaultButton2,
"Exit Blackboard")
If Response = vbYes Then End
End Sub
```

When the left mouse button is clicked, drawing is initialized at the mouse cursor location in the **picDraw_MouseDown** procedure.

```
Private Sub picDraw_MouseDown(Button As Integer, Shift As
Integer, X As Single, Y As Single)
'Drawing begins
If Button = vbLeftButton Then
DrawOn = True
  picDraw.CurrentX = X
  picDraw.CurrentY = Y
End If
End Sub
```

When drawing ends, the **DrawOn** switch is toggled in **picDraw_MouseUp**.

```
Private Sub picDraw_MouseUp(Button As Integer, Shift As
Integer, X As Single, Y As Single)
'Drawing ends
If Button = vbLeftButton Then DrawOn = False
End Sub
```

While mouse is being moved and **DrawOn** is True, draw lines in current color in the **picDraw_MouseMove** procedure.

```
Private Sub picDraw_MouseMove(Button As Integer, Shift As
Integer, X As Single, Y As Single)
'Drawing continues
If DrawOn Then picDraw.Line -(X, Y), picDraw.ForeColor
End Sub
```

Finally, when a label box is clicked, the drawing color is changed in the **lblColor_Click** procedure.

```
Private Sub lblColor_Click(Index As Integer)
'Make audible tone and reset drawing color
Beep
picDraw.ForeColor = lblColor(Index).BackColor
End Sub
```

7. Run the application. Click on the label boxes to change the color you draw with. Fun, huh? Save the application.

8. A challenge for those who like challenges. Add **Open** and **Save** options that allow you to load and save pictures you draw. Suggested steps (may take a while - I suggest trying it outside of class):

   A. Change the picture box property **AutoRedraw** to True. This is necessary to save pictures. You will notice the drawing process slows down to accommodate **persistent graphics**.

   B. Add the **Open** option. Write code that brings up a common dialog box to get a filename to open (will be a .bmp file) and put that picture in the picDraw.Picture property using the **LoadPicture** function.

   C. Add the **Save** option. Again, add code to use a common dialog box to get a proper filename. Use the **SavePicture** method to save the **Image** property of the picDraw object. We save the Image property, not the Picture property, since this is where Visual Basic maintains the persistent graphics.

   D. One last change. The Cls method in the **mnuFileNew_Click** code will not clear a picture loaded in via the Open code (has to do with using AutoRedraw). So, replace the Cls statement with code that manually erases the picture box. I'd suggest using the **BF** option of the **Line** method to simply fill the space with a box set equal to the BackColor (white). I didn't say this would be easy.

Drag and Drop Events

- Related to mouse events are **drag and drop events**. This is the process of using the mouse to pick up some object on a form and move it to another location. We use drag and drop all the time in Visual Basic design mode to locate objects on our application form.

- Drag and drop allows you to design a simple user interface where tasks can be performed without commands, menus, or buttons. Drag and drop is very intuitive and, at times, faster than other methods. Examples include dragging a file to another folder or dragging a document to a printer queue.

- Any Visual Basic object can be dragged and dropped, but we usually use **picture** and **image** boxes. The item being dragged is called the **source** object. The item being dropped on (if there is any) is called the **target**.

- Object Drag Properties:

  If an object is to be dragged, two properties must be set:

DragMode	Enables dragging of an object (turns off ability to receive Click or MouseDown events). Usually use 1-Automatic (**vbAutomatic**).
DragIcon	Specifies icon to display as object is being dragged.

  As an object is being dragged, the object itself does not move, only the DragIcon. To move the object, some additional code using the **Move** method (discussed in a bit) must be used.

- DragDrop Event:

  The **DragDrop** event is triggered whenever the source object is dropped on the target object. The procedure form is:

```
Sub ObjectName_DragDrop(Source As Control, X As Single, Y As Single)
        .
        .
End Sub
```

  The arguments are:

Source	Object being dragged.
X, Y	Current mouse cursor coordinates.

- DragOver Event:

  The **DragOver** event is triggered when the source object is dragged over another object. Its procedure form is:

```
Private Sub ObjectName_DragOver(Source As Control, X As Single,
Y As Single, State As Integer)
        .
        .
End Sub
```

  The first three arguments are the same as those for the DragDrop event. The **State** argument tells the object where the source is. Its values are 0-Entering (**vbEnter**), 1-Leaving (**vbLeave**), 2-Over (**vbOver**).

- Drag and Drop Methods:

Drag	Starts or stops manual dragging (won't be addressed here - we use Automatic dragging)
Move	Used to move the source object, if desired.

  Example

  To move the source object to the location specified by coordinates X and Y, use:

```
Source.Move X, Y
```

  The best way to illustrate the use of drag and drop is by example.

Example 7-2

Letter Disposal

1. We'll build a simple application of drag and drop where unneeded correspondence is dragged and dropped into a trash can. Start a new application. Place four image boxes and a single command button on the form. Set these properties:

Form1:

BackColor	White
BorderStyle	1-Fixed Single
Caption	Letter Disposal
Name	frmDispose

Command1:

Caption	&Reset
Name	cmdReset

Image1:

Name	imgCan
Picture	trash01.ico
Stretch	True

Image2:

Name	imgTrash
Picture	trash01.ico
Visible	False

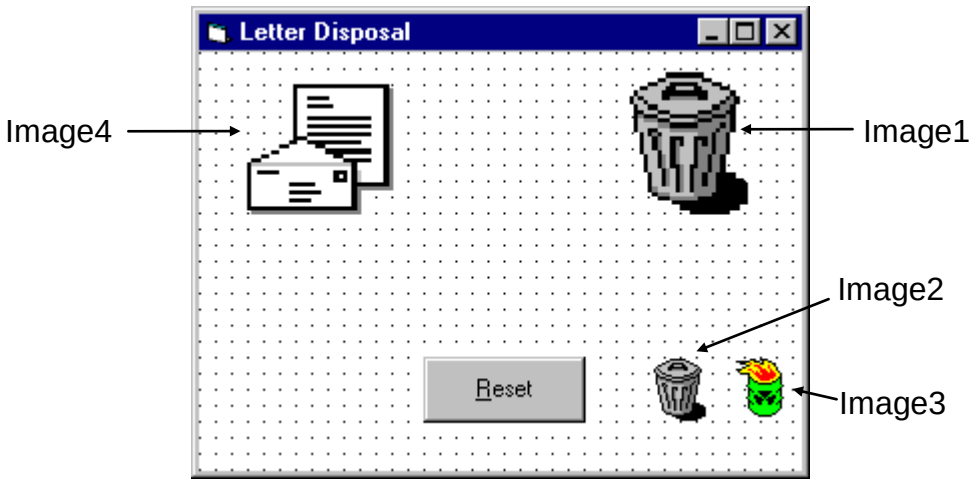
Image3:

Name	imgBurn
Picture	trash02b.ico
Visible	False

Image4:

DragIcon	drag1pg.ico
DragMode	1-Automatic
Name	imgLetter
Picture	mail06.ico
Stretch	True

The form will look like this:



Some explanation about the images on this form is needed. The letter image is the control to be dragged and the trash can (at **Image1** location) is where it will be dragged to. The additional images (the other trash can and burning can) are not visible at run-time and are used to change the state of the trash can, when needed. We could load these images from disk files at run-time, but it is much quicker to place them on the form and hide them, then use them when required.

2. The code here is minimal. The **Form_DragDrop** event simply moves the letter image if it is dropped on the form.

