Learn Visual Basic 6.0

© KIDware (206) 721-2556

This copy of **Learn Visual Basic 6.0** is licensed to a single user. Copies of the course are not to be distributed or provided to any other user. Multiple copy licenses are available for businesses and educational institutions. Please contact KIDware for license information.

Course Description:

Learn Visual Basic 6.0 is a 10 week, self-paced overview of the Visual Basic programming language and environment. Upon completion of the course, you will:

1. Understand the benefits of using Microsoft Visual Basic 6.0 as an application development tool.
2. Understand the Visual Basic event-driven programming concepts, terminology, and available tools.
3. Learn the fundamentals of designing, implementing, and distributing a wide variety of Visual Basic applications.

Learn Visual Basic 6.0 is presented using a combination of course notes (written in Microsoft Word format) and over 60 Visual Basic examples and applications.

Course Prerequisites:

To grasp the concepts presented in **Learn Visual Basic 6.0**, you should possess a working knowledge of Windows 95 and have had some exposure to programming concepts. If you have never programmed a computer before, you'll have to put in a little more effort - perhaps, find a book in your local library on programming using QBasic or some other dialect of the Basic computer language.

You will also need the ability to view and print documents saved in Microsoft Word for Windows 95 format. This can be accomplished in one of two ways. The first, and easiest, is that you already have Microsoft Word for Windows 95 on your computer. The second way, and a bit more difficult, is that you can download Word Viewer for Windows 95. This is a free Microsoft product that allows viewing Word documents - it is available for download at all the major shareware internet sites (ZDNet, Download.Com, SoftSeek).

Finally, and most obvious, you need to have Microsoft Visual Basic 6.0, preferably the Professional Edition. **Learn Visual Basic 6.0** does not cover the rudiments of navigating in Visual Basic 6.0. You should be familiar with the simple tasks of using the menus, the toolbar, resizing windows, and moving windows around. Visual Basic 6.0 provides an excellent tutorial with instruction on such tasks.

Installing Learn Visual Basic 6.0:

The course notes and code for **Learn Visual Basic 6.0** are included in two ZIP files (**LVB61.ZIP** and **LVB62.ZIP**) on separate disks. Use your favorite 'unzipping' application to write all files to your computer. After unzipping, the course is included in the folder entitled **LearnVB6**. This folder contains two other folders: **VB Notes** and **VB Code**.

The **VB Notes** folder includes all the notes needed for the class. Each file in this folder has a DOC extension and is in Word for Windows 95 format. The files are:

Start Here.doc	This file in Word format
Contents.doc	Course Table of Contents
Class 1.doc	Class 1. Introduction to the Visual Basic Language and Environment
Class 2.doc	Class 2. The Visual Basic Language
Class 3.doc	Class 3. Exploring the Visual Basic Toolbox
Class 4.doc	Class 4. More Exploration of the Visual Basic Toolbox
Class 5.doc	Class 5. Creating a Stand-Alone Visual Basic Application
Class 6.doc	Class 6. Error-Handling, Debugging and File Input/Output
Class 7.doc	Class 7. Graphics Techniques with Visual Basic
Class 8.doc	Class 8. Database Access and Management
Class 9.doc	Class 9. Dynamic Link Libraries and the Windows API
Class 10.doc	Class 10. Other Visual Basic Topics
Appendix I.doc	Appendix I. Visual Basic Symbolic Constants
Appendix II.doc	Appendix II. Common Dialog Box Constants

The **VB Code** folder includes all the Visual Basic applications developed during the course. The applications are further divided into **Class** folders.

How To Take the Course:

Learn Visual Basic 6.0 is a self-paced course. The suggested approach is to do one class a week for ten weeks. Each week's class should require about 4 to 8 hours of your time to grasp the concepts completely. Prior to doing a particular week's work, open the class notes file for that week and print it out. Then, work through the notes at your own pace. Try to do each example as they are encountered in the notes. If you need any help, all solved examples are included in the **VB Code** folder. After completing each week's notes, a homework exercise is given, covering many of the topics taught that week. Like the examples, try to work through the homework exercise, or some variation thereof, on your own. Refer to the completed project in the **VB Code** folder, if necessary.

What If You Have Questions?

It is recognized there may be times when you need clarification on some point about the notes, examples, or Visual Basic. We will gladly help. The preferred method of relaying your questions to us is via E-Mail. The E-Mail address is:

KIDware@jetcity.com

Please include a clearly defined subject for all questions to get past our anti-spamming filters. All questions should be sent to the attention of **Lou**.

Who Produces Learn Visual Basic 6.0?

This course has been developed by Lou Tylee, a partner in KIDware, a producer of quality children's educational programs for over 15 years. The course notes have evolved based on Lou's experience in writing children's software and in teaching a similar course at the university level for over four years. KIDware may be contacted via:

KIDware
15600 NE 8th, Suite B1-314
Bellevue, WA 98008
(206) 721-2556
FAX (425) 746-4655
E-Mail: KIDware@jetcity.com
Web Site: http://www.jetcity.com/~kidware

Course Notes for:

Learn Visual Basic 6.0

© Lou Tylee, 1998

KIDware
15600 NE 8th, Suite B1-314
Bellevue, WA 98008
(206) 721-2556
FAX (425) 746-4655

Notice

These notes were developed for the course, "Learn Visual Basic 6.0" They are not intended to be a complete reference to Visual Basic. Consult the *Microsoft Visual Basic Programmer's Guide* and *Microsoft Visual Basic Language Reference Manual* for detailed reference information.

The notes refer to several software and hardware products by their trade names. These references are for informational purposes only and all trademarks are the property of their respective companies.

Lou Tylee
Course Instructor

Learn Visual Basic 6.0

Contents

1.   Introduction to the Visual Basic Language and Environment

2. The Visual Basic Language

3. Exploring the Visual Basic Toolbox

4. More Exploration of the Visual Basic Toolbox

5. Creating a Stand-Alone Visual Basic Application

6.   Error-Handling, Debugging and File Input/Output

7.   Graphics Techniques with Visual Basic

7.   Graphics Techniques with Visual Basic (continued)

8.   Database Access and Management

9.   Dynamic Link Libraries and the Windows API

9. Dynamic Link Libraries and the Windows API (continued)

10. Other Visual Basic Topics

Learn Visual Basic 6.0

1. Introduction to the Visual Basic Language and Environment

Preview

- In this first class, we will do a quick overview of how to build an application in Visual Basic. You'll learn a new vocabulary, a new approach to programming, and ways to move around in the Visual Basic environment. You will leave having written your first Visual Basic program.

Course Objectives

⇒ Understand the benefits of using Microsoft Visual Basic 6.0 for Windows as an application tool

⇒ Understand the Visual Basic event-driven programming concepts, terminology, and available tools

⇒ Learn the fundamentals of designing, implementing, and distributing a Visual Basic application

⇒ Learn to use the Visual Basic toolbox

⇒ Learn to modify object properties

⇒ Learn object methods

⇒ Use the menu design window

⇒ Understand proper debugging and error-handling procedures

⇒ Gain a basic understanding of database access and management using databound controls

⇒ Obtain an introduction to ActiveX controls and the Windows Application Programming Interface (API)

What is Visual Basic?

• **Visual Basic** is a tool that allows you to develop Windows (Graphic User Interface - **GUI**) applications. The applications have a familiar appearance to the user.

• Visual Basic is **event-driven**, meaning code remains idle until called upon to respond to some event (button pressing, menu selection, ...). Visual Basic is governed by an event processor. Nothing happens until an event is detected. Once an event is detected, the code corresponding to that event (event procedure) is executed. Program control is then returned to the event processor.
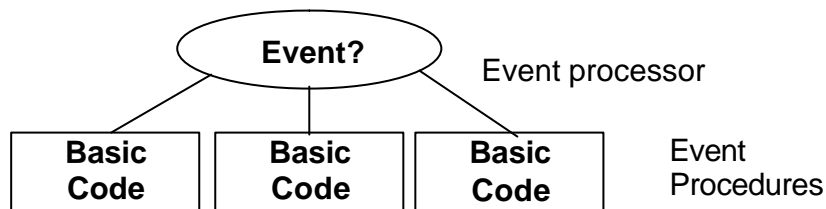
• Some Features of Visual Basic

⇒ Full set of objects - you 'draw' the application
⇒ Lots of icons and pictures for your use
⇒ Response to mouse and keyboard actions
⇒ Clipboard and printer access
⇒ Full array of mathematical, string handling, and graphics functions
⇒ Can handle fixed and dynamic variable and control arrays
⇒ Sequential and random access file support
⇒ Useful debugger and error-handling facilities
⇒ Powerful database access tools
⇒ ActiveX support
⇒ Package & Deployment Wizard makes distributing your applications simple

Visual Basic 6.0 versus Other Versions of Visual Basic

- The original Visual Basic for DOS and Visual Basic For Windows were introduced in 1991.

- Visual Basic 3.0 (a vast improvement over previous versions) was released in 1993.

- Visual Basic 4.0 released in late 1995 (added 32 bit application support).
-
- Visual Basic 5.0 released in late 1996. New environment, supported creation of ActiveX controls, deleted 16 bit application support.

- And, now Visual Basic 6.0 - some identified new features of Visual Basic 6.0:

    ⇒ Faster compiler
    ⇒ New ActiveX data control object
    ⇒ Allows database integration with wide variety of applications
    ⇒ New data report designer
    ⇒ New Package & Deployment Wizard
    ⇒ Additional internet capabilites

16 Bits versus 32 Bits

- Applications built using the Visual Basic 3.0 and the 16 bit version of Visual Basic 4.0 will run under Windows 3.1, Windows for Workgroups, Windows NT, or Windows 95

- Applications built using the 32 bit version of Visual Basic 4.0, Visual Basic 5.0 and Visual Basic 6.0 will only run with Windows 95 or Windows NT (Version 3.5.1 or higher).

- In this class, we will use Visual Basic 6.0 under Windows 95, recognizing such applications will not operate in 16 bit environments.

Structure of a Visual Basic Application

Project (.VBP, .MAK)

Form 1 (.FRM)	Form 2 (.FRM)	Form 3 (.FRM)	Module 1 (.BAS)
Control Control Control	Control Control Control	Control Control Control	
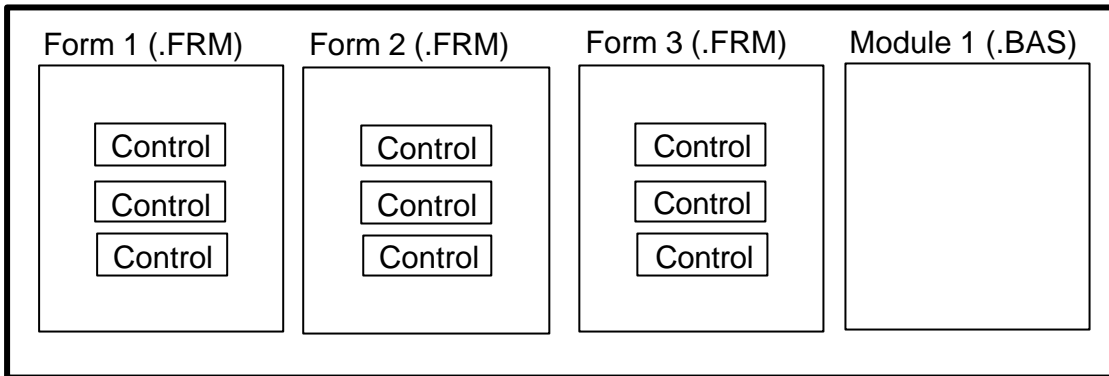
Application (Project) is made up of:

⇒ **Forms** - Windows that you create for user interface
⇒ **Controls** - Graphical features drawn on forms to allow user interaction (text boxes, labels, scroll bars, command buttons, etc.) (Forms and Controls are **objects**.)
⇒ **Properties** - Every characteristic of a form or control is specified by a property. Example properties include names, captions, size, color, position, and contents. Visual Basic applies default properties. You can change properties at design time or run time.
⇒ **Methods** - Built-in procedure that can be invoked to impart some action to a particular object.
⇒ **Event Procedures** - Code related to some object. This is the code that is executed when a certain event occurs.
⇒ **General Procedures** - Code not related to objects. This code must be invoked by the application.
⇒ **Modules** - Collection of general procedures, variable declarations, and constant definitions used by application.

Steps in Developing Application

• There are three primary steps involved in building a Visual Basic application:

1. **Draw** the user **interface**
2. **Assign properties** to controls
3. **Attach code** to controls

We'll look at each step.

Drawing the User Interface and Setting Properties

- Visual Basic operates in three modes.

  ⇒ **Design** mode - used to build application
  ⇒ **Run** mode - used to run the application
  ⇒ **Break** mode - application halted and debugger is available

  We focus here on the **design** mode.

- Six windows appear when you start Visual Basic.

  ⇒ The **Main Window** consists of the title bar, menu bar, and toolbar. The title bar indicates the project name, the current Visual Basic operating mode, and the current form. The menu bar has drop-down menus from which you control the operation of the Visual Basic environment. The toolbar has buttons that provide shortcuts to some of the menu options. The main window also shows the location of the current form relative to the upper left corner of the screen (measured in twips) and the width and length of the current form.
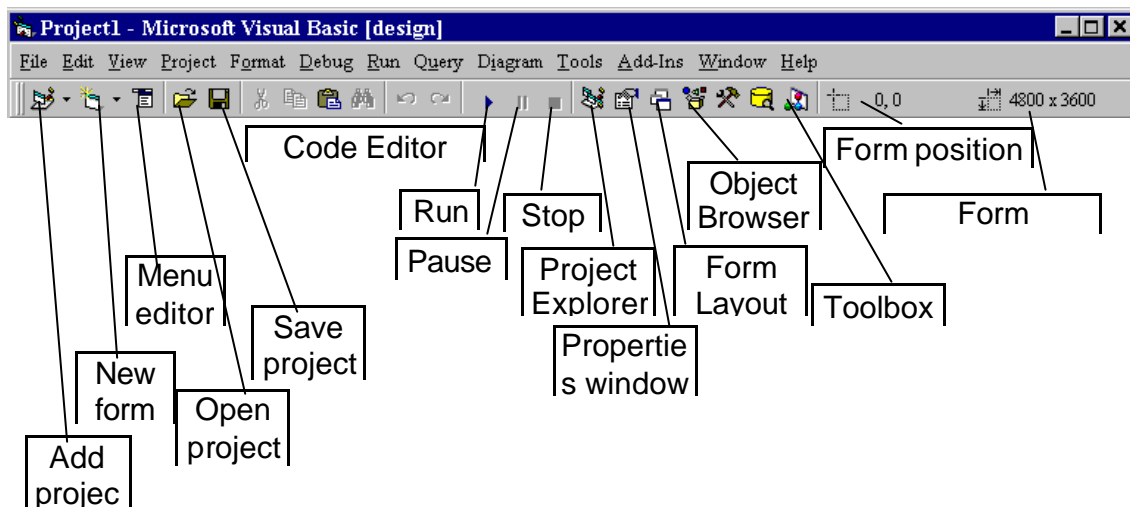
⇒ The **Form Window** is central to developing Visual Basic applications. It is where you draw your application.
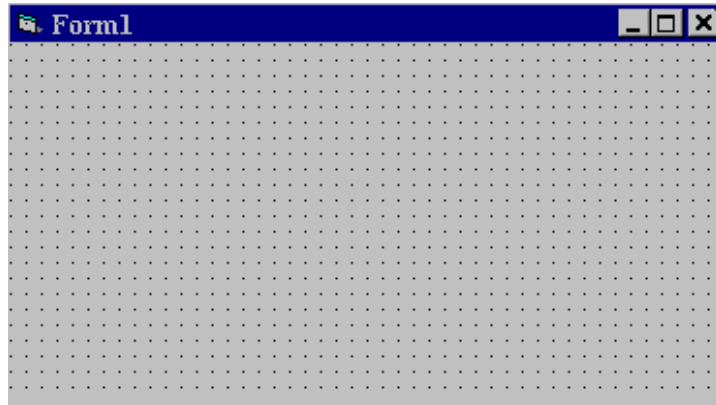
⇒ The **Toolbox** is the selection menu for controls used in your application.

Pointer	Picture Box
Label	Text Box
Frame	Command Button
Check Box	Option Button
Combo Box	List Box
Horizontal Scroll	Vertical Scroll Bar
Timer	Drive List Box
Directory List Box	File List Box
Shapes	Lines
Image Box	Data Tool
Object Linking	
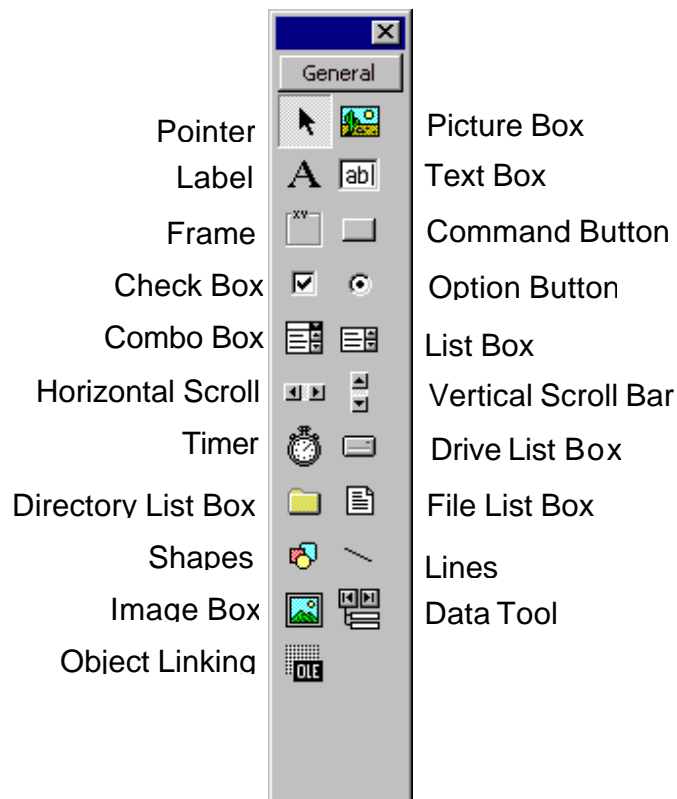
⇒ The **Properties Window** is used to establish initial property values for objects. The drop-down box at the top of the window lists all objects in the current form. Two views are available: Alphabetic and Categorized. Under this box are the available properties for the currently selected object.
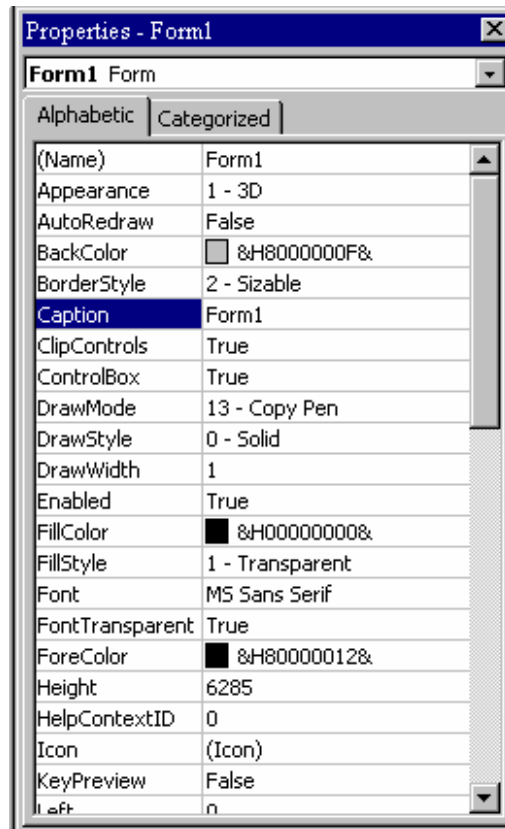
⇒ The **Form Layout Window** shows where (upon program execution) your form will be displayed relative to your monitor's screen:
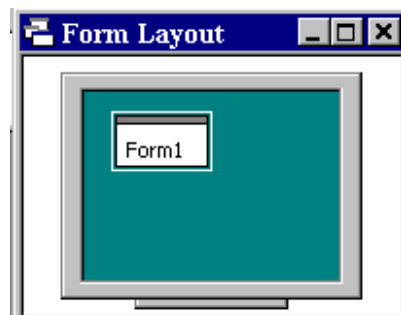
⇒ The **Project Window** displays a list of all forms and modules making up your application. You can also obtain a view of the **Form** or **Code** windows (window containing the actual Basic coding) from the Project window.
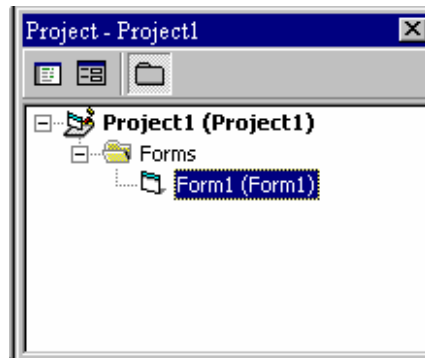
- As mentioned, the user interface is 'drawn' in the form window. There are two ways to place controls on a form:

    1. Double-click the tool in the toolbox and it is created with a default size on the form. You can then move it or resize it.

    2. Click the tool in the toolbox, then move the mouse pointer to the form window. The cursor changes to a crosshair. Place the crosshair at the upper left corner of where you want the control to be, press the left mouse button and hold it down while dragging the cursor toward the lower right corner. When you release the mouse button, the control is drawn.

- To **move** a control you have drawn, click the object in the form window and drag it to the new location. Release the mouse button.

- To **resize** a control, click the object so that it is select and sizing handles appear. Use these handles to resize the object.
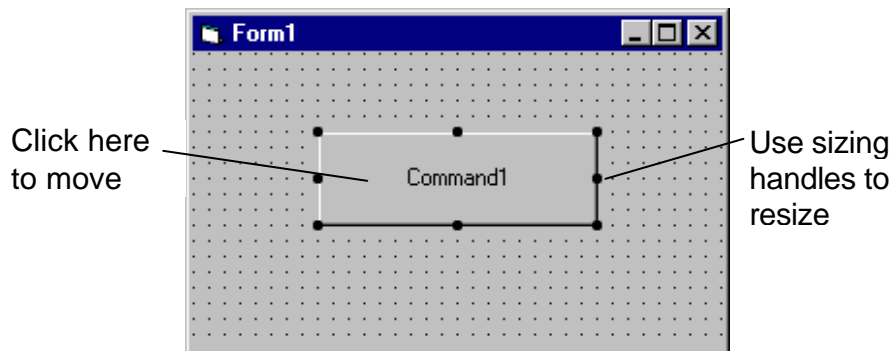
<u>Example 1-1</u>

Stopwatch Application - Drawing Controls

1. Start a new project. The idea of this project is to start a timer, then stop the timer and compute the elapsed time (in seconds).