```
Private Sub Form_DragDrop(Source As Control, X As Single,
Y As Single)
Source.Move X, Y
End Sub
```

3. The **imgCan_DragDrop** event changes the trash can to a burning pyre if the letter is dropped on it.

```
Private Sub imgCan_DragDrop(Index As Integer, Source As
Control, X As Single, Y As Single)
'Burn mail and make it disappear
imgCan.Picture = imgBurn.Picture
Source.Visible = False
End Sub
```

4. The **cmdReset_Click** event returns things to their original state.

```
Private Sub cmdReset_Click()
'Reset to trash can picture
imgCan.Picture = imgTrash.Picture
imgLetter.Visible = True
End Sub
```

5. Save and run the application. Notice how only the drag icon moves. Notice the letter moves once it is dropped. Note, too, that the letter can be dropped anywhere. The fire appears only when it is dropped in the trash.

Timer Tool and Delays

- Many times, especially in using graphics, we want to repeat certain operations at regular intervals. The **timer tool** allows such repetition. The timer tool does not appear on the form while the application is running.

- Timer tools work in the background, only being invoked at time intervals you specify. This is multi-tasking - more than one thing is happening at a time.

- Timer Properties:

Enabled	Used to turn the timer on and off. When on, it continues to operate until the Enabled property is set to False.
Interval	Number of milliseconds between each invocation of the Timer Event.

- Timer Events:

  The timer tool only has one event, **Timer**. It has the form:

```
Sub TimerName_Timer()
        .
        .
End Sub
```

  This is where you put code you want repeated every **Interval** seconds.

- Timer Example:

  To make the computer beep every second, no matter what else is going on, you add a timer tool (named **timExample**) to the form and set the **Interval** property to 1000. That timer tool's event procedure is then:

```
Sub timExample_Timer()
Beep
End Sub
```

- In complicated applications, many timer tools are often used to control numerous simultaneous operations. With experience, you will learn the benefits and advantages of using timer tools.

- Simple Delays:

  If you just want to use a simple delay in your Visual Basic application, you might want to consider the **Timer** function. This is not related to the Timer tool. The Timer function simply returns the number of seconds elapsed since midnight.

  To use the **Timer** function for a delay of **Delay** seconds (the Timer function seems to be accurate to about 0.1 seconds, at best), use this code segment:

```
Dim TimeNow As Single
        .
        .
TimeNow = Timer
Do While Timer - TimeNow < Delay
Loop
```

  One drawback to this kind of coding is that the application cannot be interrupted while in the Do loop. So, keep delays to small values.

Animation Techniques

- One of the more fun things to do with Visual Basic programs is to create animated graphics. We'll look at a few simple **animation** techniques here. I'm sure you'll come up with other ideas for animating your application.

- One of the simplest animation effects is achieved by **toggling** between two **images**. For example, you may have a picture of a stoplight with a red light. By quickly changing this picture to one with a green light, we achieve a dynamic effect - animation. **Picture** boxes and **image** boxes are used to achieve this effect.

- Another approach to animation is to **rotate** through several pictures - each a slight change in the previous picture - to obtain a longer animation. This is the principle motion pictures are based on - pictures are flashed by us at 24 frames per second and our eyes are tricked into believing things are smoothly moving. **Control arrays** are usually used to achieve this type of animation.

- More elaborate effects can be achieved by moving an image while, at the same, time changing the displayed picture. Effects such as a little guy walking across the screen are easily achieved. An object is moved using the **Move** method. You can do both absolute and relative motion (using an object's **Left** and **Top** properties).

For example, to move a picture box named **picExample** to the coordinate (100, 100), use:

```
picExample.Move 100, 100
```

To move it 20 twips to the right and 50 twips down, use:

```
picExample.Move picExample.Left + 20, picExample.Top + 50
```

Quick Example: Simple Animation

1. Start a new application. Place three image boxes on the form. Set the following properties:

   Image1:

Picture	mail02a.ico
Visible	False

   Image2:

Picture	mail02b.ico
Visible	False

   Image3:

Picture	mail02a.ico
Stretch	True

   Make Image3 larger than default size, using the 'handles.'

   A few words about what we're going to do. **Image1** holds a closed envelope, while **Image2** holds an opened one. These images are not visible - they will be selected for display in **Image3** (which is visible) as Image3 is clicked. (This is similar to hiding things in the drag and drop example.) It will seem the envelope is being torn opened, then repaired.

2. Attach the following code to the **Image3_Click** procedure.

```
Private Sub Image3_Click()
Static PicNum As Integer
If PicNum = 0 Then
  Image3.Picture = Image2.Picture : PicNum = 1
Else
  Image3.Picture = Image1.Picture : PicNum = 0
End If
End Sub
```

When the envelope is clicked, the image displayed in **Image3** is toggled (based on the value of the static variable **PicNum**).

3. Run and save the application.

Quick Example: Animation with the Timer Tool

1. In this example, we cycle through four different images using timer controlled animation. Start a new application. Put two image boxes, a timer tool, and a command button on the form. Set these properties:

Image1:

Picture	trffc01.ico
Visible	False

Now copy and paste this image box three times, so there are four elements in the **Image1** control array. Set the **Picture** properties of the other three elements to:

Image1(1):

Picture	trffc02.ico

Image1(2):

Picture	trffc03.ico

Image1(3):

Picture	trffc04.ico

Image2:

Picture	trffc01.ico
Stretch	True

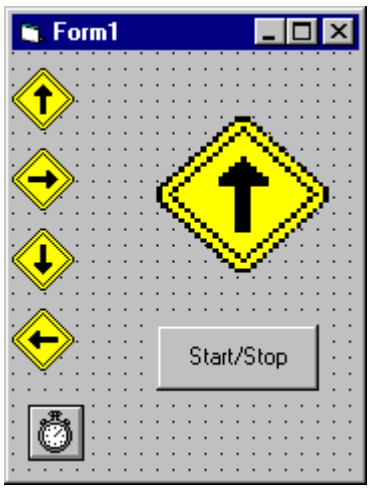
Command1:

Caption	Start/Stop

Timer1:

Enabled	False
Interval	200

The form should resemble this:



2. Attach this code to the **Command1_Click** procedure.

```
Private Sub Command1_Click()
Timer1.Enabled = Not (Timer1.Enabled)
End Sub
```

The timer is turned on or off each time this code is invoked.

3. Attach this code to the **Timer1_Timer** procedure.