2. Place three command buttons and six labels on the form. Move and size the controls and form so it looks something like this:
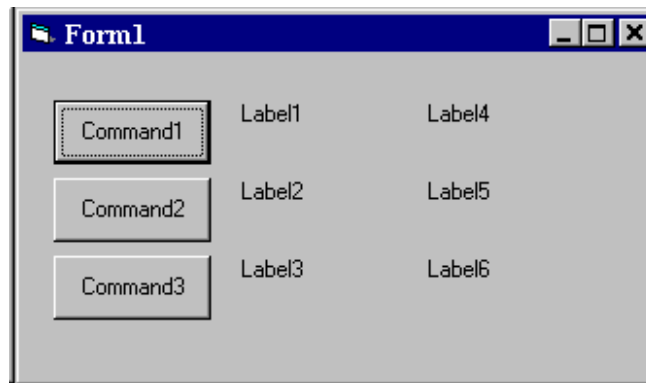
Setting Properties of Objects at Design Time

- Each form and control has **properties** assigned to it by default when you start up a new project. There are two ways to display the properties of an object. The first way is to click on the object (form or control) in the form window. Then, click on the Properties Window or the Properties Window button in the tool bar. The second way is to first click on the Properties Window. Then, select the object from the **Object** box in the Properties Window. Shown is the Properties Window for the stopwatch application:

Properties - Form1	
Form1 Form	
Alphabetic	Categorized

(Name)	Form1
Appearance	1 - 3D
AutoRedraw	False
BackColor	&H8000000F&
BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen
DrawStyle	0 - Solid
DrawWidth	1
Enabled	True
FillColor	&H00000000&
FillStyle	1 - Transparent
Font	MS Sans Serif
FontTransparent	True
ForeColor	&H80000012&
Height	6285
HelpContextID	0
Icon	(Icon)
KeyPreview	False
Left	0

The drop-down box at the top of the Properties Window is the **Object** box. It displays the name of each object in the application as well as its type. This display shows the **Form** object. The **Properties** list is directly below this box. In this list, you can scroll through the list of properties for the selected object. You may select a property by clicking on it. Properties can be changed by typing a new value or choosing from a list of predefined settings (available as a drop down list). Properties can be viewed in two ways: **Alphabetic** and **Categorized**.

A very important property for each object is its **name**. The name is used by Visual Basic to refer to a particular object in code.

- A convention has been established for naming Visual Basic objects. This convention is to use a three letter prefix (depending on the object) followed by a name you assign. A few of the prefixes are (we'll see more as we progress in the class):

Object	Prefix	Example
Form	frm	frmWatch
Command Button	cmd, btn	cmdExit, btnStart
Label	lbl	lblStart, lblEnd
Text Box	txt	txtTime, txtName
Menu	mnu	mnuExit, mnuSave
Check box	chk	chkChoice

- Object names can be up to 40 characters long, must start with a letter, must contain only letters, numbers, and the underscore (_) character. Names are used in setting properties at run time and also in establishing procedure names for object events.

Setting Properties at Run Time

- You can also set or modify properties while your application is running. To do this, you must write some code. The code format is:

  ObjectName.Property = NewValue

  Such a format is referred to as dot notation. For example, to change the **BackColor** property of a form name **frmStart**, we'd type:

  frmStart.BackColor = BLUE

How Names are Used in Object Events

- The names you assign to objects are used by Visual Basic to set up a framework of event-driven procedures for you to add code to. The format for each of these subroutines (all object procedures in Visual Basic are subroutines) is:

  Sub ObjectName_Event (Optional Arguments)
       .
       .
  End Sub

- Visual Basic provides the **Sub** line with its arguments (if any) and the **End Sub** statement. You provide any needed code.

<u>Example 1-2</u>

Stopwatch Application - Setting Properties

1. Set properties of the form, three buttons, and six labels:

    Form1:
BorderStyle	1-Fixed Single
Caption	Stopwatch Application
Name	frmStopWatch

    Command1:
Caption	&Start Timing
Name	cmdStart

    Command2:
Caption	&End Timing
Name	cmdEnd

    Command3:
Caption	E&xit
Name	cmdExit

    Label1:
Caption	Start Time

    Label2:
Caption	End Time

    Label3:
Caption	Elapsed Time

    Label4:
BorderStyle	1-Fixed Single
Caption	[Blank]
Name	lblStart

    Label5:
BorderStyle	1-Fixed Single
Caption	[Blank]
Name	lblEnd

Label6:
BorderStyle   1-Fixed Single
Caption   [Blank]
Name   lblElapsed

In the **Caption** properties of the three command buttons, notice the ampersand (**&**). The ampersand precedes a button's **access key**. That is, in addition to clicking on a button to invoke its event, you can also press its access key (no need for a mouse). The access key is pressed in conjunction with the **Alt** key. Hence, to invoke 'Begin Timing', you can either click the button or press Alt+B. Note in the button captions on the form, the access keys appear with an underscore (_).

2. Your form should now look something like this:
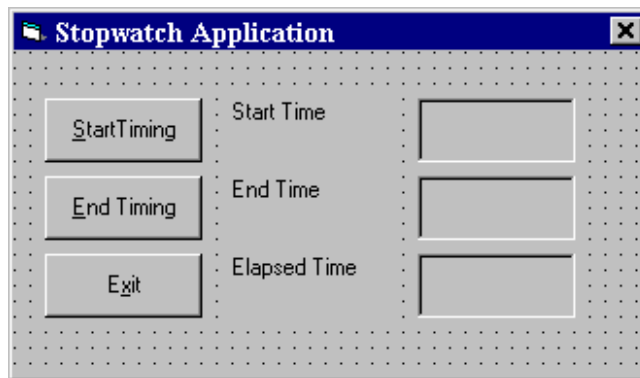
Variables

- We're now ready to attach code to our application. As objects are added to the form, Visual Basic automatically builds a framework of all event procedures. We simply add code to the event procedures we want our application to respond to. But before we do this, we need to discuss **variables**.

- Variables are used by Visual Basic to hold information needed by your application. Rules used in naming variables:

  ⇒ No more than 40 characters
  ⇒ They may include letters, numbers, and underscore (_)
  ⇒ The first character must be a letter
  ⇒ You cannot use a reserved word (word needed by Visual Basic)

Visual Basic Data Types

Data Type	Suffix
Boolean	None
Integer	%
Long (Integer)	&
Single (Floating)	!
Double (Floating)	#
Currency	@
Date	None
Object	None
String	$
Variant	None

Variable Declaration

- There are three ways for a variable to be typed (declared):

  1. Default
  2. Implicit
  3. Explicit

- If variables are not implicitly or explicitly typed, they are assigned the **variant** type by **default**. The variant data type is a special type used by Visual Basic that can contain numeric, string, or date data.

- To **implicitly** type a variable, use the corresponding suffix shown above in the data type table. For example,

    TextValue$ = "This is a string"

  creates a string variable, while

    Amount% = 300

  creates an integer variable.

- There are many advantages to **explicitly** typing variables. Primarily, we insure all computations are properly done, mistyped variable names are easily spotted, and Visual Basic will take care of insuring consistency in upper and lower case letters used in variable names. Because of these advantages, and because it is good programming practice, we will explicitly type all variables.

- To **explicitly** type a variable, you must first determine its **scope**. There are four levels of scope:

    ⇒ Procedure level
    ⇒ Procedure level, static
    ⇒ Form and module level
    ⇒ Global level

- Within a procedure, variables are declared using the **Dim** statement:

    Dim MyInt as Integer
    Dim MyDouble as Double
    Dim MyString, YourString as String

  Procedure level variables declared in this manner do not retain their value once a procedure terminates.

- To make a procedure level variable retain its value upon exiting the procedure, replace the Dim keyword with **Static**:

    Static MyInt as Integer
    Static MyDouble as Double

- Form (module) level variables retain their value and are available to all procedures within that form (module). Form (module) level variables are declared in the **declarations** part of the **general** object in the form's (module's) code window. The **Dim** keyword is used:

    Dim MyInt as Integer
    Dim MyDate as Date

- Global level variables retain their value and are available to all procedures within an application. Module level variables are declared in the **declarations** part of the **general** object of a module's code window. (It is advisable to keep all global variables in one module.) Use the **Global** keyword:

    Global MyInt as Integer
    Global MyDate as Date

- What happens if you declare a variable with the same name in two or more places? More local variables **shadow** (are accessed in preference to) less local variables. For example, if a variable MyInt is defined as Global in a module and declared local in a routine MyRoutine, while in MyRoutine, the local value of MyInt is accessed. Outside MyRoutine, the global value of MyInt is accessed.

- Example of Variable Scope:

Module1

Global X as Integer

Form1   **Form2**

Dim Y as Integer   Dim Z as Single

Sub Routine1()   Sub Routine3()
  Dim A as Double   Dim C as String
  .   .
  .   .
End Sub   End Sub

Sub Routine2()
  Static B as Double
  .
  .
End Sub
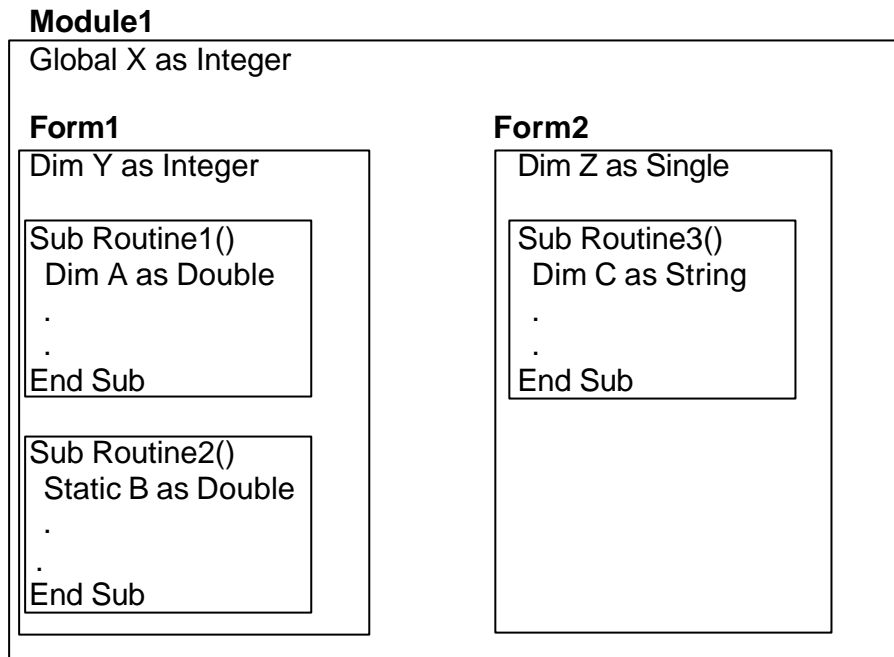
Procedure Routine1 has access to X, Y, and A (loses value upon termination)
Procedure Routine2 has access to X, Y, and B (retains value)
Procedure Routine3 has access to X, Z, and C (loses value)

<u>Example 1-3</u>

Stopwatch Application - Attaching Code

All that's left to do is attach code to the application. We write code for every event a response is needed for. In this application, there are three such events: clicking on each of the command buttons.

1. Double-click anywhere on the form to open the code window. Or, select 'View Code' from the project window.

2. Click the down arrow in the Object box and select the object named **(general)**. The Procedure box will show **(declarations)**. Here, you declare three form level variables:

```
Option Explicit
Dim StartTime As Variant
Dim EndTime As Variant
Dim ElapsedTime As Variant
```

The **Option Explicit** statement forces us to declare all variables. The other lines establish **StartTime**, **EndTime**, and **ElapsedTime** as variables global within the form.

3. Select the **cmdStart** object in the Object box. If the procedure that appears is not the Click procedure, choose **Click** from the procedure box. Type the following code which begins the timing procedure. Note the **Sub** and **End Sub** statements are provided for you:

```
Sub cmdStart_Click ()
'Establish and print starting time
StartTime = Now
lblStart.Caption = Format(StartTime, "hh:mm:ss")
lblEnd.Caption = ""
lblElapsed.Caption = ""
End Sub
```

In this procedure, once the **Start Timing** button is clicked, we read the current time and print it in a label box. We also blank out the other label boxes. In the code above (and in all code in these notes), any line beginning with a single quote (') is a comment. You decide whether you want to type these lines or not. They are not needed for proper application operation.

4. Now, code the **cmdEnd** button.

```
Sub cmdEnd_Click ()
'Find the ending time, compute the elapsed time
'Put both values in label boxes
EndTime = Now
ElapsedTime = EndTime - StartTime
lblEnd.Caption = Format(EndTime, "hh:mm:ss")
lblElapsed.Caption = Format(ElapsedTime, "hh:mm:ss")
End Sub
```

Here, when the **End Timing** button is clicked, we read the current time (**End Time**), compute the elapsed time, and put both values in their corresponding label boxes.

5. And, finally the **cmdExit** button.

```
Sub cmdExit_Click ()
End
End Sub
```

This routine simply ends the application once the **Exit** button is clicked.

6. Did you notice that as you typed in the code, Visual Basic does automatic syntax checking on what you type (if you made any mistakes, that is)?

7. Run your application by clicking the **Run** button on the toolbar, or by pressing <f5>. Pretty easy, wasn't it?

8. Save your application - see the **Primer** on the next page. Use the **Save Project As** option under the **File** menu. Make sure you save both the form and the project files.

9. If you have the time, some other things you may try with the Stopwatch Application:

    A. Try changing the form color and the fonts used in the label boxes and command buttons.

    B. Notice you can press the 'End Timing' button before the 'Start Timing' button. This shouldn't be so. Change the application so you can't do this. And make it such that you can't press the 'Start Timing' until 'End Timing' has been pressed. Hint: Look at the command button **Enabled** property.

    C. Can you think of how you can continuously display the 'End Time' and 'Elapsed Time'? This is a little tricky because of the event-driven nature of Visual Basic. Look at the **Timer** tool. Ask me for help on this one.

Quick Primer on Saving Visual Basic Applications:

When saving Visual Basic applications, you need to be concerned with saving both the forms (.FRM) and modules (.BAS) and the project file (.VBP). In either case, make sure you are saving in the desired directory. The current directory is always displayed in the Save window. Use standard Windows techniques to change the current directory.

There are four **Save** commands available under the **File** menu in Visual Basic:

Save [Form Name]	Save the currently selected form or module with the current name. The selected file is identified in the Project window.
Save [Form Name] As	Like Save File, however you have the option to change the file name
Save Project	Saves all forms and modules in the current project using their current names and also saves the project file.
Save Project As	Like Save Project, however you have the option to change file names. When you choose this option, if you have not saved your forms or modules, you will also be prompted to save those files. I always use this for new projects.

Exercise 1

Calendar/Time Display

Design a window that displays the current month, day, and year. Also, display the current time, updating it every second (look into the **Timer** control). Make the window look something like a calendar page. Play with object properties to make it pretty.

My Solution:

Form:
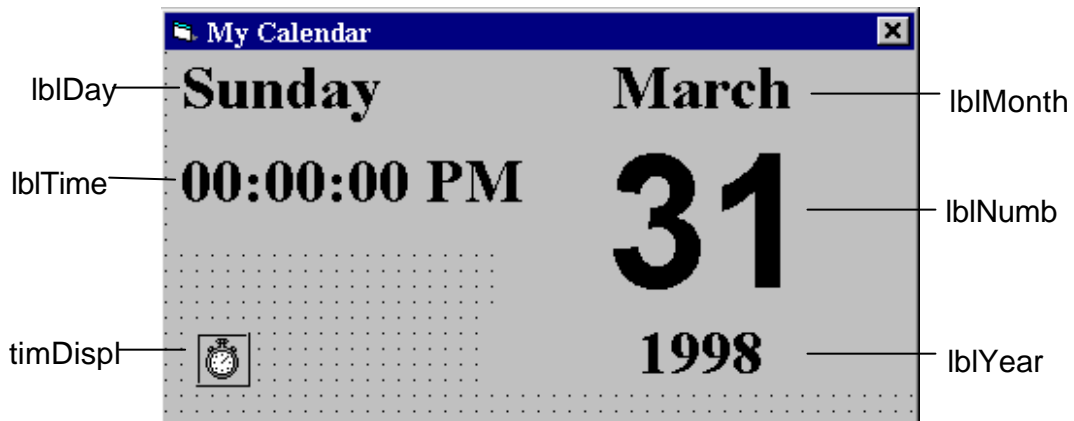
Properties:

Form **frmCalendar**:
      Caption = My Calendar
      BorderStyle = 1 - Fixed Single

Timer **timDisplay**:
      Interval = 1000

Label **lblDay**:
      Caption = Sunday
      FontName = Times New Roman
      FontBold = True
      FontSize = 24

Label **lblTime**:
Caption = 00:00:00 PM
FontName = Times New Roman
FontBold = True
FontSize = 24

Label **lblYear**:
Alignment = 2 - Center
Caption = 1998
FontName = Times New Roman
FontBold = True
FontSize = 24

Label **lblNumber**:
Alignment = 2 - Center
Caption = 31
FontName = Arial
FontBold = True
FontSize = 72

Label **lblMonth**:
Alignment = 2 - Center
Caption = March
FontName = Times New Roman
FontBold = True
FontSize = 24

Code:

General Declarations:

```
Option Explicit
```

timDisplay Timer Event:

```
Private Sub timDisplay_Timer()
Dim Today As Variant
Today = Now
lblDay.Caption = Format(Today, "dddd")
lblMonth.Caption = Format(Today, "mmmm")
lblYear.Caption = Format(Today, "yyyy")
lblnumber.Caption = Format(Today, "d")
lblTime.Caption = Format(Today, "h:mm:ss ampm")
End Sub
```

Learn Visual Basic 6.0

2. The Visual Basic Language

Review and Preview

• Last week, we found there were three primary steps involved in developing an application using Visual Basic:

    1. Draw the user interface
    2. Assign properties to controls
    3. Attach code to events

This week, we are primarily concerned with Step 3, attaching code. We will become more familiar with moving around in the Code window and learn some of the elements of the Basic language.

A Brief History of Basic

• Language developed in early 1960's at Dartmouth College:

    B (eginner's)
    A (All-Purpose)
    S (Symbolic)
    I (Instruction)
    C (Code)

• Answer to complicated programming languages (FORTRAN, Algol, Cobol ...). First timeshare language.

• In the mid-1970's, two college students write first Basic for a microcomputer (Altair) - cost $350 on cassette tape. You may have heard of them: Bill Gates and Paul Allen!

• Every Basic since then essentially based on that early version. Examples include: GW-Basic, QBasic, QuickBasic.

• Visual Basic was introduced in 1991.

Visual Basic Statements and Expressions

- The simplest statement is the **assignment** statement. It consists of a variable name, followed by the assignment operator (=), followed by some sort of **expression**.

    Examples:

        StartTime = Now
        Explorer.Caption = "Captain Spaulding"
        BitCount = ByteCount * 8
        Energy = Mass * LIGHTSPEED ^ 2
        NetWorth = Assets - Liabilities

    The assignment statement stores information.

- Statements normally take up a single line with no terminator. Statements can be **stacked** by using a colon (**:**) to separate them. Example:

        StartTime = Now : EndTime = StartTime + 10

    (Be careful stacking statements, especially with If/End If structures. You may not get the response you desire.)

- If a statement is very long, it may be continued to the next line using the **continuation** character, an underscore (_). Example:

        Months = Log(Final * IntRate / Deposit + 1) _
        / Log(1 + IntRate)

- Comment statements begin with the keyword **Rem** or a single quote (**'**). For example:

        Rem This is a remark
        ' This is also a remark
        x = 2 * y ' another way to write a remark or comment

    You, as a programmer, should decide how much to comment your code. Consider such factors as reuse, your audience, and the legacy of your code.

Visual Basic Operators

- The simplest **operators** carry out **arithmetic** operations. These operators in their order of precedence are:

Operator	Operation
^	Exponentiation
* /	Multiplication and division
\	Integer division (truncates)
Mod	Modulus
+ -	Addition and subutraction

- **Parentheses** around expressions can change precedence.

- To **concatentate** two strings, use the **&** symbol or the **+** symbol:

    lblTime.Caption = "The current time is" & Format(Now, "hh:mm")
    txtSample.Text = "Hook this " + "to this"

- There are six **comparison** operators in Visual Basic:

Operator	Comparison
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to

- The result of a comparison operation is a Boolean value (**True** or **False**).

- We will use three **logical** operators

Operator	Operation
Not	Logical not
And	Logical and
Or	Logical or

- The **Not** operator simply negates an operand.

- The **And** operator returns a True if both operands are True. Else, it returns a False.

- The **Or** operator returns a True if either of its operands is True, else it returns a False.

- Logical operators follow arithmetic operators in precedence.

Visual Basic Functions

- Visual Basic offers a rich assortment of built-in **functions**. The on-line help utility will give you information on any or all of these functions and their use. Some examples are:

Function	Value Returned
Abs	Absolute value of a number
Asc	ASCII or ANSI code of a character
Chr	Character corresponding to a given ASCI or ANSI code
Cos	Cosine of an angle
Date	Current date as a text string
Format	Date or number converted to a text string
Left	Selected left side of a text string
Len	Number of characters in a text string
Mid	Selected portion of a text stri ng
Now	Current time and date
Right	Selected right end of a text string
Rnd	Random number
Sin	Sine of an angle
Sqr	Square root of a number
Str	Number converted to a text string
Time	Current time as a text string
Timer	Number of seconds elapsed since midnight
Val	Numeric value of a given text string

A Closer Look at the Rnd Function

- In writing games and learning software, we use the **Rnd** function to introduce randomness. This insures different results each time you try a program. 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, use the formula:

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

- 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

    Place this statement in the **Form_Load** event procedure.

- **Examples**:

    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

Example 2-1

Savings Account

1. Start a new project. The idea of this project is to determine how much you save by making monthly deposits into a savings account. For those interested, the mathematical formula used is:

$$F = D \left[(1 + I)^M - 1 \right] / I$$

where

F - Final amount
D - Monthly deposit amount
I - Monthly interest rate
M - Number of months

2. Place 4 label boxes, 4 text boxes, and 2 command buttons on the form. It should look something like this:
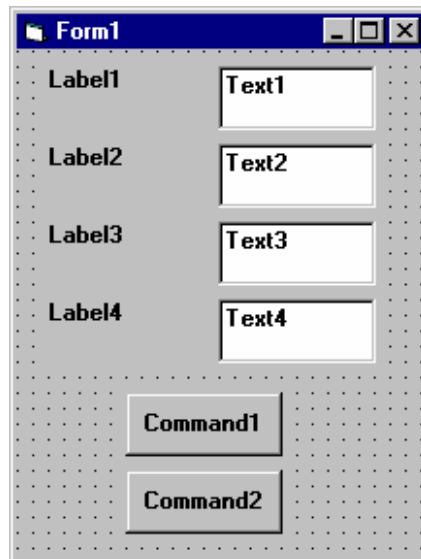
3. Set the properties of the form and each object.