```
Private Sub Timer1_Timer()
Static PicNum As Integer
PicNum = PicNum + 1
If PicNum > 3 Then PicNum = 0
Image2.Picture = Image1(PicNum).Picture
End Sub
```

This code changes the image displayed in the **Image2** box, using the static variable **PicNum** to keep track of what picture is next.

4. Save and run the application. Note how the timer tool and the four small icons do not appear on the form at run-time. The traffic sign appears to be spinning, with the display updated by the timer tool every 0.2 seconds (200 milliseconds).

5. You can make the sign 'walk off' one side of the screen by adding this line after setting the Picture property:

```
Image2.Move Image2.Left + 150
```

Random Numbers (Revisited) and Games

- Another fun thing to do with Visual Basic is to create **games**. You can write games that you play against the computer or against another opponent.

- To introduce chaos and randomness in games, we use **random numbers**. Random numbers are used to have the computer roll a die, spin a roulette wheel, deal a deck of cards, and draw bingo numbers. Visual Basic develops random numbers using its built-in **random number generator**.

- Randomize Statement:

  The random number generator in Visual Basic must be seeded. A **Seed** value initializes the generator. The **Randomize** statement is used to do this:

  ```
  Randomize Seed
  ```

  If you use the same Seed each time you run your application, the same sequence of random numbers will be generated. To insure you get different numbers every time you use your application (preferred for games), use the **Timer** function to seed the generator:

  ```
  Randomize Timer
  ```

  With this, you will always obtain a different sequence of random numbers, unless you happen to run the application at exactly the same time each day.

- Rnd Function:

  The Visual Basic function **Rnd** returns a single precision, random number between 0 and 1 (actually greater than or equal to 0 and less than 1). To produce random integers (I) between Imin and Imax (again, what we usually do in games), use the formula:

  ```
  I = Int((Imax - Imin + 1) * Rnd) + Imin
  ```

- Rnd Example:

  To roll a six-sided die, the number of spots would be computed using:

  ```
  NumberSpots = Int(6 * Rnd) + 1
  ```

  To randomly choose a number between 100 and 200, use:

  ```
  Number = Int(101 * Rnd) + 100
  ```

Randomly Sorting N Integers

- In many games, we have the need to randomly sort a number of integers. For example, to shuffle a deck of cards, we sort the integers from 1 to 52. To randomly sort the state names in a states/capitals game, we would randomize the values from 1 to 50.

- Randomly sorting N integers is a common task. Here is a 'self-documenting' general procedure that does that task. Calling arguments for the procedure are **N** (the largest integer to be sorted) and an array, **NArray**, dimensioned to N elements. After calling the routine **N_Integers**, the N randomly sorted integers are returned in NArray. Note the procedure randomizes the integers from 1 to N, not 0 to N - the zeroth array element is ignored.

```
Private Sub N_Integers(N As Integer, Narray() As Integer)
'Randomly sorts N integers and puts results in Narray
Dim I As Integer, J As Integer, T As Integer
'Order all elements initially
For I = 1 To N: Narray(I) = I: Next I
'J is number of integers remaining
For J = N to 2 Step -1
  I = Int(Rnd * J) + 1
  T = Narray(J)
  Narray(J) = Narray(I)
  Narray(I) = T
Next J
End Sub
```

Example 7-3

One-Buttoned Bandit

1. Start a new application. In this example, we will build a computer version of a slot machine. We'll use random numbers and timers to display three random pictures. Certain combinations of pictures win you points. Place two image boxes, two label boxes, and two command buttons on the form.

2. Set the following properties:

Form1:

BorderStyle	1-Fixed Single
Caption	One-Buttoned Bandit
Name	frmBandit

Command1:

Caption	&Spin It
Default	True
Name	cmdSpin

Command2:

Caption	E&xit
Name	cmdExit

Timer1:

Enabled	False
Interval	100
Name	timSpin

Timer2:

Enabled	False
Interval	2000
Name	timDone

Label1:

Caption	Bankroll
FontBold	True
FontItalic	True
FontSize	14

Label2:

Alignment	2-Center
AutoSize	True
BorderStyle	1-Fixed Single
Caption	100
FontBold	True
FontSize	14
Name	lblBank

Image1:

Name	imgChoice
Picture	earth.ico
Visible	False

Copy and paste this image box three times, creating a control element (**imgChoice**) with four elements total. Set the **Picture** property of the other three boxes.

Image1(1):

Picture	snow.ico

Image1(2):

Picture	misc44.ico

Image1(3):

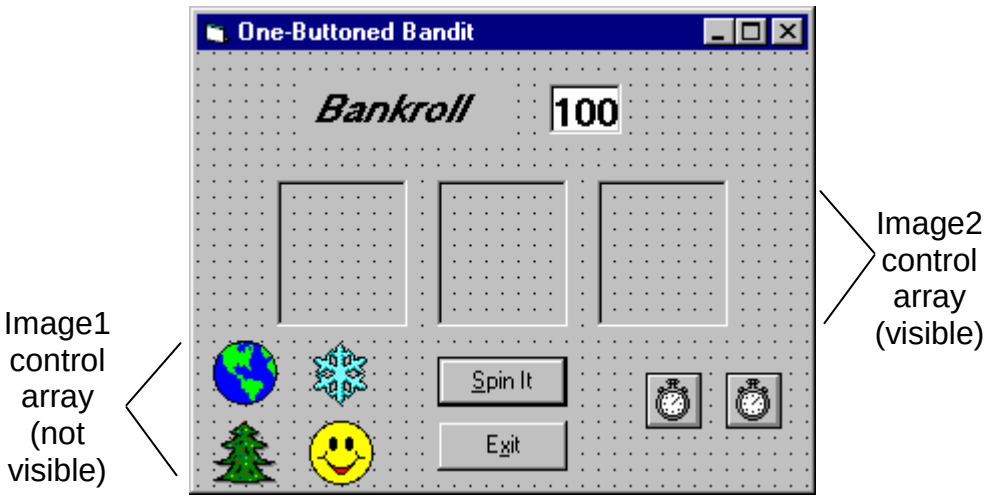
Picture	face03.ico

Image2:

BorderStyle	1-Fixed single
Name	imgBandit
Stretch	True

Copy and paste this image box two times, creating a three element control array (**Image2**). You don't have to change any properties of the newly created image boxes.

When done, the form should look something like this:



A few words on what we're doing. We will randomly fill the three large image boxes by choosing from the four choices in the non-visible image boxes. One timer (**timSpin**) will be used to flash pictures in the boxes. One timer (**timDone**) will be used to time the entire process.

3. Type the following lines in the **general declarations** area of your form's code window. **Bankroll** is your winnings.

```
Option Explicit
Dim Bankroll As Integer
```

4. Attach this code to the **Form_Load** procedure.

```
Private Sub Form_Load()
Randomize Timer
Bankroll = Val(lblBank.Caption)
End Sub
```

Here, we seed the random number generator and initialize your bankroll.

5. Attach the following code to the **cmdExit_Click** event.