Form1:
BorderStyle	1-Fixed Single
Caption	Savings Account
Name	frmSavings

Label1:
Caption	Monthly Deposit

Label2:
Caption	Yearly Interest

Label3:
Caption	Number of Months

Label4:
Caption	Final Balance

Text1:
Text	[Blank]
Name	txtDeposit

Text2:
Text	[Blank]
Name	txtInterest

Text3:
Text	[Blank]
Name	txtMonths

Text4:
Text	[Blank]
Name	txtFinal

Command1:
Caption	&Calculate
Name	cmdCalculate

Command2:
Caption	E&xit
Name	cmdExit

Now, your form should look like this:
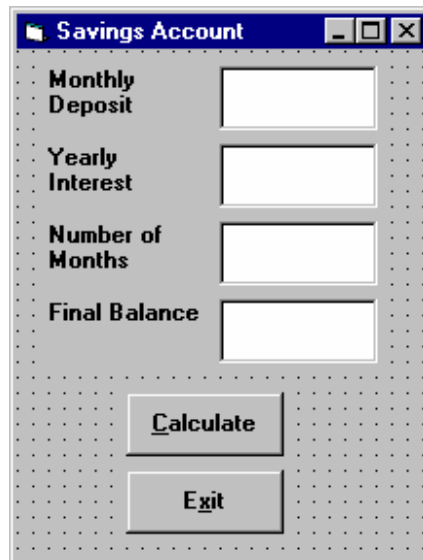
4. Declare four variables in the **general declarations** area of your form. This makes them available to all the form procedures:

```
Option Explicit
Dim Deposit As Single
Dim Interest As Single
Dim Months As Single
Dim Final As Single
```

The **Option Explicit** statement forces us to declare all variables.

5. Attach code to the **cmdCalculate** command button **Click** event.

```
Private Sub cmdCalculate_Click ()
Dim IntRate As Single
'Read values from text boxes
Deposit = Val(txtDeposit.Text)
Interest = Val(txtInterest.Text)
IntRate = Interest / 1200
Months = Val(txtMonths.Text)
'Compute final value and put in text box
Final = Deposit * ((1 + IntRate) ^ Months - 1) / IntRate
txtFinal.Text = Format(Final, "#####0.00")
End Sub
```

This code reads the three input values (monthly deposit, interest rate, number of months) from the text boxes, computes the final balance using the provided formula, and puts that result in a text box.

6. Attach code to the **cmdExit** command button **Click** event.

```
Private Sub cmdExit_Click ()
End
End Sub
```

7. Play with the program. Make sure it works properly. Save the project.

Visual Basic Symbolic Constants

* Many times in Visual Basic, functions and objects require data arguments that affect their operation and return values you want to read and interpret. These arguments and values are constant numerical data and difficult to interpret based on just the numerical value. To make these constants more understandable, Visual Basic assigns names to the most widely used values - these are called **symbolic constants**. Appendix I lists many of these constants.

* As an example, to set the background color of a form named **frmExample** to blue, we could type:

    frmExample.BackColor = 0xFF0000

    or, we could use the symbolic constant for the blue color (**vbBlue**):

    frmExample.BackColor = vbBlue

* It is strongly suggested that the symbolic constants be used instead of the numeric values, when possible. You should agree that **vbBlue** means more than the value **0xFF0000** when selecting the background color in the above example. You do not need to do anything to define the symbolic constants - they are built into Visual Basic.

Defining Your Own Constants

* You can also define your own constants for use in Visual Basic. The format for defining a constant named **PI** with a value **3.14159** is:

    Const PI = 3.14159

* **User-defined constants** should be written in all upper case letters to distinguish them from variables. The scope of constants is established the same way a variables' scope is. That is, if defined within a procedure, they are local to the procedure. If defined in the general declarations of a form, they are global to the form. To make constants global to an application, use the format:

    Global Const PI = 3.14159

    within the general declarations area of a module.

Visual Basic Branching - If Statements

- **Branching** statements are used to cause certain actions within a program if a certain condition is met.

- The simplest is the **If/Then** statement:

      If Balance - Check < 0 Then Print "You are overdrawn"

  Here, if and only if Balance - Check is less than zero, the statement "You are overdrawn" is printed.

- You can also have **If/Then/End If** blocks to allow multiple statements:

      If Balance - Check < 0 Then
        Print "You are overdrawn"
        Print "Authorities have been notified"
      End If

  In this case, if Balance - Check is less than zero, two lines of information are printed.

- Or, **If/Then/Else/End If** blocks:

      If Balance - Check < 0 Then
        Print "You are overdrawn"
        Print "Authorities have been notified"
      Else
        Balance = Balance - Check
      End If

  Here, the same two lines are printed if you are overdrawn (Balance - Check < 0), but, if you are not overdrawn (**Else**), your new Balance is computed.

- Or, we can add the **ElseIf** statement:

      If Balance - Check < 0 Then
        Print "You are overdrawn"
        Print "Authorities have been notified"
      ElseIf Balance - Check = 0 Then
        Print "Whew! You barely made it"
        Balance = 0
      Else
        Balance = Balance - Check
      End If

  Now, one more condition is added. If your Balance equals the Check amount (**ElseIf** Balance - Check = 0), a different message appears.

- In using branching statements, make sure you consider all viable possibilities in the If/Else/End If structure. Also, be aware that each If and ElseIf in a block is tested sequentially. The first time an If test is met, the code associated with that condition is executed and the If block is exited. If a later condition is also True, it will never be considered.

Key Trapping

- Note in the previous example, there is nothing to prevent the user from typing in meaningless characters (for example, letters) into the text boxes expecting numerical data. Whenever getting input from a user, we want to limit the available keys they can press. The process of interecepting unacceptable keystrokes is **key trapping**.

- Key trapping is done in the **KeyPress** procedure of an object. Such a procedure has the form (for a text box named **txtText**):

      Sub txtText_KeyPress (KeyAscii as Integer)
           .
           .
           .
      End Sub

  What happens in this procedure is that every time a key is pressed in the corresponding text box, the ASCII code for the pressed key is passed to this procedure in the argument list (i.e. **KeyAscii**). If KeyAscii is an acceptable value, we would do nothing. However, if KeyAscii is not acceptable, we would set KeyAscii equal to zero and exit the procedure. Doing this has the same result of not pressing a key at all.   ASCII values for all keys are available via the on-line help in Visual Basic. And some keys are also defined

by symbolic constants. Where possible, we will use symbolic constants; else, we will use the ASCII values.

• As an example, say we have a text box (named **txtExample**) and we only want to be able to enter upper case letters (ASCII codes 65 through 90, or, correspondingly, symbolic constants **vbKeyA** through **vbKeyZ**). The key press procedure would look like (the **Beep** causes an audible tone if an incorrect key is pressed):

```
Sub txtExample_KeyPress(KeyAscii as Integer)
  If KeyAscii >= vbKeyA And KeyAscii <= vbKeyZ Then
    Exit Sub
  Else
    KeyAscii = 0
    Beep
  End If
End Sub
```

• In key trapping, it's advisable to always allow the backspace key (ASCII code 8; symbolic constant **vbKeyBack**) to pass through the key press event. Else, you will not be able to edit the text box properly.

Example 2-2

Savings Account - Key Trapping

1. Note the acceptable ASCII codes are 48 through 57 (numbers), 46 (the decimal point), and 8 (the backspace key). In the code, we use symbolic constants for the numbers and backspace key. Such a constant does not exist for the decimal point, so we will define one with the following line in the **general declarations** area:

```
Const vbKeyDecPt = 46
```

2. Add the following code to the three procedures: **txtDeposit_KeyPress**, **txtInterest_KeyPress**, and **txtMonths_KeyPress**.

```
Private Sub txtDeposit_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or
KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then
   Exit Sub
Else
 KeyAscii = 0
 Beep
End If
End Sub

Private Sub txtInterest_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or
KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then
   Exit Sub
Else
 KeyAscii = 0
 Beep
End If
End Sub
```

```
Private Sub txtMonths_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or
KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then
   Exit Sub
Else
 KeyAscii = 0
 Beep
End If
End Sub
```

(In the If statements above, note the word processor causes a line break where there really shouldn't be one. That is, there is no line break between the words **Or KeyAscii** and **= vbKeyDecPt**. One appears due to page margins. In all code in these notes, always look for such things.)

3. Rerun the application and test the key trapping performance.

Select Case - Another Way to Branch

- In addition to If/Then/Else type statements, the **Select Case** format can be used when there are multiple selection possibilities.

- Say we've written this code using the **If** statement:

```
If Age = 5 Then
  Category = "Five Year Old"
ElseIf Age >= 13 and Age <= 19 Then
  Category = "Teenager"
ElseIf (Age >= 20 and Age <= 35) Or Age = 50 Or (Age >= 60 and Age <=
65) Then
  Category = "Special Adult"
ElseIf Age > 65 Then
  Category = "Senior Citizen"
Else
  Category = "Everyone Else"
End If
```

  The corresponding code with **Select Case** would be:

```
Select Case Age
  Case 5
    Category = "Five Year Old"
  Case 13 To 19
    Category = "Teenager"
  Case 20 To 35, 50, 60 To 65
    Category = "Special Adult"
  Case Is > 65
    Category = "Senior Citizen"
  Case Else
    Category = "Everyone Else"
End Select
```

  Notice there are several formats for the Case statement. Consult on-line help for discussions of these formats.

The GoTo Statement

- Another branching statement, and perhaps the most hated statement in programming, is the **GoTo** statement. However, we will need this to do Run-Time error trapping. The format is **GoTo *Label***, where ***Label*** is a labeled line. Labeled lines are formed by typing the *Label* followed by a colon.

- **GoTo** Example:

```
  Line10:
     .
     .
  GoTo Line10
```
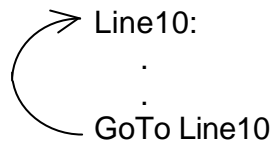
When the code reaches the GoTo statement, program control transfers to the line labeled Line10.

Visual Basic Looping

- Looping is done with the **Do/Loop** format. Loops are used for operations are to be repeated some number of times. The loop repeats until some specified condition at the beginning or end of the loop is met.

- **Do While/Loop** Example:

```
Counter = 1
Do While Counter <= 1000
  Debug.Print Counter
  Counter = Counter + 1
Loop
```

This loop repeats as long as (**While**) the variable Counter is less than or equal to 1000. Note a Do While/Loop structure will not execute even once if the While condition is violated (False) the first time through. Also note the **Debug.Print** statement. What this does is print the value Counter in the Visual Basic Debug window. We'll learn more about this window later in the course.

- **Do Until/Loop** Example:

  ```
  Counter = 1
  Do Until Counter >  1000
    Debug.Print Counter
    Counter = Counter + 1
  Loop
  ```

  This loop repeats **Until** the Counter variable exceeds 1000. Note a Do Until/Loop structure will not be entered if the Until condition is already True on the first encounter.

- **Do/Loop While** Example:

  ```
  Sum = 1
  Do
    Debug.Print Sum
    Sum = Sum + 3
  Loop While Sum <= 50
  ```

  This loop repeats **While** the Variable Sum is less than or equal to 50. Note, since the While check is at the end of the loop, a Do/Loop While structure is always executed at least once.

- **Do/Loop Until** Example:

  ```
  Sum = 1
  Do
    Debug.Print Sum
    Sum = Sum + 3
  Loop Until Sum > 50
  ```

  This loop repeats Until Sum is greater than 50. And, like the previous example, a Do/Loop Until structure always executes at least once.

- Make sure you can always get out of a loop! Infinite loops are never nice. If you get into one, try **Ctrl+Break**. That sometimes works - other times the only way out is rebooting your machine!

- The statement **Exit Do** will get you out of a loop and transfer program control to the statement following the Loop statement.

Visual Basic Counting

- Counting is accomplished using the **For/Next** loop.

    Example

    ```
    For I = 1 to 50 Step 2
     A = I * 2
      Debug.Print A
    Next I
    ```

    In this example, the variable I initializes at 1 and, with each iteration of the For/Next loop, is incremented by 2 (**Step**). This looping continues until I becomes greater than or equal to its final value (50). If Step is not included, the default value is 1. Negative values of Step are allowed.

- You may exit a For/Next loop using an **Exit For** statement. This will transfer program control to the statement following the **Next** statement.

Example 2-3

Savings Account - Decisions

1. Here, we modify the Savings Account project to allow entering any three values and computing the fourth. First, add a third command button that will clear all of the text boxes. Assign the following properties:

> **Command3:**
> | | |
> |---|---|
> | Caption | Clear &Boxes |
> | Name | cmdClear |

The form should look something like this when you're done:
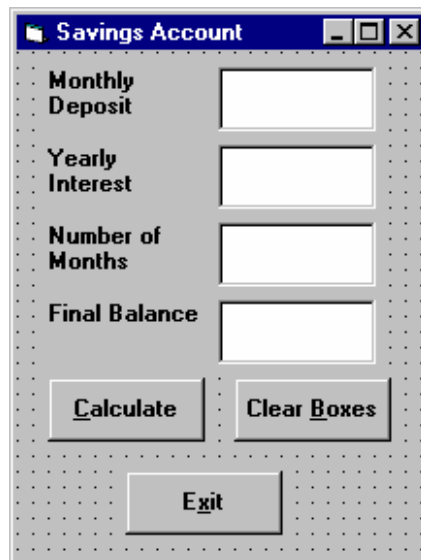
2. Code the **cmdClear** button **Click** event:

```
Private Sub cmdClear_Click ()
'Blank out the text boxes
txtDeposit.Text = ""
txtInterest.Text = ""
txtMonths.Text = ""
txtFinal.Text = ""
End Sub
```

This code simply blanks out the four text boxes when the **Clear** button is clicked.

3. Code the **KeyPress** event for the **txtFinal** object:

```
Private Sub txtFinal_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or
KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then
   Exit Sub
Else
  KeyAscii = 0
  Beep
End If
End Sub
```

We need this code because we can now enter information into the Final Value text box.

4. The modified code for the **Click** event of the **cmdCalculate** button is:

```
Private Sub cmdCalculate_Click()
Dim IntRate As Single
Dim IntNew As Single
Dim Fcn As Single, FcnD As Single
'Read the four text boxes
Deposit = Val(txtDeposit.Text)
Interest = Val(txtInterest.Text)
IntRate = Interest / 1200
Months = Val(txtMonths.Text)
Final = Val(txtFinal.Text)
'Determine which box is blank
'Compute that missing value and put in text box
If txtDeposit.Text = "" Then
'Deposit missing
  Deposit = Final / (((1 + IntRate) ^ Months - 1) /
  IntRate)
  txtDeposit.Text = Format(Deposit, "#####0.00")
ElseIf txtInterest.Text = "" Then
'Interest missing - requires iterative solution
  IntNew = (Final / (0.5* Months * Deposit) - 1) / Months
  Do
    IntRate = IntNew
    Fcn = (1 + IntRate) ^ Months - Final * IntRate /
  Deposit - 1
    FcnD = Months * (1 + IntRate) ^ (Months - 1) - Final /
  Deposit
    IntNew = IntRate - Fcn / FcnD
  Loop Until Abs(IntNew - IntRate) < 0.00001 / 12
  Interest = IntNew * 1200
```

```
  txtInterest.Text = Format(Interest, "##0.00")
ElseIf txtMonths.Text = "" Then
'Months missing
  Months = Log(Final * IntRate / Deposit + 1) / Log(1 +
  IntRate)
  txtMonths.Text = Format(Months, "###.0")
ElseIf txtFinal.Text = "" Then
'Final value missing
  Final = Deposit * ((1 + IntRate) ^ Months - 1) / IntRate
  txtFinal.Text = Format(Final, "#####0.00")
End If
End Sub
```

In this code. we first read the text information from all four text boxes and based on which one is blank, compute the missing information and display it in the corresponding text box. Solving for missing **Deposit**, **Months**, or **Final** information is a straightforward manipulation of the equation given in Example 2-2.

If the **Interest** value is missing, we have to solve an Mth-order polynomial using something called Newton-Raphson iteration - a good example of using a Do loop. Finding the **Interest** value is straightforward. What we do is guess at what the interest is, compute a better guess (using Newton-Raphson iteration), and repeat the process (loop) until the old guess and the new guess are close to each other. You can see each step in the code.

5. Test and save your application. Go home and relax.

Exercise 2-1

Computing a Mean and Standard Deviation

Develop an application that allows the user to input a sequence of numbers. When done inputting the numbers, the program should compute the mean of that sequence and the standard deviation. If N numbers are input, with the ith number represented by x_i, the formula for the mean ($\bar{x}$) is:

$$\bar{x} = (\sum_{i=1}^{N} x_i)/N$$

and to compute the standard deviation (s), take the square root of this equation:

$$s^2 = [N\sum_{i=1}^{N} x_i^2 - (\sum_{i=1}^{N} x_i)^2]/[N(N-1)]$$

The Greek sigmas in the above equations simply indicate that you add up all the corresponding elements ne xt to the sigma.

My Solution:

Form:
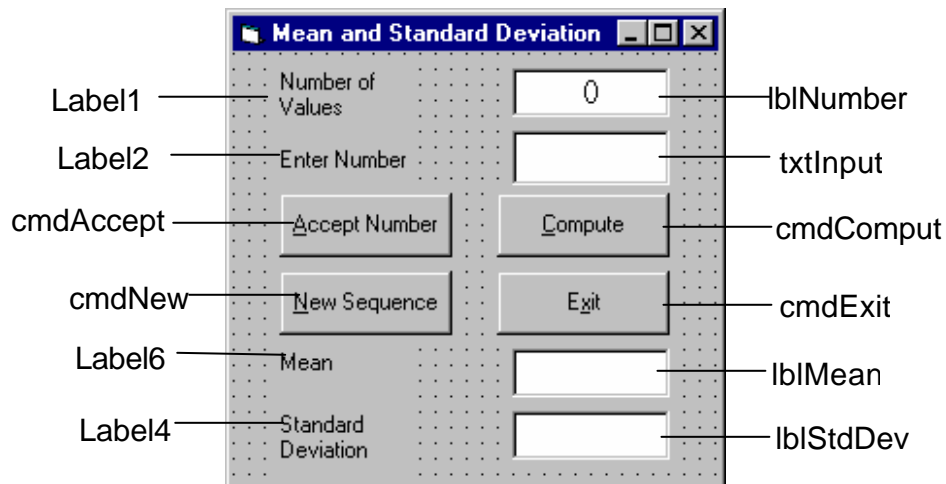
Properties:

Form **frmStats**:
    Caption = Mean and Standard Deviation

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

CommandButton **cmdAccept**:
    Caption = &Accept Number

CommandButton **cmdCompute**:
    Caption = &Compute

CommandButton **cmdNew**:
    Caption = &New Sequence

TextBox **txtInput**:
    FontName = MS Sans Serif
    FontSize = 12

Label **lblStdDev**:
    Alignment = 2 - Center
    BackColor = &H00FFFFFF& (White)
    BorderStyle = 1 - Fixed Single
    FontName = MS Sans Serif
    FontSize = 12

Label **Label6**:
    Caption = Standard Deviation

Label **lblMean**:
    Alignment = 2 - Center
    BackColor = &H00FFFFFF& (White)
    BorderStyle = 1 - Fixed Single
    FontName = MS Sans Serif
    FontSize = 12

Label **Label4**:
    Caption = Mean

Label **lblNumber**:
    Alignment = 2 - Center
    BackColor = &H00FFFFFF& (White)
    BorderStyle = 1 - Fixed Single
    FontName = MS Sans Serif
    FontSize = 12

Label **Label2**:
    Caption = Enter Number

Label **Label1**:
    Caption = Number of Values

Code:

General Declarations:

```
Option Explicit
Dim NumValues As Integer
Dim SumX As Single
Dim SumX2 As Single
Const vbKeyMinus = 45
Const vbKeyDecPt = 46
```


cmdAccept Click Event:

```
Private Sub cmdAccept_Click()
Dim Value As Single
txtInput.SetFocus
NumValues = NumValues + 1
lblNumber.Caption = Str(NumValues)
'Get number and sum number and number-squared
Value = Val(txtInput.Text)
SumX = SumX + Value
SumX2 = SumX2 + Value ^ 2
txtInput.Text = ""
End Sub
```

cmdCompute Click Event:

```
Private Sub cmdCompute_Click()
Dim Mean As Single
Dim StdDev As Single
txtInput.SetFocus
'Make sure there are at least two values
If NumValues < 2 Then
  Beep
  Exit Sub
End If
'Compute mean
Mean = SumX / NumValues
lblMean.Caption = Str(Mean)
'Compute standard deviation
StdDev = Sqr((NumValues * SumX2 - SumX ^ 2) / (NumValues *
(NumValues - 1)))
lblStdDev.Caption = Str(StdDev)
End Sub
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
End
End Sub
```

cmdNew Click Event:

```
Private Sub cmdNew_Click()
'Initialize all variables
txtInput.SetFocus
NumValues = 0
lblNumber.Caption = "0"
txtInput.Text = ""
lblMean.Caption = ""
lblStdDev.Caption = ""
SumX = 0
SumX2 = 0
End Sub
```

txtInput KeyPress Event:

```
Private Sub txtInput_KeyPress(KeyAscii As Integer)
'Only allow numbers, minus sign, decimal point, backspace,
return keys
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii
= vbKeyMinus Or KeyAscii = vbKeyDecPt Or KeyAscii =
vbKeyBack Then
  Exit Sub
ElseIf KeyAscii = vbKeyReturn Then
  Call cmdAccept_Click
Else
  KeyAscii = 0
End If
End Sub
```

Exercise 2-2

Flash Card Addition Problems

Write an application that generates random addition problems. Provide some kind of feedback and scoring system as the problems are answered.

My Solution:

Form:
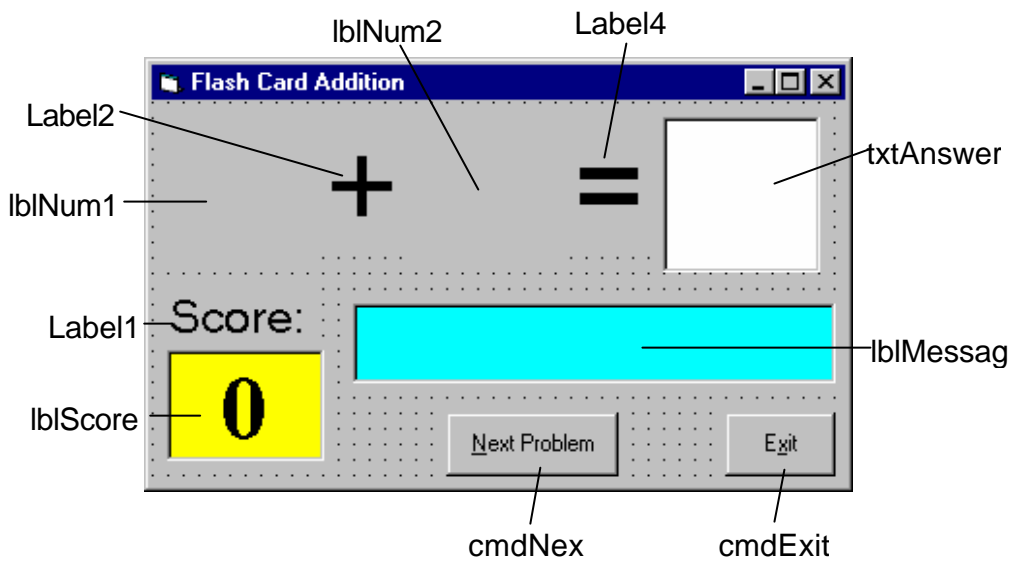
Properties:

Form **frmAdd**:
    BorderStyle = 1 - Fixed Single
    Caption = Flash Card Addition

CommandButton **cmdNext**:
    Caption = &Next Problem
    Enabled = False

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

TextBox **txtAnswer**:
    FontName = Arial
    FontSize = 48
    MaxLength = 2

Label **lblMessage**:
> Alignment = 2 - Center
> BackColor = &H00FFFF00& (Cyan)
> BorderStyle = 1 - Fixed Single
> FontName = MS Sans Serif
> FontBold = True
> FontSize = 24
> FontItalic = True

Label **lblScore**:
> Alignment = 2 - Center
> BackColor = &H0000FFFF& (Yellow)
> BorderStyle = 1 - Fixed Single
> Caption = 0
> FontName = Times New Roman
> FontBold = True
> FontSize = 36

Label **Label1**:
> Alignment = 2 - Center
> Caption = Score:
> FontName = MS Sans Serif
> FontSize = 18

Label **Label4**:
> Alignment = 2 - Center
> Caption = =
> FontName = Arial
> FontSize = 48

Label **lblNum2**:
> Alignment = 2 - Center
> FontName = Arial
> FontSize = 48

Label **Label2**:
> Alignment = 2 - Center
> Caption = +
> FontName = Arial
> FontSize = 48

Label **lblNum1**:
> Alignment = 2 - Center
> FontName = Arial
> FontSize = 48

Code:

General Declarations:

```
Option Explicit
Dim Sum As Integer
Dim NumProb As Integer, NumRight As Integer
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
End
End Sub
```

cmdNext Click Event:

```
Private Sub cmdNext_Click()
'Generate next addition problem
Dim Number1 As Integer
Dim Number2 As Integer
txtAnswer.Text = ""
lblMessage.Caption = ""
NumProb = NumProb + 1
'Generate random numbers for addends
Number1 = Int(Rnd * 21)
Number2 = Int(Rnd * 21)
lblNum1.Caption = Format(Number1, "#0")
lblNum2.Caption = Format(Number2, "#0")
'Find sum
Sum = Number1 + Number2
cmdNext.Enabled = False
txtAnswer.SetFocus
End Sub
```

Form Activate Event:

```
Private Sub Form_Activate()
Call cmdNext_Click
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
Randomize Timer
NumProb = 0
NumRight = 0
End Sub
```

txtAnswer KeyPress Event:

```
Private Sub txtAnswer_KeyPress(KeyAscii As Integer)
Dim Ans As Integer
'Check for number only input and for return key
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii
= vbKeyBack Then
  Exit Sub
ElseIf KeyAscii = vbKeyReturn Then
'Check answer
  Ans = Val(txtAnswer.Text)
  If Ans = Sum Then
    NumRight = NumRight + 1
    lblMessage.Caption = "That's correct!"
  Else
    lblMessage.Caption = "Answer is " + Format(Sum, "#0")
  End If
  lblScore.Caption = Format(100 * NumRight / NumProb,
"##0")
  cmdNext.Enabled = True
  cmdNext.SetFocus
Else
  KeyAscii = 0
End If
End Sub
```

This page intentionally not left blank.

Learn Visual Basic 6.0

3. Exploring the Visual Basic Toolbox

Review and Preview

- In this class, we begin a journey where we look at each tool in the Visual Basic toolbox. We will revisit some tools we already know and learn a lot of new tools. First, though, we look at an important Visual Basic functions.

The Message Box

- One of the best functions in Visual Basic is the **message box**. The message box displays a message, optional icon, and selected set of command buttons. The user responds by clicking a button.

- The **statement** form of the message box returns no value (it simply displays the box):

    MsgBox Message, Type, Title

where

Message	Text message to be displayed
Type	Type of message box (discussed in a bit)
Title	Text in title bar of message box

You have no control over where the message box appears on the screen.

- The **function** form of the message box returns an integer value (corresponding to the button clicked by the user). Example of use (Response is returned value):

    Dim Response as Integer
    Response = **MsgBox**(Message, Type, Title)

- The **Type** argument is formed by summing four values corresponding to the buttons to display, any icon to show, which button is the default response, and the modality of the message box.
- The first component of the **Type** value specifies the **buttons** to display:

Value	Meaning	Symbolic Constant
0	OK button only	vbOKOnly
1	OK/Cancel buttons	vbOKCancel
2	Abort/Retry/Ignore buttons	vbAbortRetryIgnore
3	Yes/No/Cancel buttons	vbYesNoCancel
4	Yes/No buttons	vbYesNo
5	Retry/Cancel buttons	vbRetryCancel

- The second component of **Type** specifies the **icon** to display in the message box:

Value	Meaning	Symbolic Constant
0	No icon	(None)
16	Critical icon	vbCritical
32	Question mark	vbQuestion
48	Exclamation point	vbExclamation
64	Information icon	vbInformation

- The third component of **Type** specifies which button is **default** (i.e. pressing Enter is the same as clicking the default button):

Value	Meaning	Symbolic Constant
0	First button default	vbDefaultButton1
256	Second button default	vbDefaultButton2
512	Third button default	vbDefaultButton3

- The fourth and final component of **Type** specifies the **modality**:

Value	Meaning	Symbolic Constant
0	Application modal	vbApplicationModal
4096	System modal	vbSystemModal

  If the box is **Application Modal**, the user must respond to the box before continuing work in the current application. If the box is **System Modal**, all applications are suspended until the user responds to the message box.

- Note for each option in **Type**, there are numeric values listed and symbolic constants. Recall, it is strongly suggested that the symbolic constants be used instead of the numeric values. You should agree that **vbOKOnly** means more than the number 0 when selecting the button type.

• The value returned by the function form of the message box is related to the button clicked:

Value	Meaning	Symbolic Constant
1	OK button selected	vbOK
2	Cancel button selected	vbCancel
3	Abort button selected	vbAbort
4	Retry button selected	vbRetry
5	Ignore button selected	vbIgnore
6	Yes button selected	vbYes
7	No button selected	vbNo

• Message Box Example:

   MsgBox "This is an example of a message box", vbOKCancel +
   vbInformation, "Message Box Example"
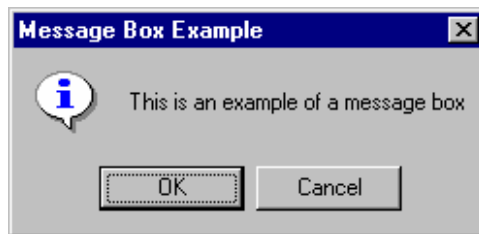
• You've seen message boxes if you've ever used a Windows application. Think of all the examples you've seen. For example, message boxes are used to ask you if you wish to save a file before exiting and to warn you if a disk drive is not ready.

Object Methods

• In previous work, we have seen that each object has properties and events associated with it. A third concept associated with objects is the **method**. A method is a procedure or function that imparts some action to an object.

• As we move through the toolbox, when appropriate, we'll discuss object methods. Methods are always enacted at run-time in code. The format for invoking a method is:

   ObjectName.Method {optional arguments}

   Note this is another use of the dot notation.

The Form Object

- The **Form** is where the user interface is drawn. It is central to the development of Visual Basic applications.

- Form Properties:

Appearance	Selects 3-D or flat appearance.
BackColor	Sets the form background color.
BorderStyle	Sets the form border to be fixed or sizeable.
Caption	Sets the form window title.
Enabled	If True, allows the form to respond to mouse and keyboard events; if False, disables form.
Font	Sets font type, style, size.
ForeColor	Sets color of text or graphics.
Picture	Places a bitmap picture in the form.
Visible	If False, hides the form.

- Form Events:

Activate	Form_Activate event is triggered when form becomes the active window.
Click	Form_Click event is triggered when user clicks on form.
DblClick	Form_DblClick event is triggered when user double-clicks on form.
Load	Form_Load event occurs when form is loaded. This is a good place to initialize variables and set any run-time properties.

- Form Methods:

Cls	Clears all graphics and text from form. Does not clear any objects.
Print	Prints text string on the form.

Examples

    frmExample.Cls ' clears the form
    frmExample.Print "This will print on the form"

Command Buttons

- We've seen the **command button** before. It is probably the most widely used control. It is used to begin, interrupt, or end a particular process.

- Command Button Properties:

Appearance	Selects 3-D or flat appearance.
Cancel	Allows selection of button with **Esc** key (only one button on a form can have this property True).
Caption	String to be displayed on button.
Default	Allows selection of button with **Enter** key (only one button on a form can have this property True).
Font	Sets font type, style, size.

- Command Button Events:

Click	Event triggered when button is selected either by clicking on it or by pressing the access key.

Label Boxes

- A **label box** is a control you use to display text that a user can't edit directly. We've seen, though, in previous examples, that the text of a label box can be changed at run-time in response to events.

- Label Properties:

Alignment	Aligns caption within border.
Appearance	Selects 3-D or flat appearance.
AutoSize	If True, the label is resized to fit the text specifed by the caption property. If False, the label will remain the size defined at design time and the text may be clipped.
BorderStyle	Determines type of border.
Caption	String to be displayed in box.
Font	Sets font type, style, size.

WordWrap	Works in conjunction with AutoSize property. If AutoSize = True, WordWrap = True, then the text will wrap and label will expand vertically to fit the Caption. If AutoSize = True, WordWrap = False, then the text will not wrap and the label expands horizontally to fit the Caption. If AutoSize = False, the text will not wrap regardless of WordWrap value.

* Label Events:

Click	Event triggered when user clicks on a label.
DblClick	Event triggered when user double-clicks on a label.

Text Boxes

* A **text box** is used to display information entered at design time, by a user at run-time, or assigned within code. The displayed text may be edited.

* Text Box Properties:

Appearance	Selects 3-D or flat appearance.
BorderStyle	Determines type of border.
Font	Sets font type, style, size.
MaxLength	Limits the length of displayed text (0 value indicates unlimited length).
MultiLine	Specifies whether text box displays single line or multiple lines.
PasswordChar	Hides text with a single character.
ScrollBars	Specifies type of displayed scroll bar(s).
SelLength	Length of selected text (run-time only).
SelStart	Starting position of selected text (run-time only).
SelText	Selected text (run-time only).
Tag	Stores a string expression.
Text	Displayed text.

- Text Box Events:

Change	Triggered every time the **Text** property changes.
LostFocus	Triggered when the user leaves the text box. This is a good place to examine the contents of a text box after editing.
KeyPress	Triggered whenever a key is pressed. Used for key trapping, as seen in last class.

- Text Box Methods:

SetFocus	Places the cursor in a specified text box.

  Example

  txtExample.SetFocus ' moves cursor to box named txtExample

Example 3-1

Password Validation

1. Start a new project. The idea of this project is to ask the user to input a password. If correct, a message box appears to validate the user. If incorrect, other options are provided.

2. Place a two command buttons, a label box, and a text box on your form so it looks something like this:
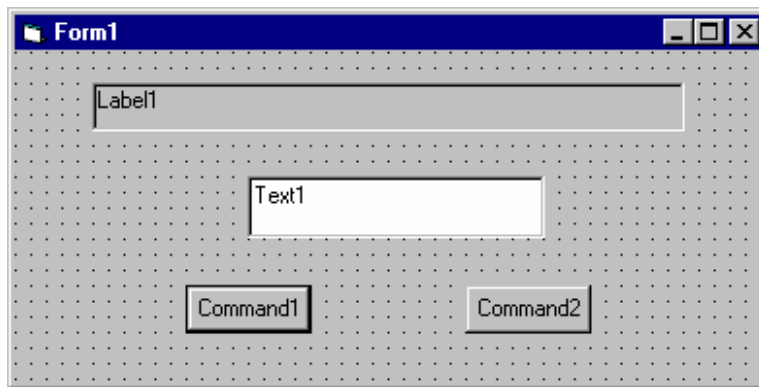
3. Set the properties of the form and each object.

    Form1:
BorderStyle	1-Fixed Single
Caption	Password Validation
Name	frmPassword

    Label1:
Alignment	2-Center
BorderStyle	1-Fixed Single
Caption	Please Enter Your Password:
FontSize	10
FontStyle	Bold

    Text1:
FontSize	14
FontStyle	Regular
Name	txtPassword
PasswordChar	*
Tag	[Whatever you choose as a password]
Text	[Blank]

Command1:

Caption	&Validate
Default	True
Name	cmdValid

Command2:

Cancel	True
Caption	E&xit
Name	cmdExit

Your form should now look like this:
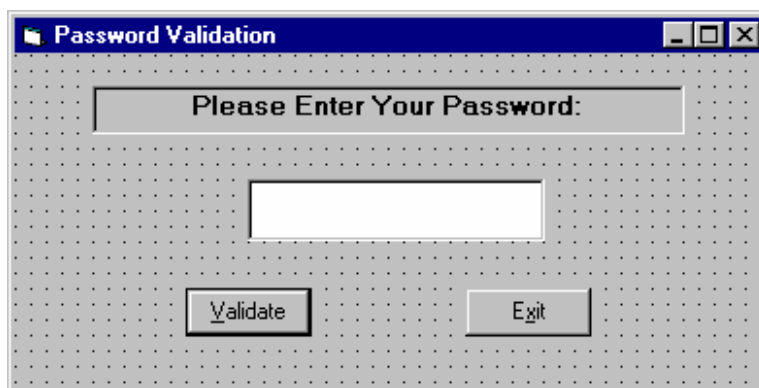
4. Attach the following code to the **cmdValid_Click** event.

```
Private Sub cmdValid_Click()
'This procedure checks the input password
Dim Response As Integer
If txtPassword.Text = txtPassword.Tag Then
'If correct, display message box
  MsgBox "You've passed security!", vbOKOnly +
  vbExclamation, "Access Granted"
Else
'If incorrect, give option to try again
  Response = MsgBox("Incorrect password", vbRetryCancel +
  vbCritical, "Access Denied")
  If Response = vbRetry Then
    txtPassword.SelStart = 0
    txtPassword.SelLength = Len(txtPassword.Text)
  Else
    End
  End If
End If
txtPassword.SetFocus
End Sub
```

This code checks the input password to see if it matches the stored value. If so, it prints an acceptance message. If incorrect, it displays a message box to that effect and asks the user if they want to try again. If Yes (Retry), another try is granted. If No (Cancel), the program is ended. Notice the use of **SelLength** and **SelStart** to highlight an incorrect entry. This allows the user to type right over the incorrect response.

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

```
Private Sub Form_Activate()
txtPassword.SetFocus
End Sub
```

6. Attach the following code to the **cmdExit_ Click** event.

```
Private Sub cmdExit_Click()
End
End Sub
```

7. Try running the program. Try both options: input correct password (note it is case sensitive) and input incorrect password. Save your project.

    If you have time, define a constant, TRYMAX = 3, and modify the code to allow the user to have just TRYMAX attempts to get the correct password. After the final try, inform the user you are logging him/her off. You'll also need a variable that counts the number of tries (make it a Static variable).

Check Boxes

- **Check boxes** provide a way to make choices from a list of potential candidates. Some, all, or none of the choices in a group may be selected.

- Check Box Properties:

Caption	Identifying text next to box.
Font	Sets font type, style, size.
Value	Indicates if unchecked (0, vbUnchecked), checked (1, vbChecked), or grayed out (2, vbGrayed).

- Check Box Events:

Click	Triggered when a box is clicked. Value property is automatically changed by Visual Basic.

Option Buttons

- **Option buttons** provide the capability to make a mutually exclusive choice among a group of potential candidate choices. Hence, option buttons work as a group, only one of which can have a True (or selected) value.

- Option Button Properties:

Caption	Identifying text next to button.
Font	Sets font type, style, size.
Value	Indicates if selected (True) or not (False). Only one option button in a group can be True. One button in each group of option buttons should always be initialized to True at design time.

- Option Button Events:

Click	Triggered when a button is clicked. **Value** property is automatically changed by Visua l Basic.

Arrays

- Up to now, we've only worked with regular variables, each having its own unique name. Visual Basic has powerful facilities for handling multi-dimensional variables, or **arrays**. For now, we'll only use single, fixed-dimension arrays.

- Arrays are declared in a manner identical to that used for regular variables. For example, to declare an integer array named **'Items'**, with dimension **9**, at the procedure level, we use:

      Dim Items(9) as Integer

  If we want the array variables to retain their value upon leaving a procedure, we use the keyword **Static**:

      Static Items(9) as Integer

  At the **form** or **module** level, in the general declarations area of the Code window, use:

      Dim Items(9) as Integer

  And, at the module level, for a **global** declaration, use:

      Global Items(9) as Integer

- The index on an array variable begins at 0 and ends at the dimensioned value. For example, the **Items** array in the above examples has **ten** elements, ranging from Items(0) to Items(9). You use array variables just like any other variable - just remember to include its name and its index. For example, to set Item(5) equal to 7, you simply write:

      Item(5) = 7

Control Arrays

* With some controls, it is very useful to define **control arrays** - it depends on the application. For example, option buttons are almost always grouped in control arrays.

* Control arrays are a convenient way to handle groups of controls that perform a similar function. All of the events available to the single control are still available to the array of controls, the only difference being an argument indicating the index of the selected array element is passed to the event. Hence, instead of writing individual procedures for each control (i.e. not using control arrays), you only have to write one procedure for each array.

* Another advantage to control arrays is that you can add or delete array elements at run-time. You cannot do that with controls (objects) not in arrays. Refer to the **Load** and **Unload** statements in on-line help for the proper way to add and delete control array elements at run-time.

* Two ways to **create** a control array:

  1. Create an individual control and set desired properties. Copy the control using the editor, then paste it on the form. Visual Basic will pop-up a dialog box that will ask you if you wish to create a control array. Respond yes and the array is created.

  2. Create all the controls you wish to have in the array. Assign the desired control array name to the first control. Then, try to name the second control with the same name. Visual Basic will prompt you, asking if you want to create a control array. Answer yes. Once the array is created, rename all remaining controls with that name.

* Once a control array has been created and named, elements of the array are referred to by their name and index. For example, to set the **Caption** property of element **6** of a label box array named **lblExample**, we would use:

    lblExample(6).Caption = "This is an example"

  We'll use control arrays in the next example.

Frames

- We've seen that both option buttons and check boxes work as a group. **Frames** provide a way of grouping related controls on a form. And, in the case of option buttons, frames affect how such buttons operate.

- To group controls in a frame, you first draw the frame. Then, the associated controls must be drawn in the frame. This allows you to move the frame and controls together. And, once a control is drawn within a frame, it can be copied and pasted to create a control array within that frame. To do this, first click on the object you want to copy. **Copy** the object. Then, click on the frame. **Paste** the object. You will be asked if you want to create a control array. Answer **Yes**.

-

- Drawing the controls outside the frame and dragging them in, copying them into a frame, or drawing the frame around existing controls will not result in a proper grouping. It is perfectly acceptable to draw frames within other frames.

- As mentioned, frames affect how option buttons work. Option buttons within a frame work as a **group**, independently of option buttons in other frames. Option buttons on the form, and not in frames, work as another independent group. That is, the form is itself a frame by default. We'll see this in the next example.

- It is important to note that an independent group of option buttons is defined by physical location within frames, not according to naming convention. That is, a control array of option buttons does not work as an independent group just because it is a control array. It would only work as a group if it were the only group of option buttons within a frame or on the form. So, remember physical location, and physical location only, dictates independent operation of option button groups.

- Frame Properties:

    Caption   Title information at top of frame.
    Font   Sets font type, style, size.

Example 3-2

Pizza Order

1. Start a new project. We'll build a form where a pizza order can be entered by simply clicking on check boxes and option buttons.

2. Draw three frames. In the first, draw three option buttons, in the second, draw two option buttons, and in the third, draw six check boxes. Draw two option buttons on the form. Add two command buttons. Make things look something like this.
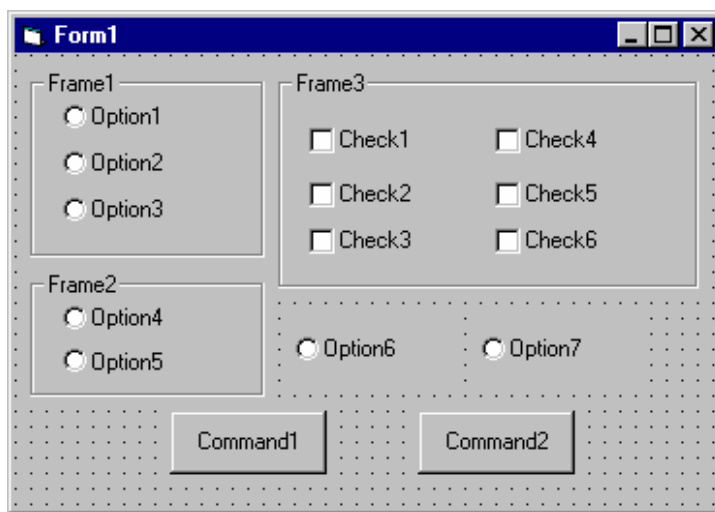
3. Set the properties of the form and each control.