```
Private Sub cmdExit_Click()
MsgBox "You ended up with" + Str(Bankroll) + " points.",
vbOKOnly, "Game Over"
End
End Sub
```

When you exit, your final earnings are displayed in a message box.

6. Attach this code to the **cmdSpin_Click** event.

```
Private Sub cmdSpin_Click()
If Bankroll = 0 Then
  MsgBox "Out of Cash!", vbOKOnly, "Game Over"
  End
End If
Bankroll = Bankroll - 1
lblBank.Caption = Str(Bankroll)
timSpin.Enabled = True
timDone.Enabled = True
End Sub
```

   Here, we first check to see if you're out of cash. If so, the game ends. If not, you are charged 1 point and the timers are turned on.

7. This is the code for the **timSpin_Timer** event.

```
Private Sub timSpin_Timer()
imgBandit(0).Picture = imgChoice(Int(Rnd * 4)).Picture
imgBandit(1).Picture = imgChoice(Int(Rnd * 4)).Picture
imgBandit(2).Picture = imgChoice(Int(Rnd * 4)).Picture
End Sub
```

   Every 0.1 seconds, the three visible image boxes are filled with a random image. This gives the effect of the spinning slot machine.

8. And, the code for the **timDone_Timer** event. This event is triggered after the bandit spins for 2 seconds.

```
Private Sub timDone_Timer()
Dim P0 As Integer, P1 As Integer, P2 As Integer
Dim Winnings As Integer
Const FACE = 3
timSpin.Enabled = False
timDone.Enabled = False
P0 = Int(Rnd * 4)
P1 = Int(Rnd * 4)
P2 = Int(Rnd * 4)
imgBandit(0).Picture = imgChoice(P0).Picture
imgBandit(1).Picture = imgChoice(P1).Picture
imgBandit(2).Picture = imgChoice(P2).Picture
```

```
If P0 = FACE Then
  Winnings = 1
  If P1 = FACE Then
    Winnings = 3
    If P2 = FACE Then
      Winnings = 10
    End If
  End If
ElseIf P0 = P1 Then
  Winnings = 2
  If P1 = P2 Then Winnings = 4
End If
Bankroll = Bankroll + Winnings
lblBank.Caption = Str(Bankroll)
End Sub
```

First, the timers are turned off. Final pictures are displayed in each position. Then, the pictures are checked to see if you won anything.

9. Save and run the application. See if you can become wealthy.

10. If you have time, try these things.

    A. Rather than display the three final pictures almost simultaneously, see if you can stop each picture from spinning at a different time. You'll need a few more **Timer** tools.

    B. Add some graphics and/or printing to the form when you win. You'll need to clear these graphics with each new spin - use the **Cls** method.

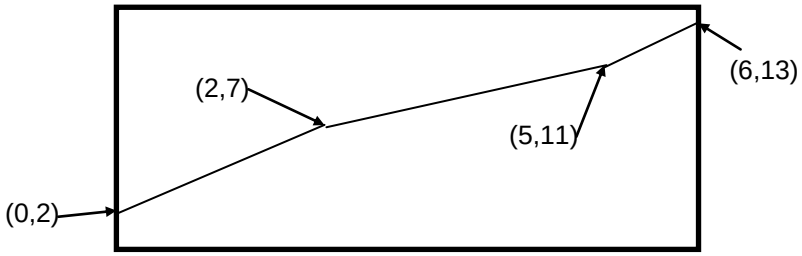
    C. See if you can figure out the logic I used to specify winning. See if you can show the one-buttoned bandit returns 95.3 percent of all the 'money' put in the machine. This is higher than what Vegas machines return. But, with truly random operation, Vegas is guaranteed their return. They can't lose!

User-Defined Coordinates

- Another major use for graphics in Visual Basic is to generate plots of data. **Line charts**, **bar charts**, and **pie charts** can all be easily generated.

- We use the **Line** tool and **Circle** tool to generate charts. The difficult part of using these tools is converting our data into the Visual Basic coordinate system. For example, say we wanted to plot the four points given by:

  x = 0, y = 2
  x = 2, y = 7
  x = 5, y = 11
  x = 6, y = 13



  To draw such a plot, for each point, we would need to scale each (x, y) pair to fit within the dimensions of the form specified by the **ScaleWidth** and **ScaleHeight** properties. This is a straightforward, but tedious computation.

- An easier solution lies in the ability to incorporate **user-defined coordinates** in a Visual Basic form. The simplest way to define such coordinates is with the **Scale** method. The form for this method is:

  ObjectName.Scale (x1, y1) - (x2, y2)

  The point **(x1, y1)** represents the **top left** corner of the newly defined coordinate system, while **(x2, y2)** represents the **lower right** corner. If **ObjectName** is omitted, the scaling is associated with the current form.

- Once the coordinate system has been redefined, all graphics methods must use coordinates in the new system. To return to the default coordinates, use the **Scale** method without any arguments.

- Scale Example:

  Say we wanted to plot the data from above. We would first define the following coordinate system:

  ```
  Scale (0, 13) - (6, 2)
  ```

  This shows that x ranges from 0 (left side of plot) to 6 (right side of plot), while y ranges from 2 (bottom of plot) to 13 (top of plot). The graphics code to plot this function is then:

  ```
  Pset (0, 2)
  Line - (2, 7)
  Line - (5, 11)
  Line - (6, 13)
  ```

  Note how much easier this is than would be converting each number pair to twips.

Simple Function Plotting (Line Charts)

- Assume we have a function specified by a known number of (x, y) pairs. Assume **N** points in two arrays dimensioned to N - 1: **x(N - 1)**, and **y(N - 1)**. Assume the points are sorted in the order they are to be plotted. Can we set up a general procedure to plot these functions, that is create a **line chart**? Of course!

- The process is:

  1. Go through all of the points and find the minimum x value (**Xmin**) , maximum x value (**Xmax**), minimum y value (**Ymin**) and the maximum y value (**Ymax**). These will be used to define the coordinate system. Extend each y extreme (Ymin and Ymax) a little bit - this avoids having a plotted point ending up right on the plot border.

  2. Define a coordinate system using **Scale**:

     ```
     Scale (Xmin, Ymax) - (Xmax, Ymin)
     ```

     Ymax is used in the first coordinate because, recall, it defines the **upper left** corner of the plot region.

3. Initialize the plotting procedure at the first point using **PSet**:

   PSet (x(0), y(0))

4. Plot subsequent points with the **Line** procedure:

   Line - (x(i), y(i))

- Here is a general procedure that does this plotting using these steps. It can be used as a basis for more elaborate plotting routines. The arguments are **ObjectName** the name of the object (form or picture box) you are plotting on, **N** the number of points, **X** the array of x points, and **Y** the array of y points.

```
Sub LineChart(ObjectName As Control, N As Integer, X() As Single, Y()
As Single)
Dim Xmin As Single, Xmax As Single
Dim Ymin As Single, Ymax As Single
Dim I As Integer
Xmin = X(0): Xmax = X(0)
Ymin = Y(0): Ymax = Y(0)
For I = 1 To N - 1
  If X(I) < Xmin Then Xmin = X(I)
  If X(I) > Xmax Then Xmax = X(I)
  If Y(I) < Ymin Then Ymin = Y(I)
  If Y(I) > Ymax Then Ymax = Y(I)
Next I
Ymin = (1 - 0.05 * Sgn(Ymin)) * Ymin ‘ Extend Ymin by 5 percent
Ymax = (1 + 0.05 * Sgn(Ymax)) * Ymax ‘ Extend Ymax by 5 percent
ObjectName.Scale (Xmin, Ymax) - (Xmax, Ymin)
ObjectName.Cls
ObjectName.PSet (X(0), Y(0))
For I = 1 To N - 1
  ObjectName.Line - (X(I), Y(I))
Next I
End Sub
```

Simple Bar Charts

- Here, we have a similar situation, **N** points in arrays **X(N - 1)** and **Y(N - 1)**. Can we draw a bar chart using these points? The answer again is yes.

- The procedure to develop a bar chart is similar to that for line charts:

  1. Find the minimum x value (**Xmin**), the maximum x value (**Xmax**), the minimum y value (**Ymin**) and the maximum y value (**Ymax**). Extend the y extremes a bit.

  2. Define a coordinate system using **Scale**:

     Scale (Xmin, Ymax) - (Xmax, Ymin)

  3. For each point, draw a bar using the **Line** procedure:

     Line (x(i), 0) - (x(i), y(i))

  Here, we assume the bars go from 0 to the corresponding y value. You may want to modify this. You could also add color and widen the bars by using the **DrawWidth** property (the example uses blue bars).

- Here is a general procedure that draws a bar chart. Note its similarity to the line chart procedure. Modify it as you wish. The arguments are **ObjectName** the name of the object (form or picture box) you are plotting on, **N** the number of points, **X** the array of x points, and **Y** the array of y points.

```
Sub BarChart(ObjectName As Control, N As Integer, X() As Single, Y() As
Single)
Dim Xmin As Single, Xmax As Single
Dim Ymin As Single, Ymax As Single
Dim I As Integer
Xmin = X(0): Xmax = X(0)
Ymin = Y(0): Ymax = Y(0)
For I = 1 To N - 1
  If X(I) < Xmin Then Xmin = X(I)
  If X(I) > Xmax Then Xmax = X(I)
  If Y(I) < Ymin Then Ymin = Y(I)
  If Y(I) > Ymax Then Ymax = Y(I)
Next I
Ymin = (1 - 0.05 * Sgn(Ymin)) * Ymin ‘ Extend Ymin by 5 percent
Ymax = (1 + 0.05 * Sgn(Ymax)) * Ymax ‘ Extend Ymax by 5 percent
ObjectName.Scale (Xmin, Ymax) - (Xmax, Ymin)
ObjectName.Cls
For I = 0 To N - 1
  ObjectName.Line (X(I), 0) - (X(I), Y(I)), vbBlue
Next I
End Sub
```

Example 7-4

Line Chart and Bar Chart Application

1. Start a new application. Here, we'll use the general line chart and bar chart procedures to plot a simple sine wave.

2. Put a picture box on a form. Set up this simple menu structure using the Menu Editor:

   Plot
   - Line Chart
   - Bar Chart
   - Spiral Chart
   - ———
   - Exit

   Properties for these menu items should be:

Caption	**Name**
&Plot	mnuPlot
&Line Chart	mnuPlotLine
&Bar Chart	mnuPlotBar
&Spiral Chart	mnuPlotSpiral
-	mnuPlotSep
E&xit	mnuPlotExit

   Other properties should be:

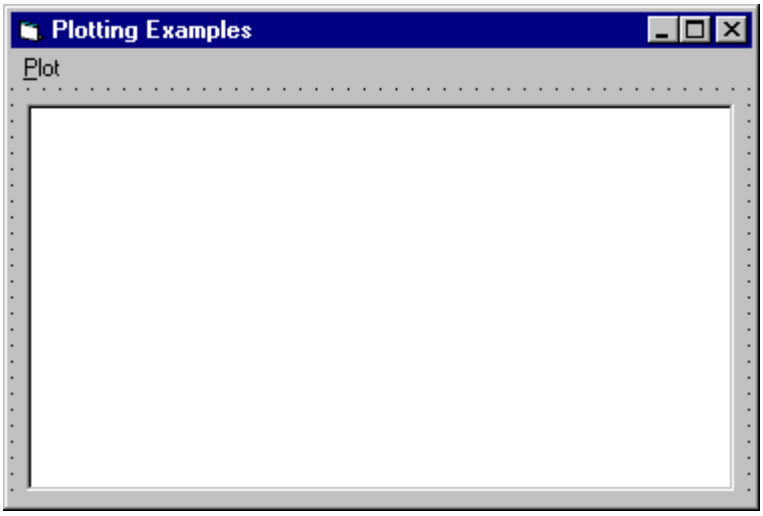
Form1:

BorderStyle	1-Fixed Single
Caption	Plotting Examples
Name	frmPlot

Picture1:

BackColor	White
Name	picPlot

The form should resemble this:



3. Place this code in the **general declarations** area. This makes the x and y arrays and the number of points global.

```
Option Explicit
Dim N As Integer
Dim X(199) As Single
Dim Y(199) As Single
Dim YD(199) As Single
```

4. Attach this code to the **Form_Load** procedure. This loads the arrays with the points to plot.

```
Private Sub form_Load()
Dim I As Integer
Const PI = 3.14159
N = 200
For I = 0 To N - 1
  X(I) = I
  Y(I) = Exp(-0.01 * I) * Sin(PI * I / 10)
  YD(I) = Exp(-0.01 * I) * (PI * Cos(PI * I / 10) / 10 -
0.01 * Sin(PI * I / 10))
Next I
End Sub
```

5. Attach this code to the **mnuPlotLine_Click** event. This draws the line chart.

```
Private Sub mnuPlotLine_Click()
Call LineChart(picPlot, N, X, Y)
End Sub
```

6. Attach this code to the **mnuPlotBar_Click** event. This draws the bar chart.

```
Private Sub mnuPlotBar_Click()
Call BarChart(picPlot, N, X, Y)
End Sub
```

7. Attach this code to the **mnuPlotSpiral_Click** event. This draws a neat little spiral. [Using the line chart, it plots the magnitude of the sine wave (Y array) on the x axis and its derivative (YD array) on the y axis, in case you are interested.]

```
Private Sub mnuPlotSpiral_Click()
Call LineChart(picPlot, N, Y, YD)
End Sub
```

8. And, code for the **mnuPlotExit_Click** event. This stops the application.