   Form1:
BorderStyle	1-Fixed Single
Caption	Pizza Order
Name	frmPizza

   Frame1:
Caption	Size

   Frame2:
Caption	Crust Type

   Frame3
Caption	Toppings

Option1:
    Caption   Small
    Name   optSize
    Value   True

Option2:
    Caption   Medium
    Name   optSize (yes, create a control array)

Option3:
    Caption   Large
    Name   optSize

Option4:
    Caption   Thin Crust
    Name   optCrust
    Value   True

Option5:
    Caption   Thick Crust
    Name   optCrust (yes, create a control array)

Option6:
    Caption   Eat In
    Name   optWhere
    Value   True

Option7:
    Caption   Take Out
    Name   optWhere (yes, create a control array)

Check1:
    Caption   Extra Cheese
    Name   chkTop

Check2:
    Caption   Mushrooms
    Name   chkTop (yes, create a control array)

Check3:
    Caption   Black Olives
    Name   chkTop

Check4:
    Caption   Onions
    Name   chkTop

Check5:
    Caption   Green Peppers
    Name   chkTop

Check6:
    Caption   Tomatoes
    Name   chkTop

Command1:
    Caption   &Build Pizza
    Name   cmdBuild

Command2:
    Caption   E&xit
    Name   cmdExit

The form should look like this now:
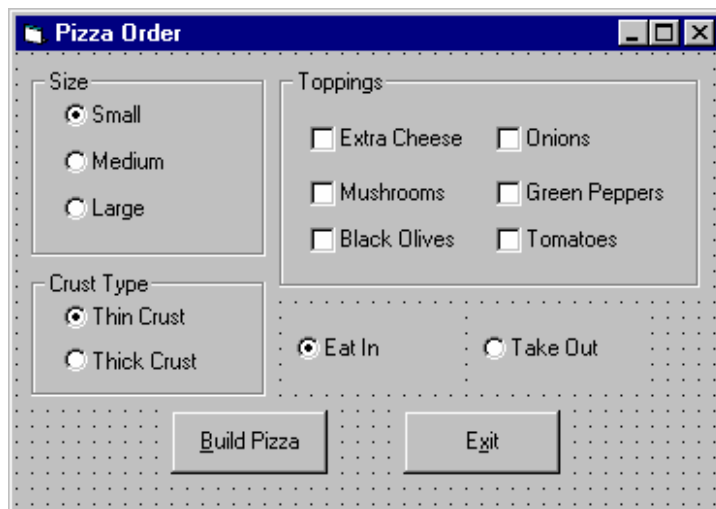
4. Declare the following variables in the **general declarations** area:

```
Option Explicit
Dim PizzaSize As String
Dim PizzaCrust As String
Dim PizzaWhere As String
```

This makes the size, crust, and location variables global to the form.

5. Attach this code to the **Form_Load** procedure. This initializes the pizza size, crust, and eating location.

```
Private Sub Form_Load()
'Initialize pizza parameters
PizzaSize = "Small"
PizzaCrust = "Thin Crust"
PizzaWhere = "Eat In"
End Sub
```

Here, the global variables are initialized to their default values, corresponding to the default option buttons.

6. Attach this code to the three option button array **Click** events. Note the use of the Index variable:

```
Private Sub optSize_Click(Index As Integer)
'Read pizza size
PizzaSize = optSize(Index).Caption
End Sub

Private Sub optCrust_Click(Index As Integer)
'Read crust type
PizzaCrust = optCrust(Index).Caption
End Sub

Private Sub optWhere_Click(Index As Integer)
'Read pizza eating location
PizzaWhere = optWhere(Index).Caption
End Sub
```

In each of these routines, when an option button is clicked, the value of the corresponding button's caption is loaded into the respective variable.

7. Attach this code to the **cmdBuild_Click** event.

```
Private Sub cmdBuild_Click()
'This procedure builds a message box that displays your
  pizza type
Dim Message As String
Dim I As Integer
Message = PizzaWhere + vbCr
Message = Message + PizzaSize + " Pizza" + vbCr
Message = Message + PizzaCrust + vbCr
For I = 0 To 5
  If chkTop(I).Value = vbChecked Then Message = Message +
  chkTop(I).Caption + vbCr
Next I
MsgBox Message, vbOKOnly, "Your Pizza"
End Sub
```

This code forms the first part of a message for a message box by concatenating the pizza size, crust type, and eating location (**vbCr** is a symbolic constant representing a 'carriage return' that puts each piece of ordering information on a separate line). Next, the code cycles through the six topping check boxes and adds any checked information to the message. The code then displays the pizza order in a message box.

8. Attach this code to the **cmdExit_Click** event.

```
Private Sub cmdExit_Click()
End
End Sub
```

9. Get the application working. Notice how the different selection buttons work in their individual groups. Save your project.

10. If you have time, try these modifications:

   A. Add a new program button that resets the order form to the initial default values. You'll have to reinitialize the three global variables, reset all check boxes to unchecked, and reset all three option button groups to their default values.

   B. Modify the code so that if no toppings are selected, the message "Cheese Only" appears on the order form. You'll need to figure out a way to see if no check boxes were checked.

List Boxes

- A **list box** displays a list of items from which the user can select one or more items. If the number of items exceeds the number that can be displayed, a scroll bar is automatically added.

- List Box Properties:

Appearance	Selects 3-D or flat appearance.
List	Array of items in list box.
ListCount	Number of items in list.
ListIndex	The number of the most recently selected item in list. If no item is selected, ListIndex = -1.
MultiSelect	Controls how items may be selected (0-no multiple selection allowed, 1-multiple selection allowed, 2-group selection allowed).
Selected	Array with elements set equal to True or False, depending on whether corresponding list item is selected.
Sorted	True means items are sorted in 'Ascii' order, else items appear in order added.
Text	Text of most recently selected item.

- List Box Events:

Click	Event triggered when item in list is clicked.
DblClick	Event triggered when item in list is double-clicked. Primary way used to process selection.

- List Box Methods:

AddItem	Allows you to insert item in list.
Clear	Removes all items from list box.
RemoveItem	Removes item from list box, as identified by index of item to remove.

Examples

```
lstExample.AddItem "This is an added item" ' adds text string to list
lstExample.Clear ' clears the list box
lstExample.RemoveItem 4 ' removes lstExample.List(4) from list box
```

- Items in a list box are usually initialized in a Form_Load procedure. It's always a good idea to **Clear** a list box before initializing it.

- You've seen list boxes before. In the standard 'Open File' window, the Directory box is a list box with MultiSelect equal to zero.

Combo Boxes

- The **combo box** is similar to the list box. The differences are a combo box includes a text box on top of a list box and only allows selection of one item. In some cases, the user can type in an alternate response.

- Combo Box Properties:

  Combo box properties are nearly identical to those of the list box, with the deletion of the MultiSelect property and the addition of a Style property.

Appearance	Selects 3-D or flat appearance.
List	Array of items in list box portion.
ListCount	Number of items in list.
ListIndex	The number of the most recently selected item in list. If no item is selected, ListIndex = -1.
Sorted	True means items are sorted in 'Ascii' order, else items appear in order added.
Style	Selects the combo box form.
	Style = 0, Dropdown combo; user can change selection.
	Style = 1, Simple combo; user can change selection.
	Style = 2, Dropdown combo; user cannot change selection.
Text	Text of most recently selected item.

- Combo Box Events:

Click	Event triggered when item in list is clicked.
DblClick	Event triggered when item in list is double-clicked. Primary way used to process selection.

* Combo Box Methods:

AddItem	Allows you to insert item in list.
Clear	Removes all items from list box.
RemoveItem	Removes item from list box, as identified by index of item to remove.

  Examples

  cboExample.AddItem "This is an added item" ' adds text string to list
  cboExample.Clear ' clears the combo box
  cboExample.RemoveItem 4 ' removes cboExample.List(4) from list box

* You've seen combo boxes before. In the standard 'Open File' window, the File Name box is a combo box of Style 2, while the Drive box is a combo box of Style 3.

Example 3-3

Flight Planner

1. Start a new project. In this example, you select a destination city, a seat location, and a meal preference for airline passengers.

2. Place a list box, two combo boxes, three label boxes and two command buttons on the form. The form should appear similar to this:
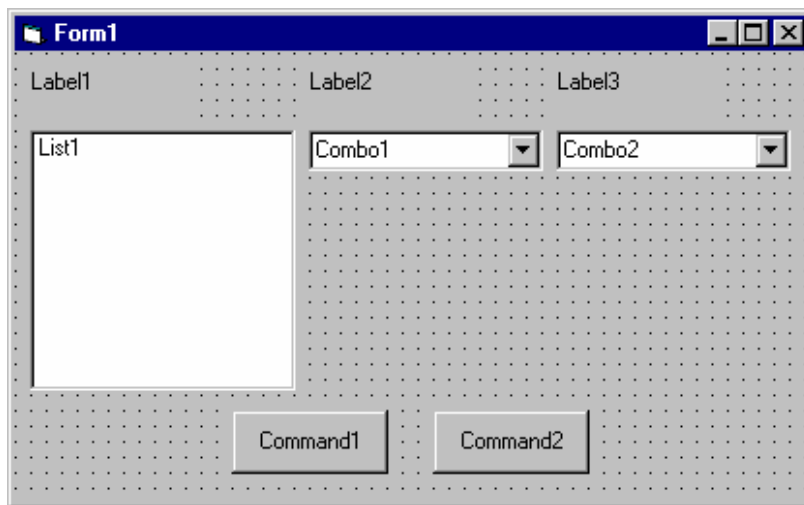
3. Set the form and object properties:

    Form1:

BorderStyle	1-Fixed Single
Caption	Flight Planner
Name	frmFlight

    List1:

Name	lstCities
Sorted	True

    Combo1:

Name	cboSeat
Style	2-Dropdown List

Combo2:
    Name   cboMeal
    Style   1-Simple
    Text   [Blank]

(After setting properties for this combo box, resize it until it is large enough to hold 4 to 5 entries.)

Label1:
    Caption   Destination City

Label2:
    Caption   Seat Location

Label3:
    Caption   Meal Preference

Command1:
   Caption   &Assign
    Name   cmdAssign

Command2:
    Caption   E&xit
    Name   cmdExit

Now, the form should look like this:
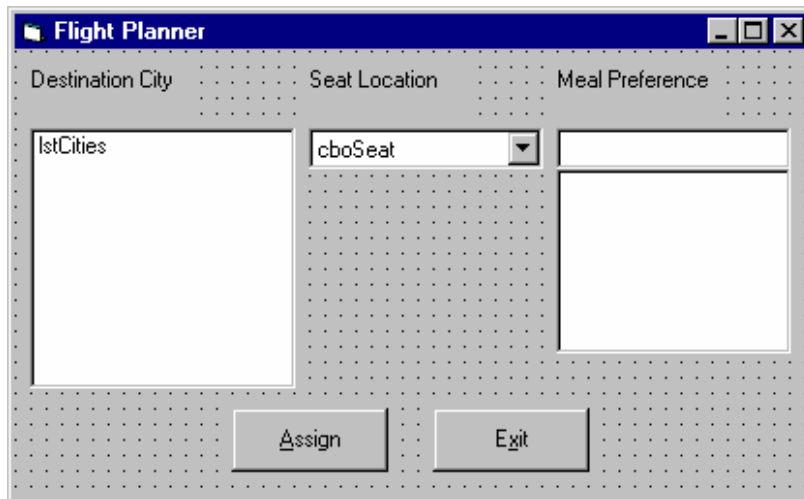
4. Attach this code to the **Form_Load** procedure:

```
Private Sub Form_Load()
'Add city names to list box
lstCities.Clear
lstCities.AddItem "San Diego"
lstCities.AddItem "Los Angeles"
lstCities.AddItem "Orange County"
lstCities.AddItem "Ontario"
lstCities.AddItem "Bakersfield"
lstCities.AddItem "Oakland"
lstCities.AddItem "Sacramento"
lstCities.AddItem "San Jose"
lstCities.AddItem "San Francisco"
lstCities.AddItem "Eureka"
lstCities.AddItem "Eugene"
lstCities.AddItem "Portland"
lstCities.AddItem "Spokane"
lstCities.AddItem "Seattle"
lstCities.ListIndex = 0

'Add seat types to first combo box
cboSeat.AddItem "Aisle"
cboSeat.AddItem "Middle"
cboSeat.AddItem "Window"
cboSeat.ListIndex = 0

'Add meal types to second combo box
cboMeal.AddItem "Chicken"
cboMeal.AddItem "Mystery Meat"
cboMeal.AddItem "Kosher"
cboMeal.AddItem "Vegetarian"
cboMeal.AddItem "Fruit Plate"
cboMeal.Text = "No Preference"
End Sub
```

This code simply initializes the list box and the list box portions of the two combo boxes.

5. Attach this code to the **cmdAssign_Click** event:

```
Private Sub cmdAssign_Click()
'Build message box that gives your assignment
Dim Message As String
Message = "Destination: " + lstCities.Text + vbCr
Message = Message + "Seat Location: " + cboSeat.Text + vbCr
Message = Message + "Meal: " + cboMeal.Text + vbCr
MsgBox Message, vbOKOnly + vbInformation, "Your Assignment"
End Sub
```

When the **Assign** button is clicked, this code forms a message box message by concatenating the selected city (from the list box **lstCities**), seat choice (from **cboSeat**), and the meal preference (from **cboMeal**).

6. Attach this code to the **cmdExit_Click** event:

```
Private Sub cmdExit_Click()
End
End Sub
```

7. Run the application. Save the project.

Exercise 3

Customer Database Input Screen

A new sports store wants you to develop an input screen for its customer database. The required input information is:

1. Name
2. Age
3. City of Residence
4. Sex (Male or Female)
5. Activities (Running, Walking, Biking, Swimming, Skiing and/or In-Line Skating)
6. Athletic Level (Extreme, Advanced, Intermediate, or Beginner)

Set up the screen so that only the Name and Age (use text boxes) and, perhaps, City (use a combo box) need to be typed; all other inputs should be set with check boxes and option buttons. When a screen of information is complete, display the summarized profile in a message box. This profile message box should resemble this:

My Solution:

Form:
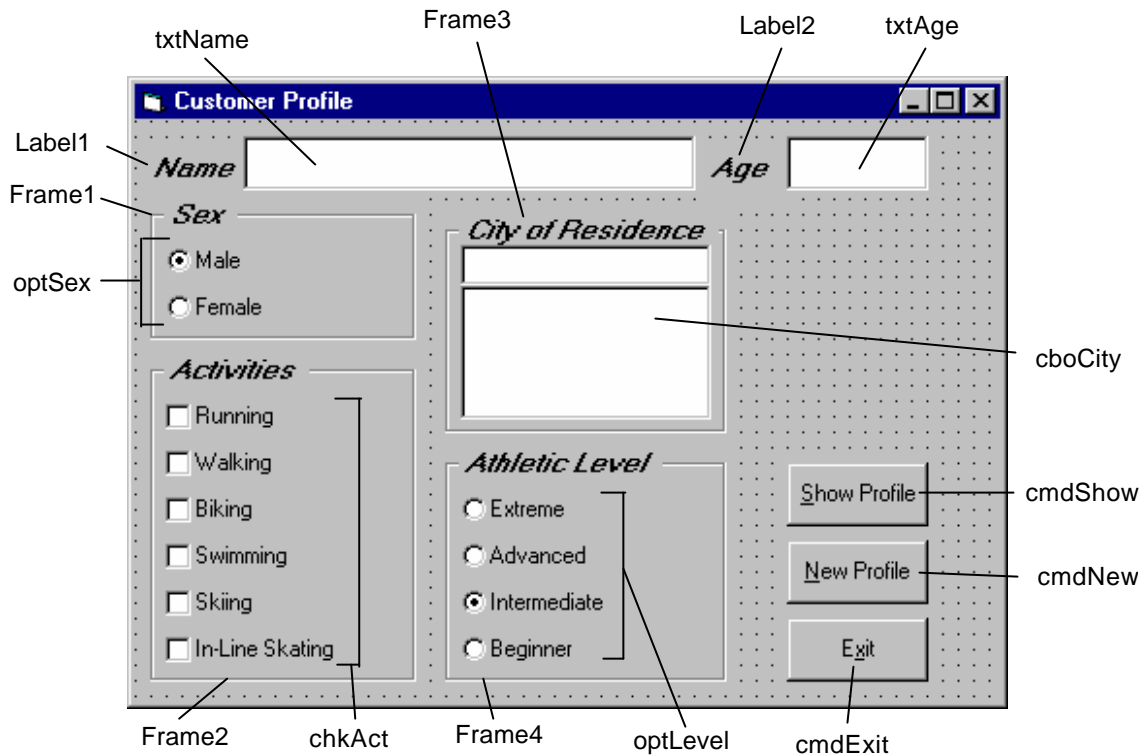
Properties:

Form **frmCustomer**:
    BorderStyle = 1 - Fixed Single
    Caption = Customer Profile

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

Frame **Frame3**:
    Caption = City of Residence
    FontName = MS Sans Serif
    FontBold = True
    FontSize = 9.75
    FontItalic = True

ComboBox **cboCity**:
    Sorted = True
    Style = 1 - Simple Combo

CommandButton **cmdNew**:
  Caption = &New Profile

CommandButton **cmdShow**:
  Caption = &Show Profile

Frame **Frame4**:
  Caption = Athletic Level
  FontName = MS Sans Serif
  FontBold = True
  FontSize = 9.75
  FontItalic = True

OptionButton **optLevel**:
  Caption = Beginner
  Index = 3

OptionButton **optLevel**:
  Caption = Intermediate
  Index = 2
  Value = True

OptionButton **optLevel**:
  Caption = Advanced
  Index = 1

OptionButton **optLevel**:
  Caption = Extreme
  Index = 0

Frame **Frame1**:
  Caption = Sex
  FontName = MS Sans Serif
  FontBold = True
  FontSize = 9.75
  FontItalic = True

OptionButton **optSex**:
  Caption =   Female
  Index = 1

OptionButton **optSex**:
  Caption = Male
  Index = 0
  Value = True

Frame **Frame2**:
      Caption = Activities
      FontName = MS Sans Serif
      FontBold = True
      FontSize = 9.75
      FontItalic = True

CheckBox **chkAct**:
      Caption = In-Line Skating
      Index = 5

CheckBox **chkAct**:
      Caption = Skiing
      Index = 4

CheckBox **chkAct**:
      Caption = Swimming
      Index = 3

CheckBox **chkAct**:
      Caption = Biking
      Index = 2

CheckBox **chkAct**:
      Caption = Waking
      Index = 1

CheckBox **chkAct**:
      Caption = Running
      Index = 0

TextBox **txtName**:
      FontName = MS Sans Serif
      FontSize = 12

Label **Label1**:
      Caption = Name
      FontName = MS Sans Serif
      FontBold = True
      FontSize = 9.75
      FontItalic = True

TextBox **txtAge**:
      FontName = MS Sans Serif
      FontSize = 12

Label **Label2** :
     Caption = Age
     FontName = MS Sans Serif
     FontBold = True
     FontSize = 9.75
     FontItalic = True

Code:

General Declarations:

```
Option Explicit
Dim Activity As String
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
End
End Sub
```

cmdNew Click Event:

```
Private Sub cmdNew_Click()
'Blank out name and reset check boxes
Dim I As Integer
txtName.Text = ""
txtAge.Text = ""
For I = 0 To 5
  chkAct(I).Value = vbUnchecked
Next I
End Sub
```

cmdShow Click Event:

```
Private Sub cmdShow_Click()
Dim NoAct As Integer, I As Integer
Dim Msg As String, Pronoun As String

'Check to make sure name entered
If txtName.Text = "" Then
  MsgBox "The profile requires a name.", vbOKOnly +
vbCritical, "No Name Entered"
  Exit Sub
```

```
End If

'Check to make sure age entered
If txtAge.Text = "" Then
  MsgBox "The profile requires an age.", vbOKOnly +
vbCritical, "No Age Entered"
  Exit Sub
End If

'Put together customer profile message
Msg = txtName.Text + " is" + Str$(txtAge.Text) + " years
old." + vbCr
If optSex(0).Value = True Then Pronoun = "He " Else Pronoun
= "She "
Msg = Msg + Pronoun + "lives in " + cboCity.Text + "." +
vbCr
Msg = Msg + Pronoun + "is a"
If optLevel(3).Value = False Then Msg = Msg + "n " Else Msg
= Msg + " "
Msg = Msg + Activity + " level athlete." + vbCr
NoAct = 0
For I = 0 To 5
  If chkAct(I).Value = vbChecked Then NoAct = NoAct + 1
Next I
If NoAct > 0 Then
  Msg = Msg + "Activities include:" + vbCr
  For I = 0 To 5
    If chkAct(I).Value = vbChecked Then Msg = Msg +
String$(10, 32) + chkAct(I).Caption + vbCr
  Next I
Else
  Msg = Msg + vbCr
End If
MsgBox Msg, vbOKOnly, "Customer Profile"
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
'Load combo box with potential city names
cboCity.AddItem "Seattle"
cboCity.Text = "Seattle"
cboCity.AddItem "Bellevue"
cboCity.AddItem "Kirkland"
cboCity.AddItem "Everett"
cboCity.AddItem "Mercer Island"
```

```
cboCity.AddItem "Renton"
cboCity.AddItem "Issaquah"
cboCity.AddItem "Kent"
cboCity.AddItem "Bothell"
cboCity.AddItem "Tukwila"
cboCity.AddItem "West Seattle"
cboCity.AddItem "Edmonds"
cboCity.AddItem "Tacoma"
cboCity.AddItem "Federal Way"
cboCity.AddItem "Burien"
cboCity.AddItem "SeaTac"
cboCity.AddItem "Woodinville"
Activity = "intermediate"
End Sub
```

optLevel Click Event:

```
Private Sub optLevel_Click(Index As Integer)
'Determine activity level
Select Case Index
Case 0
   Activity = "extreme"
Case 1
  Activity = "advanced"
Case 2
  Activity = "intermediate"
Case 3
  Activity = "beginner"
End Select
End Sub
```

txtAge KeyPress Event:

```
Private Sub txtAge_KeyPress(KeyAscii As Integer)
'Only allow numbers for age
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii
= vbKeyBack Then
  Exit Sub
Else
  KeyAscii = 0
End If
End Sub
```

This page intentionally not left blank.

Learn Visual Basic 6.0

4. More Exploration of the Visual Basic Toolbox

Review and Preview

- In this class, we continue looking at tools in the Visual Basic toolbox. We will look at some drawing tools, scroll bars, and tools that allow direct interaction with drives, directories, and files. In the examples, try to do as much of the building and programming of the applications you can with minimal reference to the notes. This will help you build your programming skills.

Display Layers

- In this class, we will look at our first graphic type controls: line tools, shape tools, picture boxes, and image boxes. And, with this introduction, we need to discuss the idea of display **layers**.

- Items shown on a form are not necessarily all on the same layer of display. A form's display is actually made up of three layers as sketched below. All information

displayed directly on the form (by printing or drawing with graphics methods) appears on the **bottom-layer**. Information from label boxes, image boxes, line tools, and shape tools, appears on the **middle-layer**. And, all other objects are displayed on the **top-layer**.

Bottom-layer: form

Middle-layer: label, image, shape, line

Top-layer: other controls and objects
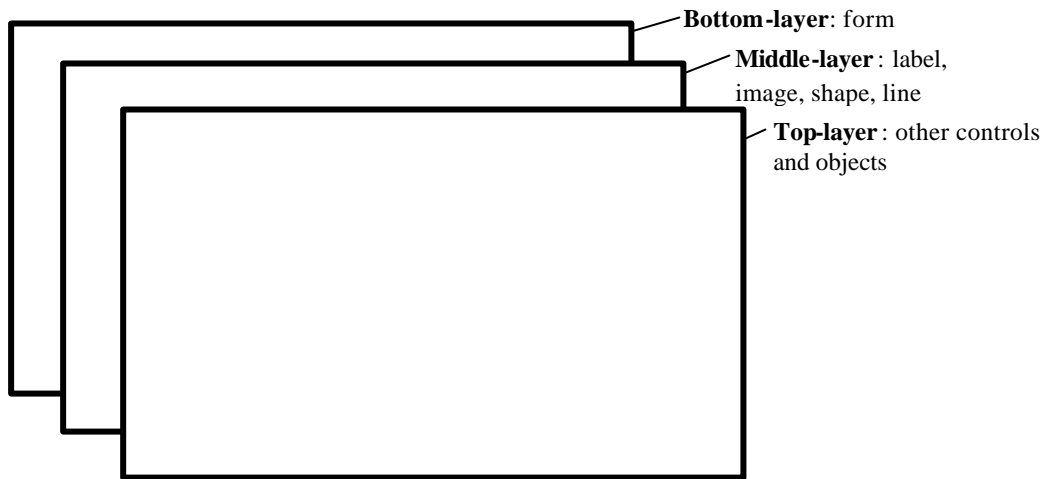
- What this means is you have to be careful where you put things on a form or something could be covered up. For example, text printed on the form would be hidden by a command button placed on top of it. Things drawn with the shape tool are covered by all controls except the image box.

- The next question then is what establishes the relative location of objects in the same layer. That is, say two command buttons are in the same area of a form - which one lies on top of which one? The order in which objects in the same layer overlay each other is called the **Z-order**. This order is first established when you draw the form. Items drawn last lie over items drawn earlier. Once drawn, however, the Z-order can be modified by clicking on the desired object and choosing **Bring to Front** from Visual Basic's **Edit** menu. The **Send to Back** command has the opposite effect. Note these two commands only work within a layer; middle-layer objects will always appear behind top-layer objects and lower layer objects will always appear behind middle-layer objects.

Line Tool

- The **line tool** creates simple straight line segments of various width and color. Together with the shape tool discussed next, you can use this tool to 'dress up' your application.

- Line Tool Properties:

BorderColor	Determines the line color.
BorderStyle	Determines the line 'shape'. Lines can be transparent, solid, dashed, dotted, and combinations.
BorderWidth	Determines line width.

- There are no events or methods associated with the line tool.

- Since the line tool lies in the middle-layer of the form display, any lines drawn will be obscured by all controls except the shape tool or image box.

Shape Tool

- The **shape tool** can create circles, ovals, squares, rectangles, and rounded squares and rectangles. Colors can be used and various fill patterns are available.

- Shape Tool Properties:

|	
BackColor	Determines the background color of the shape (only used when FillStyle not Solid.
BackStyle	Determines whether the background is transparent or opaque.
BorderColor	Determines the color of the shape's outline.
BorderStyle	Determines the style of the shape's outline. The border can be transparent, solid, dashed, dotted, and combinations.
BorderWidth	Determines the width of the shape border line.
FillColor	Defines the interior color of the shape.
FillStyle	Determines the interior pattern of a shape. Some choices are: solid, transparent, cross, etc.
Shape	Determines whether the shape is a square, rectangle, circle, or some other choice.

- Like the line tool, events and methods are not used with the shape tool.

- Shapes are covered by all objects except perhaps line tools and image boxes (depends on their Z-order) and printed or drawn information. This is a good feature in that you usually use shapes to contain a group of control objects and you'd want them to lie on top of the shape.

Horizontal and Vertical Scroll Bars

* Horizontal and vertical **scroll bars** are widely used in Windows applications. Scroll bars provide an intuitive way to move through a list of information and make great input devices.

* Both type of scroll bars are comprised of three areas that can be clicked, or dragged, to change the scroll bar value. Those areas are:
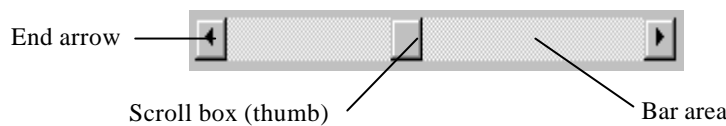
Clicking an **end arrow** increments the **scroll box** a small amount, clicking the **bar area** increments the scroll box a large amount, and dragging the scroll box (thumb) provides continuous motion. Using the properties of scroll bars, we can completely specify how one works. The scroll box position is the only output information from a scroll bar.

* Scroll Bar Properties:

|	
LargeChange	Increment added to or subtracted from the scroll bar **Value** property when the bar area is clicked.
Max	The value of the horizontal scroll bar at the far right and the value of the vertical scroll bar at the bottom. Can range from -32,768 to 32,767.
Min	The other extreme value - the horizontal scroll bar at the left and the vertical scroll bar at the top. Can range from -32,768 to 32,767.
SmallChange	The increment added to or subtracted from the scroll bar **Value** property when either of the scroll arrows is clicked.
Value	The current position of the scroll box (thumb) within the scroll bar. If you set this in code, Visual Basic moves the scroll box to the proper position.

Properties for horizontal scroll bar:
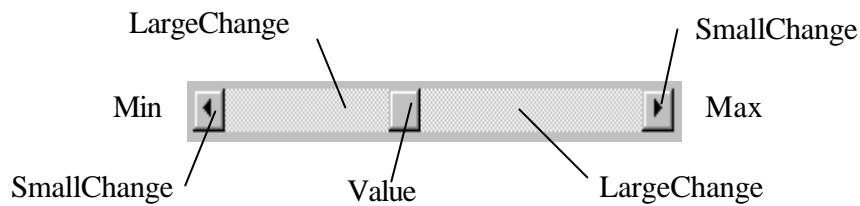
Properties for vertical scroll bar:
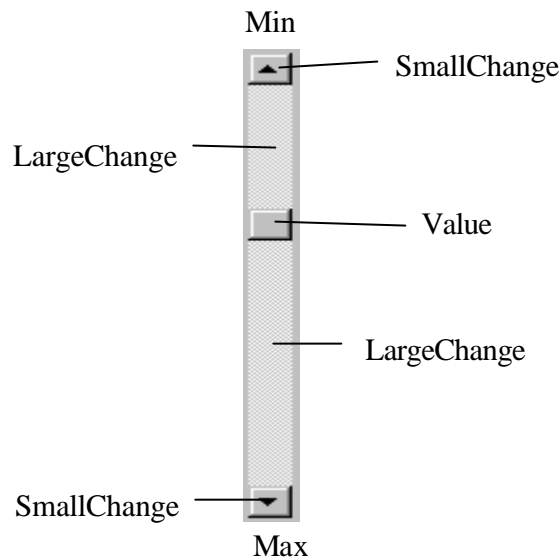
- A couple of important notes about scroll bars:

    1. Note that although the extreme values are called **Min** and **Max**, they do not necessarily represent minimum and maximum values. There is nothing to keep the Min value from being greater than the Max value. In fact, with vertical scroll bars, this is the usual case. Visual Basic automatically adjusts the sign on the **SmallChange** and **LargeChange** properties to insure proper movement of the scroll box from one extreme to the other.

    2. If you ever change the **Value**, **Min**, or **Max** properties in code, make sure Value is at all times between Min and Max or and the program will stop with an error message.

- Scroll Bar Events:

Change	Event is triggered after the scroll box's position has been modified. Use this event to retrieve the Value property after any changes in the scroll bar.
Scroll	Event triggered continuously whenever the scroll box is being moved.

Example 4-1

Temperature Conversion

Start a new project. In this project, we convert temperatures in degrees Fahrenheit (set using a scroll bar) to degrees Celsius. As mentioned in the **Review and Preview** section, you should try to build this application with minimal reference to the notes. To that end, let's look at the project specifications.

**Temperature Conversion Application Specifications**

The application should have a scroll bar which adjusts temperature in degrees Fahrenheit from some reasonable minimum to some maximum. As the user changes the scroll bar value, both the Fahrenheit temperature and Celsius temperature (you have to calculate this) in integer format should be displayed. The formula for converting Fahrenheit (F) to Celsius (C) is:

$$C = (F - 32)*5/9$$

To convert this number to a rounded integer, use the Visual Basic **CInt**() function. To change numeric information to strings for display in label or text boxes, use the **Str**() or **Format**() function. Try to build as much of the application as possible before looking at my approach. Try incorporating lines and shapes into your application if you can.

One Possible Approach to Temperature Conversion Application:

1. Place a shape, a vertical scroll bar, four labels, and a command button on the form. Put the scroll bar within the shape - since it is in the top-layer of the form, it will lie in the shape. It should resemble this:
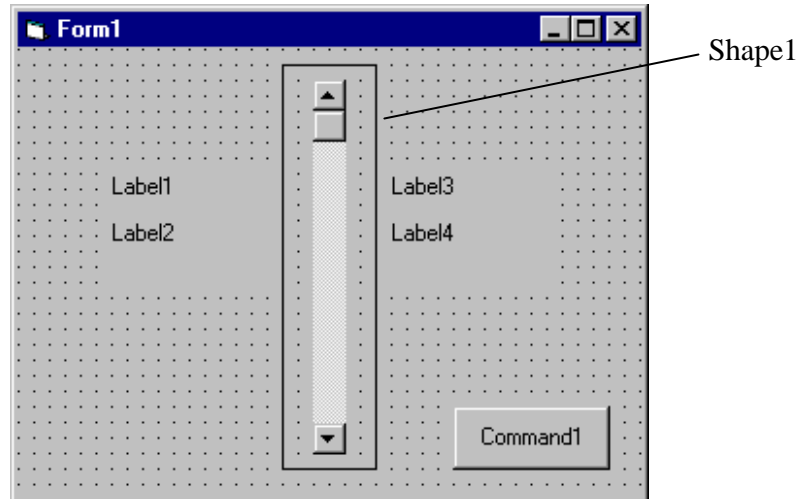
2. Set the properties of the form and each object:

> **Form1** :
>> BorderStyle   1-Fixed Single
>> Caption   Temperature Conversion
>> Name   frmTemp
>
> **Shape1**:
>> BackColor   White
>> BackStyle   1-Opaque
>> FillColor   Red
>> FillStyle   7-Diagonal Cross
>> Shape   4-Rounded Rectangle
>
> **VScroll1:**
>> LargeChange   10
>> Max   -60
>> Min   120
>> Name   vsbTemp
>> SmallChange   1
>> Value   32

Label1:

Alignment	2-Center
Caption	Fahrenheit
FontSize	10
FontStyle	Bold

Label2:

Alignment	2-Center
AutoSize	True
BackColor	White
BorderStyle	1-Fixed Single
Caption	32
FontSize	14
FontStyle	Bold
Name	lblTempF

Label3:

Alignment	2-Center
Caption	Celsius
FontSize	10
FontStyle	Bold

Label4:

Alignment	2-Center
AutoSize	True
BackColor	White
BorderStyle	1-Fixed Single
Caption	0
FontSize	14
FontStyle	Bold
Name	lblTempC

Command1:

Cancel	True
Caption	E&xit
Name	cmdExit

Note the temperatures are initialized at 32F and 0C, known values.

When done, the form should look like this:
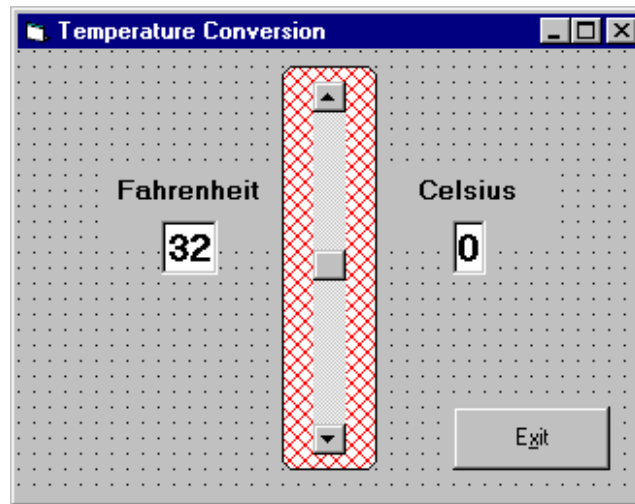
3. Put this code in the general declarations of your code window.

```
Option Explicit
Dim TempF As Integer
Dim TempC As Integer
```

This makes the two temperature variables global.

4. Attach the following code to the scroll bar **Scroll** event.

```
Private Sub vsbTemp_Scroll()
'Read F and convert to C
TempF = vsbTemp.Value
lblTempF.Caption = Str(TempF)
TempC = CInt((TempF - 32) * 5 / 9)
lblTempC.Caption = Str(TempC)
End Sub
```

This code determines the scroll bar Value as it scrolls, takes that value as Fahrenheit temperature, computes Celsius temperature, and displays both values.

5. Attach the following code to the scroll bar **Change** event.

```
Private Sub vsbTemp_Change()
'Read F and convert to C
TempF = vsbTemp.Value
lblTempF.Caption = Str(TempF)
TempC = CInt((TempF - 32) * 5 / 9)
lblTempC.Caption = Str(TempC)
End Sub
```

Note this code is identical to that used in the Scroll event. This is almost always the case when using scroll bars.

6. Attach the following code to the **cmdExit_Click** procedure.

```
Private Sub cmdExit_Click()
End
End Sub
```

7. Give the program a try. Make sure it provides correct information at obvious points. For example, 32 F better always be the same as 0 C! Save the project - we'll return to it briefly in Class 5.

Other things to try:

A. Can you find a point where Fahrenheit temperature equals Celsius temperature? If you don't know this off the top of your head, it's obvious you've never lived in extremely cold climates. I've actually witnessed one of those bank temperature signs flashing degrees F and degrees C and seeing the same number!

B. Ever wonder why body temperature is that odd figure of 98.6 degrees F? Can your new application give you some insight to an answer to this question?

C. It might be interesting to determine how wind affects perceived temperature - the wind chill. Add a second scroll bar to input wind speed and display both the actual and wind adjusted temperatures. You would have to do some research to find the mathematics behind wind chill computations. This is not a trivial extension of the application.

Picture Boxes

- The **picture box** allows you to place graphics information on a form. It is best suited for dynamic environments - for example, when doing animation.

- Picture boxes lie in the top layer of the form display. They behave very much like small forms within a form, possessing most of the same properties as a form.

- Picture Box Properties:

AutoSize	If True, box adjusts its size to fit the displayed graphic.
Font	Sets the font size, style, and size of any printing done in the picture box.
Picture	Establishes the graphics file to display in the picture box.

- Picture Box Events:

Click	Triggered when a picture box is clicked.
DblClick	Triggered when a picture box is double-clicked.

- Picture Box Methods:

Cls	Clears the picture box.
Print	Prints information to the picture box.

Examples

```
picExample.Cls ' clears the box picExample
picExample.Print "a picture box" ' prints text string to picture box
```

- Picture Box LoadPicture Procedure:

  An important function when using picture boxes is the **LoadPicture** procedure. It is used to set the **Picture** property of a picture box at run-time.

  Example

     picExample.Picture = LoadPicture("c:\pix\sample.bmp")

  This command loads the graphics file c:\pix\sample.bmp into the Picture property of the picExample picture box. The argument in the LoadPicture function must be a legal, complete path and file name, else your program will stop with an error message.

- Five types of graphics files can be loaded into a picture box:

Bitmap	An image represented by pixels and stored as a collection of bits in which each bit corresponds to one pixel. Usually has a **.bmp** extension. Appears in original size.
Icon	A special type of bitmap file of maximum 32 x 32 size. Has a **.ico** extension. We'll create icon files in Class 5. Appears in original size.
Metafile	A file that stores an image as a collection of graphical objects (lines, circles, polygons) rather than pixels. Metafiles preserve an image more accurately than bitmaps when resized. Has a **.wmf** extension. Resizes itself to fit the picture box area.
JPEG	JPEG (Joint Photographic Experts Group) is a compressed bitmap format which supports 8 and 24 bit color. It is popular on the Internet. Has a .**jpg** extension and scales nicely.
GIF	GIF (Graphic Interchange Format) is a compressed bitmap format originally developed by CompuServe. It supports up to 256 colors and is popular on the Internet. Has a **.gif** extension and scales nicely.

Image Boxes

• An **image box** is very similar to a picture box in that it allows you to place graphics information on a form. Image boxes are more suited for static situations - that is, cases where no modifications will be done to the displayed graphics.

• Image boxes appear in the middle-layer of form display, hence they could be obscured by picture boxes and other objects. Image box graphics can be resized by using the **Stretch** property.

• Image Box Properties:

Picture	Establishes the graphics file to display in the image box.
Stretch	If False, the image box resizes itself to fit the graphic. If True, the graphic resizes to fit the control area.

• Image Box Events:

Click	Triggered when a image box is clicked.
DblClick	Triggered when a image box is double-clicked.

• The image box does not support any methods, however it does use the **LoadPicture** function. It is used in exactly the same manner as the picture box uses it. And image boxes can load the same file types: bitmap (.bmp), icon (.ico), metafiles (.wmf), GIF files (.gif), and JPEG files (.jpg). With Stretch = True, all three graphic types will expand to fit the image box area.

Quick Example: Picture and Image Boxes

1. Start a new project. Draw one picture box and one image box.

2. Set the **Picture** property of the picture and image box to the same file. If you have graphics files installed with Visual Basic, bitmap files can be found in the bitmaps folder, icon files in the icons folder, and metafiles are in the metafile folder.

3. Notice what happens as you resize the two boxes. Notice the layer effect when you move one box on top of the other. Notice the effect of the image box **Stretch** property. Play around with different file types - what differences do you see?

Drive List Box

- The **drive list box** control allows a user to select a valid disk drive at run-time. It displays the available drives in a drop-down combo box. No code is needed to load a drive list box; Visual Basic does this for us. We use the box to get the current drive identification.

- Drive List Box Properties:

    Drive   Contains the name of the currently selected drive.

- Drive List Box Events:

    Change   Triggered whenever the user or program changes the
                           drive selection.

Directory List Box

- The **directory list box** displays an ordered, hierarchical list of the user's disk directories and subdirectories. The directory structure is displayed in a list box. Like, the drive list box, little coding is needed to use the directory list box - Visual Basic does most of the work for us.

- Directory List Box Properties:

    Path   Contains the current directory path.

- Directory List Box Events:

    Change   Triggered when the directory selection is changed.

File List Box

- The **file list box** locates and lists files in the directory specified by its Path property at run-time. You may select the types of files you want to display in the file list box.

- File List Box Properties:

FileName	Contains the currently selected file name.
Path	Contains the current path directory.
Pattern	Contains a string that determines which files will be displayed. It supports the use of * and ? wildcard characters. For example, using *.dat only displays files with the .dat extension.

- File List Box Events:

DblClick	Triggered whenever a file name is double-clicked.
PathChange	Triggered whenever the path changes in a file list box.

- You can also use the **MultiSelect** property of the file list box to allow multiple file selection.

Synchronizing the Drive, Directory, and File List Boxes

- The drive, directory, and file list boxes are almost always used together to obtain a file name. As such, it is important that their operation be synchronized to insure the displayed information is always consistent.

- When the drive selection is changed (drive box **Change** event), you should update the directory path. For example, if the drive box is named drvExample and the directory box is dirExample, use the code:

dirExample.Path = drvExample.Drive

- When the directory selection is changed (directory box **Change** event), you should update the displayed file names. With a file box named filExample, this code is:

filExample.Path = dirExample.Path

- Once all of the selections have been made and you want the file name, you need to form a text string that correctly and completely specifies the file identifier. This string concatenates the drive, directory, and file name information. This should be an easy task, except for one problem. The problem involves the backslash (\) character. If you are at the root directory of your drive, the path name ends with a backslash. If you are not at the root directory, there is no backslash at the end of the path name and you have to add one before tacking on the file name.

- Example code for concatenating the available information into a proper file name and then loading it into an image box is:

```
Dim YourFile as String

If Right(filExample.Path,1) = "\" Then
  YourFile = filExample.Path + filExample.FileName
Else
  YourFile = filExample.Path + "\" + filExample.FileName
End If
imgExample.Picture = LoadPicture(YourFile)
```

Note we only use properties of the file list box. The drive and directory box properties are only used to create changes in the file list box via code.

Example 4-2

Image Viewer

Start a new project. In this application, we search our computer's file structure for graphics files and display the results of our search in an image box.

Image Viewer Application Specifications

Develop an application where the user can search and find graphics files (*.ico, *.bmp, *.wmf) on his/her computer. Once a file is selected, print the corresponding file name on the form and display the graphic file in an image box using the **LoadPicture**() function.

One possible solution to the Image Viewer Application:

1. Place a drive list box, directory list box, file list box, four label boxes, a line (use the line tool) and a command button on the form. We also want to add an image box, but make it look like it's in some kind of frame. Build this display area in these steps: draw a 'large shape', draw another shape within this first shape that is the size of the image display area, and lastly, draw an image box right on top of this last shape. Since the two shapes and image box are in the same display layer, the image box is on top of the second shape which is on top of the first shape, providing the desired effect of a kind of picture frame. The form should look like this:
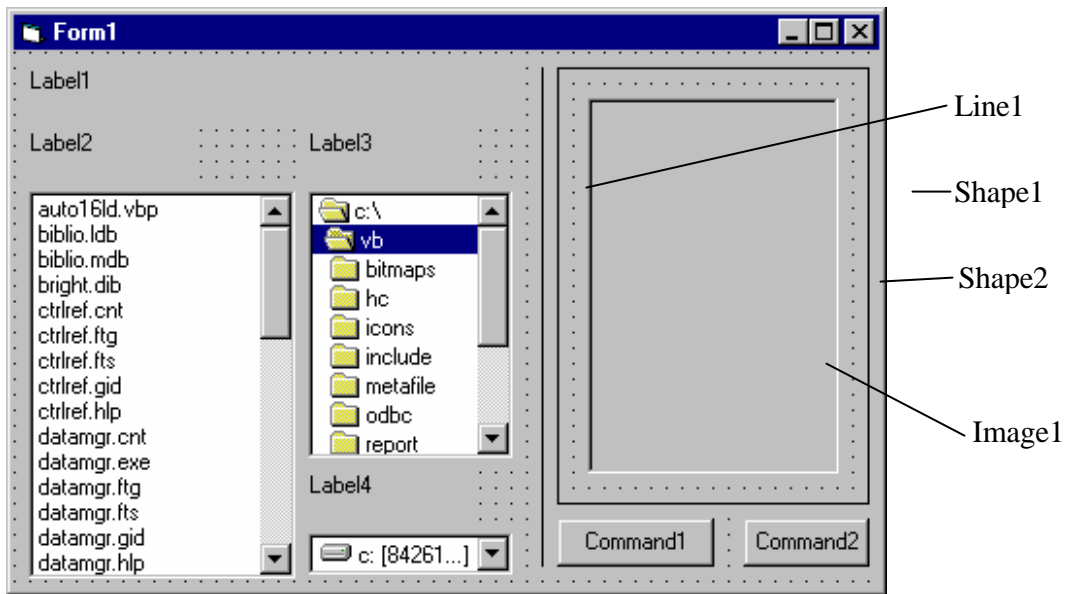
Note the second shape is directly beneath the image box.