```
Private Sub mnuPlotExit_Click()
End
End Sub
```

9. Put the **LineChart** and **BarChart** procedures from these notes in your form as general procedures.

10. Finally, save and run the application. You're ready to tackle any plotting job now.

11. These routines just call out for enhancements. Some things you might try.

    A. Label the plot axes using the **Print** method.

    B. Draw grid lines on the plots. Use dotted or dashed lines at regular intervals.

    C. Put titling information on the axes and the plot.

    D. Modify the line chart routine to allow plotting more than one function. Use colors or different line styles to differentiate the lines. Add a legend defining each plot.

    E. See if you can figure out how to draw a pie chart. Use the **Circle** method to draw the pie segments. Figure out how to fill these segments with different colors and patterns. Label the pie segments.

Exercise 7-1

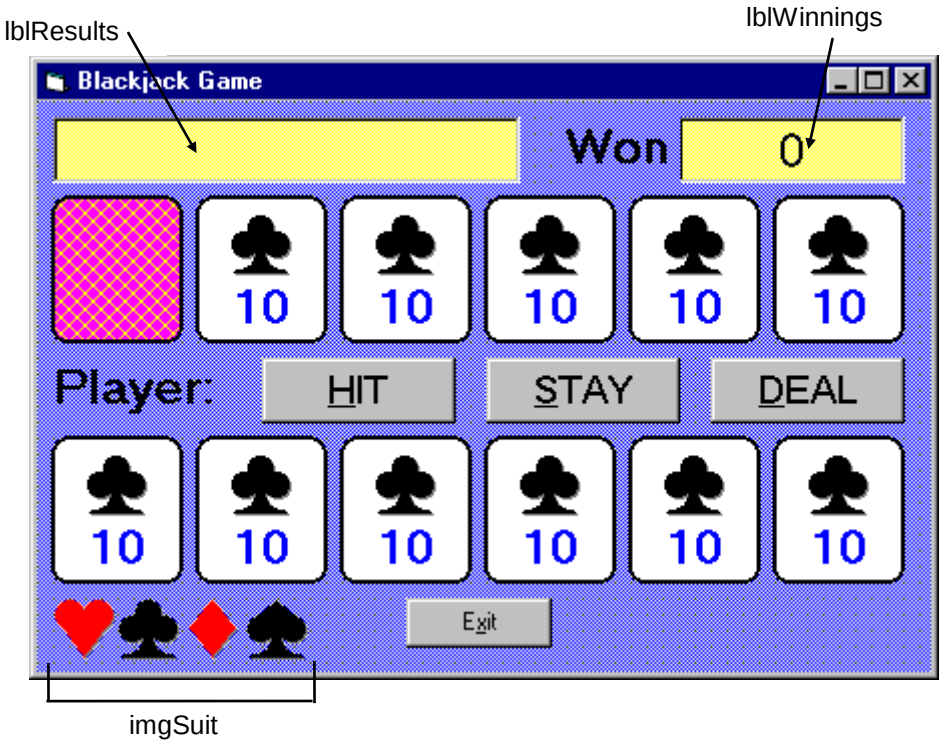
Blackjack

Develop an application that simulates the playing of the card game Blackjack. The idea of Blackjack is to score higher than a Dealer's hand without exceeding twenty-one. Cards count their value, except face cards (jacks, queens, kings) count for ten, and aces count for either one or eleven (your pick). If you beat the Dealer, you get 10 points. If you get Blackjack (21 with just two cards) and beat the Dealer, you get 15 points.

The game starts by giving two cards (from a standard 52 card deck) to the Dealer (one face down) and two cards to the player. The player decides whether to Hit (get another card) or Stay. The player can choose as many extra cards as desired. If the player exceeds 21 before staying, it is a loss (-10 points). If the player does not exceed 21, it becomes the dealer's turn. The Dealer add cards until 16 is exceeded. When this occurs, if the dealer also exceeds 21 or if his total is less than the player's, he loses. If the dealer total is greater than the player total (and under 21), the dealer wins. If the dealer and player have the same total, it is a Push (no points added or subtracted). There are lots of other things you can do in Blackjack, but these simple rules should suffice here. The cards should be reshuffled whenever there are fewer than fifteen (or so) cards remaining in the deck.

My Solution (not a trivial problem):

Form:



There are so many things here, I won't label them all. The button names are obvious. The definition of the cards is not so obvious. Each card is made up of three different objects (each a control array). The card itself is a shape (**shpDealer** for dealer cards, **shpPlayer** for player cards), the number on the card is a label box (**lblDealer** for dealer cards, **lblPlayer** for player cards), and the suit is an image box (**imgDealer** for dealer cards, **imgPlayer** for player cards). There are six elements (one for each card) in each of these control arrays, ranging from element 0 at the left to element 5 at the right. The zero elements of the dealer card controls are obscured by **shpBack** (used to indicate a face down card).

Properties:

Form **frmBlackJack**:
BackColor = &H00FF8080& (Light Blue)
BorderStyle = 1 - Fixed Single
Caption = Blackjack Game

CommandButton **cmdDeal**:
Caption = &DEAL
FontName = MS Sans Serif
FontSize= 13.5

CommandButton **cmdExit:**
Caption = E&xit

CommandButton **cmdStay**:
Caption = &STAY
FontName = MS Sans Serif
FontSize= 13.5

CommandButton **cmdHit**:
Caption = &HIT
FontName = MS Sans Serif
FontSize= 13.5

Image **imgSuit**:
Index = 3
Picture = misc37.ico
Visible = False

Image **imgSuit**:
Index = 2
Picture = misc36.ico
Visible = False

Image **imgSuit**:
Index = 1
Picture = misc35.ico
Visible = False

Image **imgSuit**:
Index = 0
Picture = misc34.ico
Visible = False

Shape **shpBack**:
BackColor = &H00FF00FF& (Magenta)
BackStyle = 1 - Opaque
BorderWidth = 2
FillColor = &H0000FFFF& (Yellow)
FillStyle = 7 - Diagonal Cross
Shape = 4 - Rounded Rectangle

Label **lblPlayer**:
Alignment = 2 - Center
BackColor = &H00FFFFFF&
Caption = 10
FontName = MS Sans Serif
FontBold = True
FontSize = 18
ForeColor = &H00C00000& (Blue)
Index = 5, 4, 3, 2, 1, 0

Image **imgPlayer**:
Picture = misc35.ico
Stretch = True
Index = 5, 4, 3, 2, 1, 0

Shape **shpPlayer**:
BackColor = &H00FFFFFF& (White)
BackStyle = 1 - Opaque
BorderWidth = 2
Shape = 4 - Rounded Rectangle
Index = 5, 4, 3, 2, 1, 0

Label **lblDealer**:
Alignment = 2 - Center
BackColor = &H00FFFFFF&
Caption = 10
FontName = MS Sans Serif
FontBold = True
FontSize = 18
ForeColor = &H00C00000& (Blue)
Index = 5, 4, 3, 2, 1, 0

Image **imgDealer**:
Picture = misc35.ico
Stretch = True
Index = 5, 4, 3, 2, 1, 0

Shape **shpDealer**:
BackColor = &H00FFFFFF& (White)
BackStyle = 1 - Opaque
BorderWidth = 2
Shape = 4 - Rounded Rectangle
Index = 5, 4, 3, 2, 1, 0

Label **Label2**:
BackColor = &H00FF8080& (Light Blue)
Caption = Player:
FontName = MS Sans Serif
FontBold = True
FontSize = 18

Label **lblResults**:
Alignment = 2 - Center
BackColor = &H0080FFFF& (Light Yellow)
BorderStyle = 1 - Fixed Single
FontName = MS Sans Serif
FontSize = 18

Label **Label3**:
BackColor = &H00FF8080& (Light Blue)
Caption = Won
FontName = MS Sans Serif
FontBold = True
FontSize = 18

Label **lblWinnings**:
Alignment = 2 - Center
BackColor = &H0080FFFF& (Light Yellow)
BorderStyle = 1 - Fixed Single
Caption = 0
FontName = MS Sans Serif
FontSize = 18

Code:

General Declarations:

```
Option Explicit
Dim CardName(52) As String
Dim CardSuit(52) As Integer
Dim CardValue(52) As Integer
Dim Winnings As Integer, CurrentCard As Integer
Dim Aces_Dealer As Integer, Aces_Player As Integer
Dim Score_Dealer As Integer, Score_Player As Integer
Dim NumCards_Dealer As Integer, NumCards_Player As Integer
```

Add_Dealer General Procedure:

```
Sub Add_Dealer()
Dim I As Integer
'Adds a card at index I to dealer hand
NumCards_Dealer = NumCards_Dealer + 1
I = NumCards_Dealer - 1
lblDealer(I).Caption = CardName(CurrentCard)
imgDealer(I).Picture = imgSuit(CardSuit(CurrentCard)).Picture
Score_Dealer = Score_Dealer + CardValue(CurrentCard)
If CardValue(CurrentCard) = 1 Then Aces_Dealer = Aces_Dealer
+ 1
CurrentCard = CurrentCard + 1
lblDealer(I).Visible = True
imgDealer(I).Visible = True
shpDealer(I).Visible = True
End Sub
```

Add_Player General Procedure:

```
Sub Add_Player()
Dim I As Integer
'Adds a card at index I to player hand
NumCards_Player = NumCards_Player + 1
I = NumCards_Player - 1
lblPlayer(I).Caption = CardName(CurrentCard)
imgPlayer(I).Picture = imgSuit(CardSuit(CurrentCard)).Picture
Score_Player = Score_Player + CardValue(CurrentCard)
If CardValue(CurrentCard) = 1 Then Aces_Player = Aces_Player
+ 1
lblPlayer(I).Visible = True
imgPlayer(I).Visible = True
shpPlayer(I).Visible = True
CurrentCard = CurrentCard + 1
End Sub
```

End_Hand General Procedure:

```
Sub End_Hand(Msg As String, Change As Integer)
shpBack.Visible = False
lblResults.Caption = Msg
'Hand has ended - update winnings
Winnings = Winnings + Change
lblwinnings.Caption = Str(Winnings)
cmdHit.Enabled = False
cmdStay.Enabled = False
cmdDeal.Enabled = True
End Sub
```

New_Hand General Procedure:

```
Sub New_Hand()
'Deal a new hand
Dim I As Integer
'Clear table of cards
For I = 0 To 5
  lblDealer(I).Visible = False
  imgDealer(I).Visible = False
  shpDealer(I).Visible = False
  lblPlayer(I).Visible = False
  imgPlayer(I).Visible = False
  shpPlayer(I).Visible = False
Next I
lblResults.Caption = ""
cmdHit.Enabled = True
cmdStay.Enabled = True
cmdDeal.Enabled = False
If CurrentCard > 35 Then Call Shuffle_Cards
'Get two dealer cards
Score_Dealer = 0: Aces_Dealer = 0: NumCards_Dealer = 0
shpBack.Visible = True
Call Add_Dealer
Call Add_Dealer
'Get two player cards
Score_Player = 0: Aces_Player = 0: NumCards_Player = 0
Call Add_Player
Call Add_Player
'Check for blackjacks
If Score_Dealer = 11 And Aces_Dealer = 1 Then Score_Dealer =
21
If Score_Player = 11 And Aces_Player = 1 Then Score_Player =
21
If Score_Dealer = 21 And Score_Player = 21 Then
  Call End_Hand("Two Blackjacks!", 0)
  Exit Sub
```

```
ElseIf Score_Dealer = 21 Then
  Call End_Hand("Dealer Blackjack!", -10)
  Exit Sub
ElseIf Score_Player = 21 Then
  Call End_Hand("Player Blackjack!", 15)
  Exit Sub
End If
End Sub
```

N_Integers General Procedure:

```
Private Sub N_Integers(N As Integer, Narray() As Integer)
'Randomly sorts N integers and puts results in Narray
Dim I As Integer, J As Integer, T As Integer
'Order all elements initially
For I = 1 To N: Narray(I) = I: Next I
'J is number of integers remaining
For J = N to 2 Step -1
  I = Int(Rnd * J) + 1
  T = Narray(J)
  Narray(J) = Narray(I)
  Narray(I) = T
Next J
End Sub
```

Shuffle_Cards General Procedure:

```
Sub Shuffle_Cards()
'Shuffle a deck of cards.  That is, randomly sort
'the integers from 1 to 52 and convert to cards.
'Cards 1-13 are the ace through king of hearts
'Cards 14-26 are the ace through king of clubs
'Cards 27-39 are the ace through king of diamonds
'Cards 40-52 are the ace through king of spades
'When done:
'The array element CardName(i) has the name of the ith card
'The array element CardSuit(i) is the index to the ith card
suite
'The array element CardValue(i) has the point value of the
ith card
Dim CardUsed(52) As Integer
Dim J As Integer
Call N_Integers(52, CardUsed())
For J = 1 to 52
  Select Case (CardUsed(J) - 1) Mod 13 + 1
    Case 1
      CardName(J) = "A"
      CardValue(J) = 1
```

```
    Case 2
      CardName(J) = "2"
      CardValue(J) = 2
    Case 3
      CardName(J) = "3"
      CardValue(J) = 3
    Case 4
      CardName(J) = "4"
      CardValue(J) = 4
    Case 5
      CardName(J) = "5"
      CardValue(J) = 5
    Case 6
      CardName(J) = "6"
      CardValue(J) = 6
    Case 7
      CardName(J) = "7"
      CardValue(J) = 7
    Case 8
      CardName(J) = "8"
      CardValue(J) = 8
    Case 9
      CardName(J) = "9"
      CardValue(J) = 9
    Case 10
      CardName(J) = "10"
      CardValue(J) = 10
    Case 11
      CardName(J) = "J"
      CardValue(J) = 10
    Case 12
      CardName(J) = "Q"
      CardValue(J) = 10
    Case 13
      CardName(J) = "K"
      CardValue(J) = 10
  End Select
  CardSuit(J) = Int((CardUsed(J) - 1) / 13)
Next J
CurrentCard = 1
End Sub
```

cmdDeal Click Event:

```
Private Sub cmdDeal_Click()
Call New_Hand
End Sub
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
'Show final winnings and quit
If Winnings > 0 Then
  MsgBox "You won" + Str(Winnings) + " points!", vbOKOnly,
"Game Over"
ElseIf Winnings = 0 Then
  MsgBox "You broke even.", vbOKOnly, "Game Over"
Else
  MsgBox "You lost" + Str(Abs(Winnings)) + " points!",
vbOKOnly, "Game Over"
End If
End
End Sub
```

cmdHit Click Event:

```
Private Sub cmdHit_Click()
'Add a card if player requests
Call Add_Player
If Score_Player > 21 Then
  Call End_Hand("Player Busts!", -10)
  Exit Sub
End If
If NumCards_Player = 6 Then
  cmdHit.Enabled = False
  Call cmdStay_Click
  Exit Sub
End If
End Sub
```

cmdStay Click Event:

```
Private Sub cmdStay_Click()
Dim ScoreTemp As Integer, AcesTemp As Integer
'Check for aces in player hand and adjust score
'to highest possible
cmdHit.Enabled = False
cmdStay.Enabled = False
If Aces_Player <> 0 Then
  Do
    Score_Player = Score_Player + 10
    Aces_Player = Aces_Player - 1
  Loop Until Aces_Player = 0 Or Score_Player > 21
  If Score_Player > 21 Then Score_Player = Score_Player - 10
End If
'Uncover dealer face down card and play dealer hand
```

```
shpBack.Visible = False
NextTurn:
ScoreTemp = Score_Dealer: AcesTemp = Aces_Dealer
'Check for aces and adjust score
If AcesTemp <> 0 Then
  Do
    ScoreTemp = ScoreTemp + 10
    AcesTemp = AcesTemp - 1
  Loop Until AcesTemp = 0 Or ScoreTemp > 21
  If ScoreTemp > 21 Then ScoreTemp = ScoreTemp - 10
End If
'Check if dealer won
If ScoreTemp > 16 Then
  If ScoreTemp > Score_Player Then
    Call End_Hand("Dealer Wins!", -10)
    Exit Sub
  ElseIf ScoreTemp = Score_Player Then
    Call End_Hand("It's a Push!", 0)
    Exit Sub
  Else
    Call End_Hand("Player Wins!", 10)
    Exit Sub
  End If
End If
'If six cards shown and dealer hasn't won, player wins
If NumCards_Dealer = 6 Then
  Call End_Hand("Player Wins!", 10)
  Exit Sub
End If
'See if hit is needed
If ScoreTemp < 17 Then Call Add_Dealer
If Score_Dealer > 21 Then
  Call End_Hand("Dealer Busts!", 10)
  Exit Sub
End If
GoTo NextTurn
End Sub
```

Form_Load Event:

```
Private Sub Form_Load()
'Seed random number generator, shuffle cards, deal new hand
Randomize Timer
Call Shuffle_Cards
Call New_Hand
End Sub
```

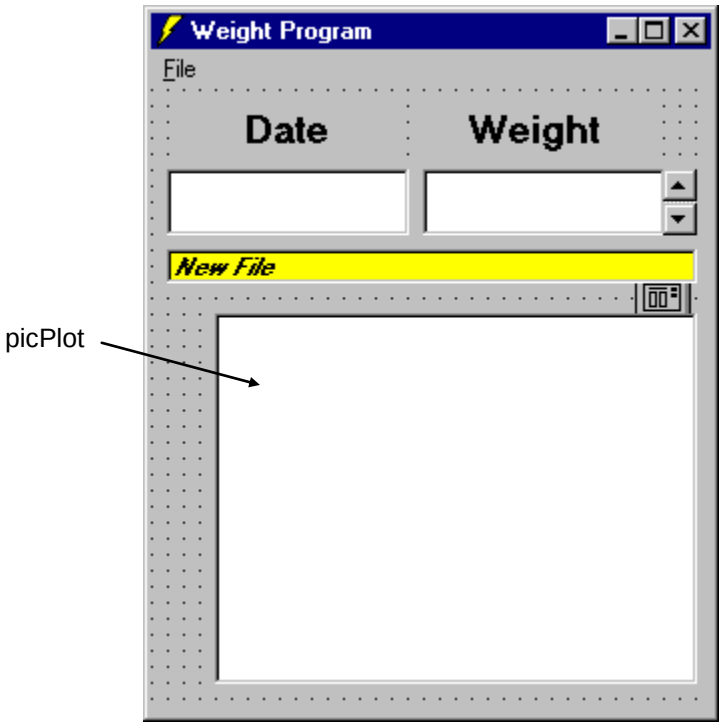
Exercise 7-2

Information Tracking Plotting

Add plotting capabilities to the information tracker you developed in Class 6. Plot whatever information you stored versus the date. Use a line or bar chart.

My Solution:

Form (like form in Homework 6, with a picture box and Plot menu item added):



New Properties:

Form **frmWeight**:
FontName = MS Sans Serif
FontSize = 10

PictureBox **picPlot**:
BackColor = &H00FFFFFF& (White)
DrawWidth = 2

Menu **mnuFilePlot**:
Caption = &Plot

New Code:

mnuFilePlot Click Event:

```
Private Sub mnuFilePlot_Click()
Dim X(100) As Integer, Y(100) As Integer
Dim I As Integer
Dim Xmin As Integer, Xmax As Integer
Dim Ymin As Integer, Ymax As Integer
Dim Legend As String
Xmin = 0: Xmax = 0
Ymin = Val(Weights(1)): Ymax = Ymin
For I = 1 To NumWts
  X(I) = DateDiff("d", Dates(1), Dates(I))
  Y(I) = Val(Weights(I))
  If X(I) < Xmin Then Xmin = X(I)
  If X(I) > Xmax Then Xmax = X(I)
  If Y(I) < Ymin Then Ymin = Y(I)
  If Y(I) > Ymax Then Ymax = Y(I)
Next I
Xmin = Xmin - 1: Xmax = Xmax + 1
Ymin = (1 - 0.05 * Sgn(Ymin)) * Ymin
Ymax = (1 + 0.05 * Sgn(Ymax)) * Ymax
picplot.Scale (Xmin, Ymax)-(Xmax, Ymin)
Cls
picplot.Cls
For I = 1 To NumWts
  picplot.Line (X(I), Ymin)-(X(I), Y(I)), QBColor(1)
Next I
Legend = Str(Ymax)
CurrentX = picplot.Left - TextWidth(Legend)
CurrentY = picplot.Top - 0.5 * TextHeight(Legend)
Print Legend
Legend = Str(Ymin)
CurrentX = picplot.Left - TextWidth(Legend)
CurrentY = picplot.Top + picplot.Height - 0.5 *
TextHeight(Legend)
Print Legend
End Sub
```

This page intentionally not left blank.