2. Set properties of the form and each object.

> **Form1:**
> | | |
> |---|---|
> | BorderStyle | 1-Fixed Single |
> | Caption | Image Viewer |
> | Name | frmImage |
>
> **Drive1:**
> | | |
> |---|---|
> | Name | drvImage |
>
> **Dir1:**
> | | |
> |---|---|
> | Name | dirImage |

File1:

Name	filImage
Pattern	*.bmp;*.ico;*.wmf;* gif;*jpg
	[type this line with <u>no</u> spaces]

Label1:

Caption	[Blank]
BackColor	Yellow
BorderStyle	1-Fixed Single
Name	lblImage

Label2:

Caption	Files:

Label3:

Caption	Directories:

Label4:

Caption	Drives:

Command1:

Caption	&Show Image
Default	True
Name	cmdShow

Command2:

Cancel	True
Caption	E&xit
Name	cmdExit

Line1:

BorderWidth	3

Shape1:

BackColor	Cyan
BackStyle	1-Opaque
FillColor	Blue
FillStyle	4-Upward Diagonal
Shape	4-Rounded Rectangle

Shape2:

BackColor	White
BackStyle	1-Opaque

Image1:
BorderStyle	1-Fixed Single
Name	imgImage
Stretch	True

3. Attach the following code to the **drvImage_Change** procedure.

```
Private Sub drvImage_Change()
'If drive changes, update directory
dirImage.Path = drvImage.Drive
End Sub
```

When a new drive is selected, this code forces the directory list box to display directories on that drive.

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

```
Private Sub dirImage_Change()
'If directory changes, update file path
filImage.Path = dirImage.Path
End Sub
```

Likewise, when a new directory is chosen, we want to see the files on that directory.

5. Attach this code to the **cmdShow_Click** event.

```
Private Sub cmdShow_Click()
'Put image file name together and
'load image into image box
Dim ImageName As String
'Check to see if at root directory
If Right(filImage.Path, 1) = "\" Then
  ImageName = filImage.Path + filImage.filename
Else
  ImageName = filImage.Path + "\" + filImage.filename
End If
lblImage.Caption = ImageName
imgImage.Picture = LoadPicture(ImageName)
End Sub
```

This code forms the file name (**ImageName**) by concatenating the directory path with the file name. It then displays the complete name and loads the picture into the image box.

6. Copy the code from the **cmdShow_Click** procedure and paste it into the
    fillmage_DblClick procedure. The code is identical because we want to display the
    image either by double-clicking on the filename or clicking the command button once
    a file is selected. Those of you who know how to call routines in Visual Basic should
    note that this duplication of code is unnecessary - we could simply have the
    fillmage_DblClick procedure call the **cmdShow_Click** procedure. We'll learn more
    about this next class.

7. Attach this code to the **cmdExit_Click** procedure.

```
Private Sub cmdExit_Click()
End
End Sub
```

8. Save your project. Run and try the application. Find bitmaps, icons, and metafiles.
    Notice how the image box Stretch property affects the different graphics file types.
    Here's how the form should look when displaying one example metafile:
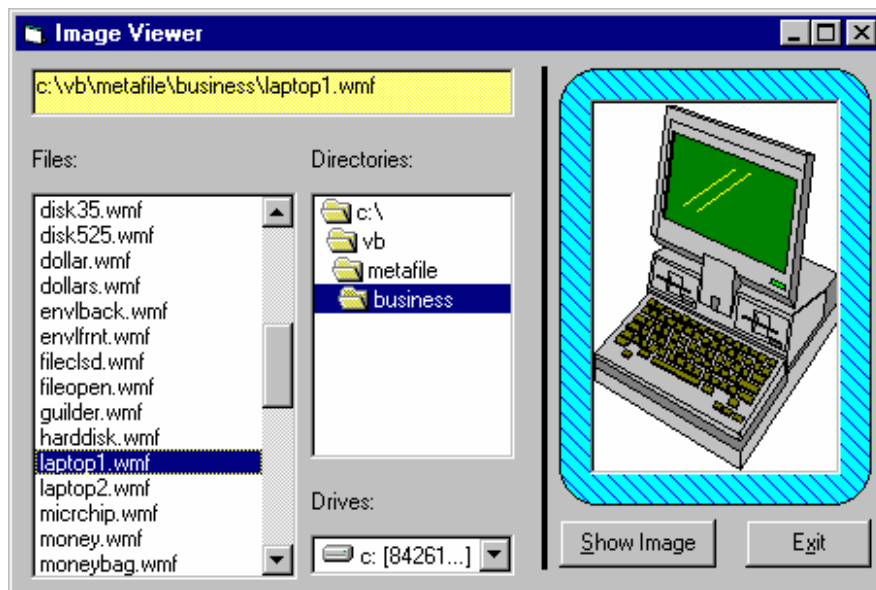
Common Dialog Boxes

- The primary use for the drive, directory, and file name list boxes is to develop custom file access routines. Two common file access routines in Windows-based applications are the **Open File** and **Save File** operations. Fortunately, you don't have to build these routines.

- To give the user a standard interface for common operations in Windows-based applications, Visual Basic provides a set of **common dialog boxes**, two of which are the **Open** and **Save As** dialog boxes. Such boxes are familiar to any Windows user and give your application a professional look. And, with Windows 95, some context-sensitive help is available while the box is displayed. Appendix II lists many symbolic constants used with common dialog boxes.

- The Common Dialog control is a **'custom control'** which means we have to make sure some other files are present to use it. In normal setup configurations, Visual Basic does this automatically. If the common dialog box does not appear in the Visual Basic toolbox, you need to add it. This is done by selecting **Components** under the **Project** menu. When the selection box appears, click on **Microsoft Common Dialog Control**, then click **OK**.

- The common dialog tool, although it appears on your form, is invisible at run-time. You cannot control where the common dialog box appears on your screen. The tool is invoked at run-time using one of five '**Show'** methods. These methods are:

Method	Common Dialog Box
ShowOpen	Open dialog box
ShowSave	Save As dialog box
ShowColor	Color dialog box
ShowFont	Font dialog box
ShowPrinter	Printer dialog box

- The format for establishing a common dialog box named **cdlExample** so that an **Open** box appears is:

cdlExample.ShowOpen

Control to the program returns to the line immediately following this line, once the dialog box is closed in some manner. Common dialog boxes are system modal.

- Learning proper use of all the common dialog boxes would require an extensive amount of time. In this class, we'll limit ourselves to learning the basics of getting file names from the **Open** and **Save As** boxes in their default form.

Open Common Dialog Box

- The **Open** common dialog box provides the user a mechanism for specifying the name of a file to open. We'll worry about how to open a file in Class 6. The box is displayed by using the **ShowOpen** method. Here's an example of an Open common dialog box:
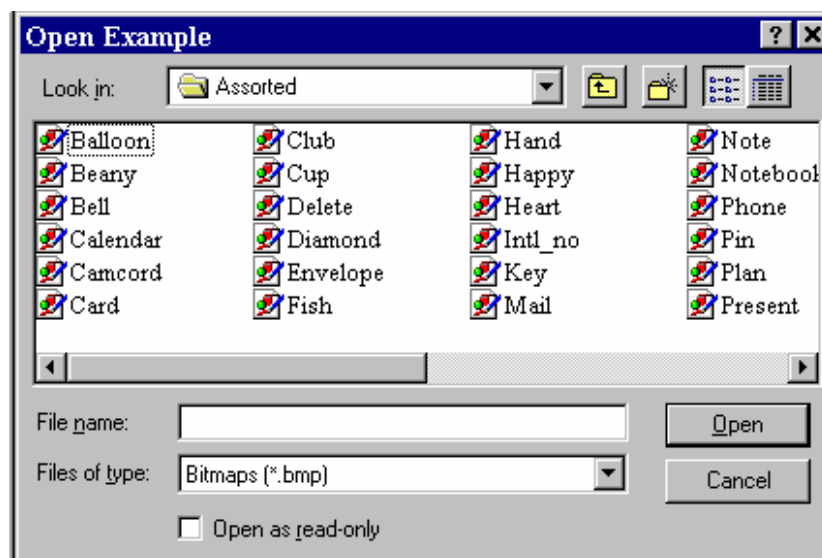
- Open Dialog Box Properties:

CancelError	If True, generates an error if the Cancel button is clicked. Allows you to use error-handling procedures to recognize that Cancel was clicked.
DialogTitle	The string appearing in the title bar of the dialog box. Default is Open. In the example, the DialogTitle is Open Example.
FileName	Sets the initial file name that appears in the File name box. After the dialog box is closed, this property can be read to determine the name of the selected file.

Filter	Used to restrict the filenames that appear in the file list box. Complete filter specifications for forming a Filter can be found using on-line help. In the example, the Filter was set to allow Bitmap (*.bmp), Icon (*.ico), Metafile (*.wmf), GIF (*.gif), and JPEG (*.jpg) types (only the Bitmap choice is seen).
FilterIndex	Indicates which filter component is default. The example uses a 1 for the FilterIndex (the default value).
Flags	Values that control special features of the Open dialog box (see Appendix II). The example uses no Flags value.

* When the user closes the Open File box, you should check the returned file name to make sure it meets the specifications your application requires before you try to open the file.

Quick Example: The Open Dialog Box

1. Start a new project. Place a common dialog control, a label box, and a command button on the form. Set the following properties:

    Form1:
Caption	Common Dialo g Examples
Name	frmCommon

    CommonDialog1:
 | | |
    |---|---|
    | DialogTitle | Open Example |
    | Filter | Bitmaps (*.bmp)|*.bmp| |
    | | Icons (*.ico)|*.ico|Metafiles (*.wmf)|*.wmf |
    | | GIF Files (*.gif)|*.gif|JPEG Files (*.jpg)|*.jpg |
    | | (all on one line) |
    | Name | cdlExample |

    Label1:
BorderStyle	1-Fixed Single
Caption	[Blank]
Name	lblExample

    Command1:
Caption	&Display Box
Name	cmdDisplay

When done, the form should look like this (make sure your label box is very long):
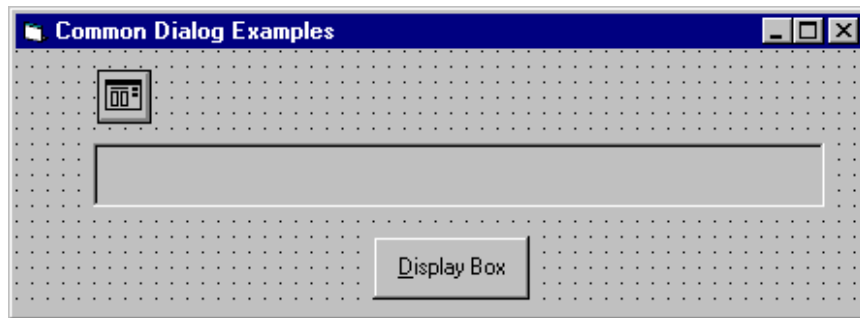
2. Attach this code to the **cmdDisplay_Click** procedure.

```
Private Sub cmdDisplay_Click()
cdlExample.ShowOpen
lblExample.Caption = cdlExample.filename
End Sub
```

This code brings up the Open dialog box when the button is clicked and shows the file name selected by the user once it is closed.

3. Save the application. Run it and try selecting file names and typing file names. Notice names can be selected by highlighting and clicking the **OK** button or just by double-clicking the file name. In this example, clicking the **Cancel** button is not trapped, so it has the same effect as clicking **OK**.

4. Notice once you select a file name, the next time you open the dialog box, that selected name appears as default, since the FileName property is not affected in code.

Save As Common Dialog Box

- The **Save As** common dialog box provides the user a mechanism for specifying the name of a file to save. We'll worry about how to save a file in Class 6. The box is displayed by using the **ShowSave** method.. Here's an example of a Save As common dialog box:
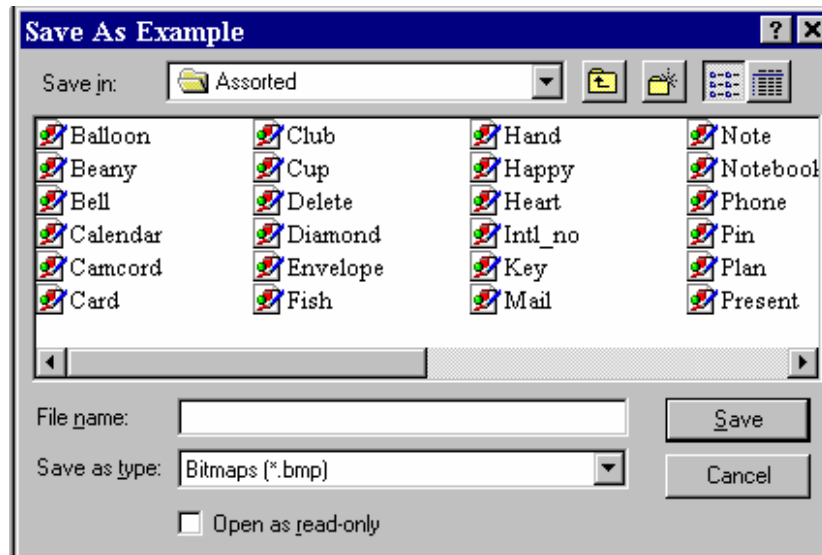
- Save As Dialog Box Properties (mostly the same as those for the Open box):

CancelError	If True, generates an error if the Cancel button is clicked. Allows you to use error-handling procedures to recognize that Cancel was clicked.
DefaultExt	Sets the default extension of a file name if a file is listed without an extension.
DialogTitle	The string appearing in the title bar of the dialog box. Default is Save As. In the example, the DialogTitle is Save As Example.
FileName	Sets the initial file name that appears in the File name box. After the dialog box is closed, this property can be read to determine the name of the selected file.
Filter	Used to restrict the filenames that appear in the file list box.
FilterIndex	Indicates which filter component is default.
Flags	Values that control special features of the dialog box (see Appendix II).

- The Save File box is commonly configured in one of two ways. If a file is being saved for the first time, the **Save As** configuration, with some default name in the FileName property, is used. In the **Save** configuration, we assume a file has been previously opened with some name. Hence, when saving the file again, that same name should appear in the FileName property. You've seen both configuration types before.

- When the user closes the Save File box, you should check the returned file name to make sure it meets the specifications your application requires before you try to save the file. Be especially aware of whether the user changed the file extension to something your application does not allow.

Quick Example: The Save As Dialog Box

1. We'll just modify the Open example a bit. Change the **DialogTitle** property of the common dialog control to "**Save As Example**" and set the **DefaultExt** property equal to "**bmp**".

2. In the **cmdDisplay_Click** procedure, change the method to **ShowSave** (opens Save As box).

3. Save the application and run it. Try typing names without extensions and note how **.bmp** is added to them. Notice you can also select file names by double-clicking them or using the **OK** button. Again, the **Cancel** button is not trapped, so it has the same effect as clicking **OK**.

Exercise 4

Student Database Input Screen

You did so well with last week's assignment that, now, a school wants you to develop the beginning structure of an input screen for its students. The required input information is:

1. Student Name
2. Student Grade (1 through 6)
3. Student Sex (Male or Female)
4. Student Date of Birth (Month, Day, Year)
5. Student Picture (Assume they can be loaded as bitmap files)

Set up the screen so that only the Name needs to be typed; all other inputs should be set with option buttons, scroll bars, and common dialog boxes. When a screen of information is complete, display the summarized profile in a message box. This profile message box should resemble this:
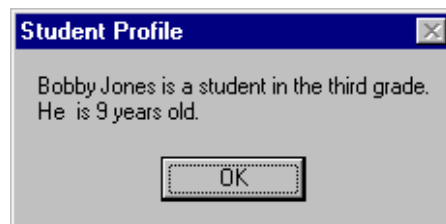
Note the student's age must be computed from the input birth date - watch out for pitfalls in doing the computation. The student's picture does not appear in the profile, only on the input screen.

My Solution:

Form:
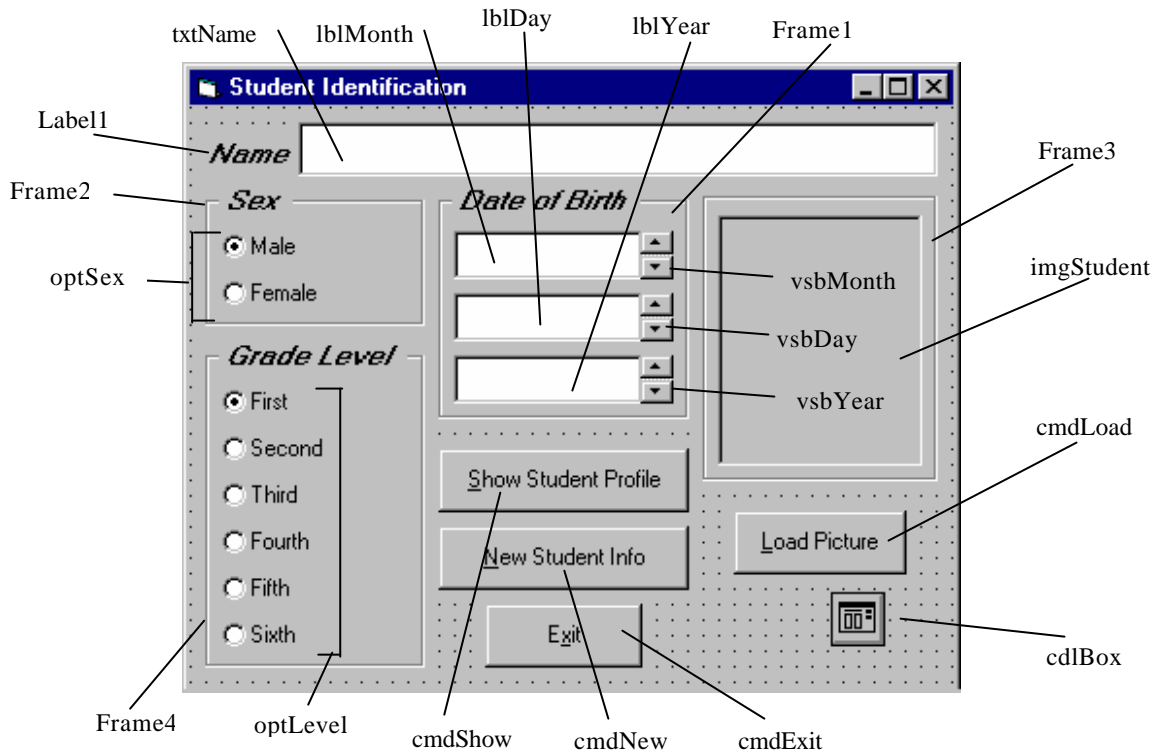
Properties:

Form **frmStudent:**
>   BorderStyle = 1- Fixed Single
>   Caption = Student Profile

CommandButton **cmdLoad**:
>   Caption = &Load Picture

Frame **Frame3**:
>   Caption = Picture
>   FontName = MS Sans Serif
>   FontBold = True
>   FontSize = 9.75
>   FontItalic = True

Image **imgStudent**:
  BorderStyle = 1 - Fixed Single
  Stretch = True

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

CommandButton **cmdNew**:
  Caption = &New Profile

CommandButton **cmdShow**:
  Caption = &Show Profile

Frame **Frame4**:
  Caption = Grade Level
  FontName = MS Sans Serif
  FontBold = True
  FontSize = 9.75
  FontItalic = True

OptionButton **optLevel**:
  Caption = Grade 6
  Index = 5

OptionButton **optLevel**:
  Caption = Grade 5
  Index = 4

OptionButton **optLevel**:
  Caption = Grade 4
  Index = 3

OptionButton **optLevel**:
  Caption = Grade 3
  Index = 2

OptionButton **optLevel**:
  Caption = Grade 2
  Index = 1

OptionButton **optLevel**:
  Caption = Grade 1
  Index = 0

Frame **Frame2**:
> Caption = Sex
> FontName = MS Sans Serif
> FontBold = True
> FontSize = 9.75
> FontItalic = True

OptionButton **optSex**:
> Caption = Female
> Index = 1

OptionButton **optSex**:
> Caption = Male
> Index = 0

Frame **Frame1**:
> Caption = Date of Birth
> FontName = MS Sans Serif
> FontBold = True
> FontSize = 9.75
> FontItalic = True

VScrollBar **vsbYear**:
> Max = 1800
> Min = 2100
> Value = 1960

VScrollBar **vsbDay**:
> Max = 1
> Min = 31
> Value = 1

VScrollBar **vsbMonth**:
> Max = 1
> Min = 12
> Value = 1

Label **lblYear**:
> Alignment = 2 - Center
> BackColor = &H00FFFFFF& (White)
> BorderStyle = 1 - Fixed Single
> FontName = MS Sans Serif
> FontSize = 10.8

Label **lblDay**:
    Alignment = 2 - Center
    BackColor = &H00FFFFFF& (White)
    BorderStyle = 1 - Fixed Single
    FontName = MS Sans Serif
    FontSize = 10.8

Label **lblMonth**:
    Alignment = 2 - Center
    BackColor = &H00FFFFFF& (White)
    BorderStyle = 1 - Fixed Single
    FontName = MS Sans Serif
    FontSize = 10.8

TextBox **txtName**:
    FontName = MS Sans Serif
    FontSize = 10.8

CommonDialog **cdlBox**:
    Filter = Bitmaps (*.bmp)|*.bmp

Label **Label1**:
    Caption = Name
    FontName = MS Sans Serif
    FontBold = True
    FontSize = 9.75
    FontItalic = True

Code:

General Declarations:

```
Option Explicit
Dim Months(12) As String
Dim Days(12) As Integer
Dim Grade As String
```


cmdExit Click Event:

```
Private Sub cmdExit_Click()
End
End Sub
```

cmdLoad Click Event:

```
Private Sub cmdLoad_Click()
cdlbox.ShowOpen
imgStudent.Picture = LoadPicture(cdlbox.filename)
End Sub
```

cmdNew Click Event:

```
Private Sub cmdNew_Click()
'Blank out name and picture
txtName.Text = ""
imgStudent.Picture = LoadPicture("")
End Sub
```

cmdShow Click Event:

```
Private Sub cmdShow_Click()
Dim Is_Leap As Integer
Dim Msg As String, Age As Integer, Pronoun As String
Dim M As Integer, D As Integer, Y As Integer

'Check for leap year and if February is current month
If vsbMonth.Value = 2 And ((vsbYear.Value Mod 4 = 0 And
vsbYear.Value Mod 100 <> 0) Or vsbYear.Value Mod 400 = 0)
Then
  Is_Leap = 1
Else
  Is_Leap = 0
End If
'Check to make sure current day doesn't exceed number of
days in month
If vsbDay.Value > Days(vsbMonth.Value) + Is_Leap Then
  MsgBox "Only" + Str(Days(vsbMonth.Value) + Is_Leap) + "
days in " + Months(vsbMonth.Value), vbOKOnly + vbCritical,
"Invalid Birth Date"
  Exit Sub
End If
'Get current date to compute age
M = Val(Format(Now, "mm"))
D = Val(Format(Now, "dd"))
Y = Val(Format(Now, "yyyy"))
Age = Y - vsbYear
If vsbMonth.Value > M Or (vsbMonth.Value = M And vsbDay >
D) Then Age = Age - 1
```

```
'Check for valid age
If Age < 0 Then
  MsgBox "Birth date is before current date.", vbOKOnly +
vbCritical, "Invalid Birth Date"
  Exit Sub
End If

'Check to make sure name entered
If txtName.Text = "" Then
  MsgBox "The profile requires a name.", vbOKOnly +
vbCritical, "No Name Entered"
  Exit Sub
End If

'Put together student profile message
Msg = txtName.Text + " is a student in the " + Grade + "
grade." + vbCr
If optSex(0).Value = True Then Pronoun = "He " Else Pronoun
= "She "
Msg = Msg + Pronoun + " is" + Str(Age) + " years old." +
vbCr
MsgBox Msg, vbOKOnly, "Student Profile"
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
'Set arrays for dates and initialize labels
Months(1) = "January": Days(1) = 31
Months(2) = "February": Days(2) = 28
Months(3) = "March": Days(3) = 31
Months(4) = "April": Days(4) = 30
Months(5) = "May": Days(5) = 31
Months(6) = "June": Days(6) = 30
Months(7) = "July": Days(7) = 31
Months(8) = "August": Days(8) = 31
Months(9) = "September": Days(9) = 30
Months(10) = "October": Days(10) = 31
Months(11) = "November": Days(11) = 30
Months(12) = "December": Days(12) = 31
lblMonth.Caption = Months(vsbMonth.Value)
lblDay.Caption = Str(vsbDay.Value)
lblYear.Caption = Str(vsbYear.Value)
Grade = "first"
End Sub
```

optLevel Click Event:

```
Private Sub optLevel_Click(Index As Integer)
Select Case Index
Case 0
   Grade = "first"
Case 1
   Grade = "second"
Case 2
   Grade = "third"
Case 3
   Grade = "fourth"
Case 4
   Grade = "fifth"
Case 5
   Grade = "sixth"
End Select
End Sub
```

vsbDay Change Event:

```
Private Sub vsbDay_Change()
lblDay.Caption = Str(vsbDay.Value)
End Sub
```

vsbMonth Change Event:

```
Private Sub vsbMonth_Change()
lblMonth.Caption = Months(vsbMonth.Value)
End Sub
```

vsbYear Change Event:

```
Private Sub vsbYear_Change()
lblYear.Caption = Str(vsbYear.Value)
End Sub
```

Learn Visual Basic 6.0

5. Creating a Stand-Alone Visual Basic Application

Review and Preview

- We've finished looking at most of the Visual Basic tools and been introduced to most of the Basic language features. Thus far, to run any of the applications studied, we needed Visual Basic. In this class, we learn the steps of developing a stand-alone application that can be run on any Windows-based machine. We'll also look at some new components that help make up applications.

Designing an Application

- Before beginning the actual process of building your application by drawing the Visual Basic interface, setting the object properties, and attaching the Basic code, many things should be considered to make your application useful.

- A first consideration should be to determine what processes and functions you want your application to perform. What are the inputs and outputs? Develop a framework or flow chart of all your application's processes.

- Decide what tools you need. Do the built-in Visual Basic tools and functions meet your needs? Do you need to develop some tools or functions of your own?

- Design your user interface. What do you want your form to look like? Consider appearance and ease of use. Make the interface consistent with other Windows applications. Familarity is good in program design.

- Write your code. Make your code readable and traceable - future code modifiers will thank you. Consider developing reusable code - modules with utility outside your current development. This will save you time in future developments.

- Make your code 'user-friendly.' Try to anticipate all possible ways a user can mess up in using your application. It's fairly easy to write an application that works properly when the user does everything correctly. It's difficult to write an application that can handle all the possible wrong things a user can do and still not bomb out.

- Debug your code completely before distrib uting it. There's nothing worse than having a user call you to point out flaws in your application. A good way to find all the bugs is to let several people try the code - a mini beta-testing program.

Using General Sub Procedures in Applications

- So far in this class, the only procedures we have studied are the event-driven procedures associated with the various tools. Most applications have tasks not related to objects that require some code to perform these tasks. Such tasks are usually coded in a general **Sub** procedure (essentially the same as a subroutine in other languages).

- Using general Sub procedures can help divide a complex application into more manageable units of code. This helps meet the above stated goals of readability and reusability.

- Defining a Sub Procedure:

  The form for a general Sub procedure named **GenlSubProc** is:

      Sub GenlSubProc(Arguments)   'Definition header
          .
          .
      End Sub

  The definition header names the **Sub** procedure and defines any arguments passed to the procedure. **Arguments** are a comma-delimited list of variables passed to and/or from the procedure. If there are arguments, they must be declared and typed in the definition header in this form:

      Var1 As Type1, Var2 As Type2, ...

- Sub Procedure Example:

  Here is a Sub procedure (**USMexConvert**) that accepts as inputs an amount in US dollars (**USDollars**) and an exchange rate (**UStoPeso**). It then outputs an amount in Mexican pesos (**MexPesos**).

  ```
  Sub USMexConvert (USDollars As Single, UStoPeso As Single,
  MexPesos As Single)
  MexPesos = UsDollars * UsToPeso
  End Sub
  ```

- Calling a Sub Procedure:

  There are two ways to **call** or invoke a **Sub** procedure. You can also use these to call event-driven procedures.

  Method 1:

  ```
  Call GenlSubProc(Arguments)
  ```
  (if there are no Arguments, do not type the parentheses)

  Method 2:

  ```
  GenlSubProc Arguments
  ```

  I prefer Method 1 - it's more consistent with calling protocols in other languages and it cleanly delineates the argument list. It seems most Visual Basic programmers use Method 2, though. I guess they hate typing parentheses! Choose the method you feel more comfortable with.

  Example

  To call our dollar exchange routine, we could use:

  ```
  Call USMexConvert (USDollars, UStoMex, MexPesos)
  ```

  or

  ```
  USMexConvert USDollars, UStoMex, MexPesos
  ```

- Locating General Sub Procedures:

  General Sub procedures can be located in one of two places in your application: attached to a **form** or attached to a **module**. Place the procedure in the form if it has a purpose specifically related to the form. Place it in a module if it is general purpose procedure that might be used by another form or module or another application.

  Whether placing the procedure in a form or module, the methods of creating the procedure are the same. Select or open the form or module's code window. Make sure the window's **Object** list says (**General**) and the **Procedure** list says (**Declarations**). You can now create the procedure by selecting **Add Procedure** from Visual Basic's **Tools** menu. A window appears allowing you to select Type **Sub** and enter a name for your procedure. Another way to create a Sub is to go to the last line of the General Declarations section, type **Sub** followed by a space and the name of your procedure. Then, hit **Enter**. With either method for establishing a Sub, Visual Basic will form a template for your procedure. Fill in the Argument list and write your Basic code. In selecting the Insert Procedure menu item, note another option for your procedure is **Scope**. You have the choice of **Public** or **Private**. The scope word appears before the Sub word in the definition heading. If a module procedure is Public, it can be called from any other procedure in any other module. If a module procedure is Private, it can only be called from the module it is defined in. Note, scope only applies to procedures in modules. By default, all event procedures and general procedures in a form are Private - they can only be called from within the form. You must decide the scope of your procedures.

- Passing Arguments to Sub Procedures:

  A quick word on **passing arguments** to procedures. By default, they are passed by **reference**. This means if an argument is changed within the procedure, it will remain changed once the procedure is exited.

  C programmers have experienced the concept of passing by **value**, where a parameter changed in a procedure will retain the value it had prior to calling the routine. Visual Basic also allows calling by value. To do this, place the word **ByVal** in front of each such variable in the Argument list.

Creating a Code Module

• If you're going to put code in a module, you'll need to know how to create and save a module. A good way to think about modules is to consider them forms without any objects, just code.

• To create a module, click on the **New Module** button on the toolbar, or select the **Module** option from the **Insert** menu. The module will appear. Note any modules appear in the Project Window, along with your form(s). You use the Project Window to move among forms and modules.

• Once the module is active, establish all of your procedures as outlined above. To name the module, click on the properties window while the module is active. Note **Name** is the only property associated with a module. Saving a module is just like saving a form - use the Save File and Save File As options.

Using General Function Procedures in Applications

• Related to Sub procedures are **Function** procedures. A Function procedure, or simply Function, performs a specific task within a Visual Basic program and returns a value. We've seen some built-in functions such as the **MsgBox** and the **Format** function.

• Defining a Function:

The form for a general Function named **GenlFcn** is:

    Function GenlFcn(Arguments) As Type 'Definition header
        .
        .
    GenlFcn = ...
    End Function

The definition header names the **Function** and specifies its Type (the type of the returned value) and defines any input **Arguments** passed to the function. Note that somewhere in the function, a value for GenlFcn must be computed for return to the calling procedure.

- Function Example:

  Here is a Function named **CylVol** that computes the volume of a cylinder of known height (**Height**) and radius (**Radius**).

  ```
  Function CylVol(Height As Single, Radius As Single) As Single
  Dim Area As Single
  Const PI = 3.1415926
  Area = PI * Radius ^ 2
  CylVol = Area * Height
  End Sub
  ```

- Calling a Function:

  To **call** or use a **Function**, you equate a variable (of proper type) to the Function, with its arguments. That is, if the Function **GenlFunc** is of Type Integer, then use the code segment:

  ```
  Dim RValue as Integer
       .
       .
  RValue = GenlFunc(Arguments)
  ```

  Example

  To call the volume computation function, we could use:

  ```
  Dim Volume As Single
       .
       .
  Volume = CylVol(Height, Radius)
  ```

- Locating Function Procedures:

  Like Sub procedures, Functions can be located in forms or modules. They are created using exactly the same process described for Sub procedures, the only difference being you use the keyword **Function**.

  And, like Sub procedures, Functions (in modules) can be Public or Private.

Quick Example: Temperature Conversion

1. Open the Temperature Conversion application from last class. Note in the **vsbTemp_Change** and **vsbTemp_Scroll** procedures, there is a lot of repeated code. We'll replace this code with a **Sub** procedure that prints the values and a **Function** procedure that does the temperature conversion.

2. Add a module to your application. Create a Function (Public by default) named **DegF_To_DegC**.

```
Public Function DegF_To_DegC(DegF As Integer) As Integer
DegF_To_DegC = CInt((DegF - 32) * 5 / 9)
End Function
```

3. Go back to your form. Create a **Sub** procedure named **ShowTemps**. Fill in the code by cutting from an old procedure. Note this code uses the new Function to convert temperature and prints both values in their respective label boxes.

```
Private Sub ShowTemps()
lblTempF.Caption = Str(TempF)
TempC = DegF_To_DegC(TempF)
lblTempC.Caption = Str(TempC)
End Sub
```

No arguments are needed since TempF and TempC are global to the form.

4. Rewrite the **vsbTemp_Change** and **vsbTemp_Scroll** procedures such that they call the new Sub procedure:

```
Private Sub vsbTemp_Change()
TempF = vsbTemp.Value
Call ShowTemps
End Sub

Private Sub vsbTemp_Scroll()
Call vsbTemp_Change
End Sub
```

Note how **vsbTemp_Scroll** simply calls **vsbTemp_Change** since they use the same code. This is an example of calling an event procedure.

5. Save the application and run it. Note how much neater and modular the code is.

Quick Example: Image Viewer (Optional)

1. Open the Image Viewer application from last class. Note the code in the
    cmdShow_Click and **filImage_DblClick** events is exactly the same. Delete the code
    in the **filImage_DblClick** procedure and simply have it call the **cmdShow_Click**
    procedure. That is, replace the **filImage_DblClick** procedure with:

```
Private Sub filImage_DblClick()
Call cmdShow_Click
End Sub
```

2. This is another example of calling an event procedure. Save your application.

Adding Menus to an Application

*   As mentioned earlier, it is important that the interface of your application be familar
    to a seasoned, or not-so-seasoned, Windows user. One such familiar application
    component is the Menu bar. Menus are used to provide a user with choices that
    control the application. Menus are easily incorporated into Visual Basic programs
    using the **Menu Editor**.

*   A good way to think about elements of a menu structure is to consider them as a
    hierarchical list of command buttons that only appear when pulled down from the
    menu bar. When you click on a menu item, some action is taken. Like command
    buttons, menu items are named, have captions, and have properties.

    Example

    Here is a typical menu structure:

File	Edit	Format
New	Cut	Bold
Open	Copy	Italic
Save	Paste	Underline
————		Size
Exit		10
		15
		20

The underscored characters are access keys, just like those on command buttons. The level of indentation indicates position of a menu item within the hierarchy. For example, **New** is a sub-element of the **File** menu. The line under **Save** in the **File** menu is a separator bar (separates menu items).

With this structure, the Menu bar would display:

    File   Edit   Format

The sub-menus appear when one of these 'top' level menu items is selected. Note the **Size** sub-menu under **Format** has another level of hierarchy. It is good practice to not use more than two levels in menus. Each menu element will have a **Click** event associated with it.

- The **Menu Editor** allows us to define the menu structure, adding access keys and shortcut keys, if desired. We then add code to the **Click** events we need to respond to. The Menu Editor is selected from the **Tools** menu bar or by clicking the **Menu Editor** on the toolbar. This selection can only be made when the form needing the menu is active. Upon selecting the editor, and entering the example menu structure, the editor window looks like this:
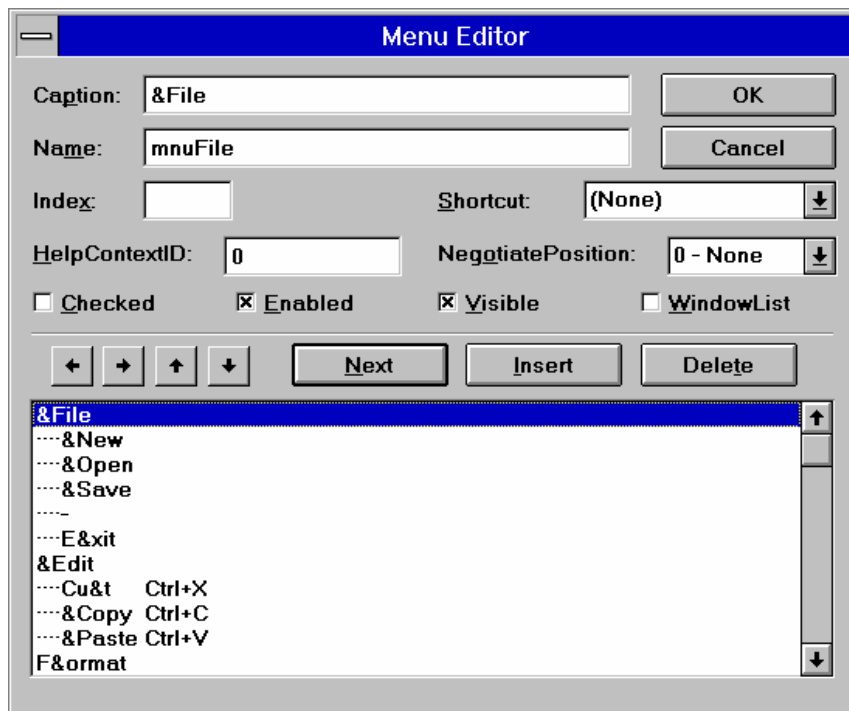
Each item in the menu structure requires several entries in this design box.

- The **Caption** box is where you type the text that appears in the menu bar. Access keys are defined in the standard way using the ampersand (&). Separator bars (a horizontal line used to separate menu items) are defined by using a Caption of a single hyphen (-). When assigning captions and access keys, try to use conform to any established Windows standards.

- The **Name** box is where you enter a control name for each menu item. This is analogous to the Name property of command buttons and is the name used to set properties and establish the **Click** event procedure for each menu item. Each menu item must have a name, even separator bars! The prefix **mnu** is used to name menu items. Sub-menu item names usually refer back to main menu headings. For example, if the menu item **New** is under the main heading **File** menu, use the name **mnuFileNew**.

- The **Index** box is used for indexing any menu items defined as control arrays.

- The **Shortcut** dropdown box is used to assign shortcut keystrokes to any item in a menu structure. The shortcut keystroke will appear to the right of the caption for the menu item. An example of such a keystroke is using Ctrl+X to cut text.

- The **HelpContextID** and **NegotiatePosition** boxes relate to using on-line help and object linking embedding, and are beyond the scope of this discussion.

- Each menu item has four properties associated with it. These properties can be set at design time using the Menu Editor or at run-time using the standard dot notation. These properties are:

Checked	Used to indicate whether a toggle option is turned on or off. If True, a check mark appears next to the menu item.
Enabled	If True, menu item can be selected. If False, menu item is grayed and cannot be selected.
Visible	Controls whether the menu item appears in the structure.
WindowList	Used with Multiple Document Interface (MDI) - not discussed here.

- At the bottom of the Menu Editor form is a list box displaying the hierarchical list of menu items. Sub-menu items are indented to their level in the hierarchy. The right and left arrows adjust the levels of menu items, while the up and down arrows move items within the same level. The **Next**, **Insert**, and **Delete** buttons are used to move the selection down one line, insert a line above the current selection, or delete the current selection, respectively.

- Let's look at the process of entering the example menu structure. To do this, we '**stack'** the three menus on top of each other, that is enter items as a long list. For each item in this list, we provide a Caption (with access key, if any), a Name (indicative of where it is in the structure), a shortcut key (if any), and provide proper indentation to indication hierarchical location (use the left and right arrows to move in and out).

- After entering this structure, the complete list box at the bottom of the Menu Editor would look like this (notice access keys are indicated with ampersands and shortcut keys are listed at the right, and, the assigned names are shown at the left - these don't really appear in the Menu Editor list box; they are shown to illustrate one possible naming convention):

Name

mnuFile	&File	
mnuFileNew	.......&New	
mnuFileOpen	.......&Open	
mnuFileSave	.......&Save	
mnuFileBar	.......-	
mnuFileExit	.......E&xit	
mnuEdit	&Edit	
mnuEditCut	.......Cu&t	Ctrl+X
mnuEditCopy	.......&Copy	Ctrl+C
mnuEditPaste	.......&Paste	Ctrl+V
mnuFmt	F&ormat	
mnuFmtBold	.......Bold	
mnuFmtItalic	.......Italic	
mnuFmtUnderline	.......Underline	
mnuFmtSize	.......Size	
mnuFmtSize10	10	
mnuFmtSize15	15	
mnuFmtSize20	20	

- At first, using the Menu Editor may seem a little clunky. After you've done a couple of menu structures, however, its use becomes obvious. A neat thing is: after setting up your menu, you can look at it in the Visual Basic design mode and see if it looks like it should. In the next example, you'll get practice setting up a similar menu structure.

Example 5-1

Note Editor

1. Start a new project. We will use this application the rest of this class. We will build a note editor with a menu structure that allows us to control the appearance of the text in the editor box. Since this is the first time we've built menus, I'll provide the steps involved.

2. Place a large text box on a form. Set the properties of the form and text box:

Form1:
BorderStyle	1-Fixed Single
Caption	Note Editor
Name	frmEdit

Text1:
BorderStyle	1-Fixed Single
MultiLine	True
Name	txtEdit
ScrollBars	2-Vertical
Text	[Blank]

The form should look something like this when you're done:
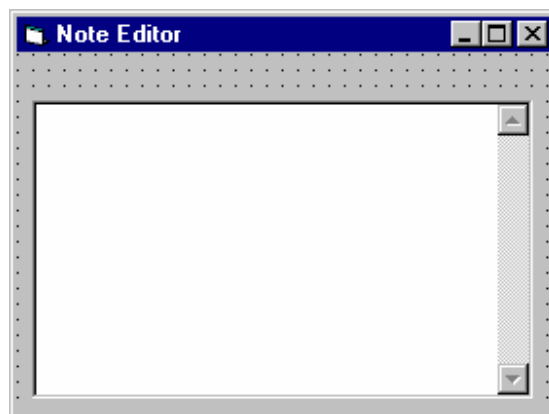
3. We want to add this menu structure to the Note Editor:

<u>F</u>ile   F<u>o</u>rmat
          <u>N</u>ew   <u>B</u>old
          —————   <u>I</u>talic
   E<u>x</u>it   <u>U</u>nderline
                              <u>S</u>ize
                                   <u>S</u>mall
                                   <u>M</u>edium
                                   <u>L</u>arge

Note the identified access keys. Bring up the Menu Editor and assign the following
    Captions, Names, and Shortcut Keys to each item. Make sure each menu item is at is
    proper location in the hierarchy.

Caption	Name	Shortcut
&File	mnuFile	[None]
&New	mnuFileNew	[None]
-	mnuFileBar	[None]
E&xit	mnuFileExit	[None]
F&ormat	mnuFmt	[None]
& Bold	mnuFmt Bold	Ctrl+B
&Italic	mnuFmtItalic	Ctrl+I
&Underline	mnuFmtUnderline	Ctrl+U
&Size	mnuFmtSize	[None]
&Small	mnuFmtSizeSmall	Ctrl+S
&Medium	mnuFmtSizeMedium	Ctrl+M
&Large	mnuFmtSizeLarge	Ctrl+L

The **Small** item under the **Size** sub-menu should also be **Checked** to indicate the initial
    font size. When done, look through your menu structure in design mode to make sure
    it looks correct. With a menu, the form will appear like:
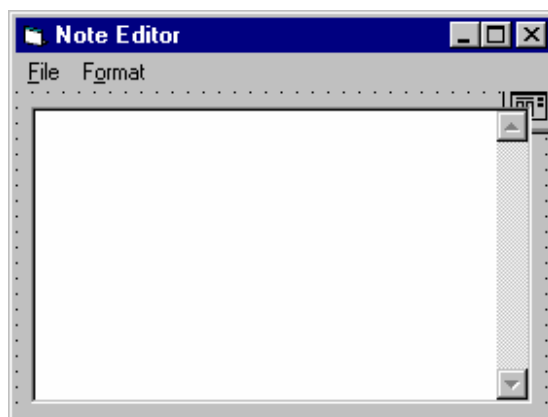
4. Each menu item that performs an action requires code for its **Click** event. The only menu items that do <u>not</u> have events are the menu and sub-menu headings, namely File, Format, and Size. All others need code. Use the following code for each menu item **Click** event. (This may look like a lot of typing, but you should be able to use a lot of cut and paste.)

If **mnuFileNew** is clicked, the program checks to see if the user really wants a new file and, if so (the default response), clears out the text box:

```
Private Sub mnuFileNew_Click()
'If user wants new file, clear out text
Dim Response As Integer
Response = MsgBox("Are you sure you want to start a new
   file?", vbYesNo + vbQuestion, "New File")
If Response = vbYes Then txtEdit.Text = ""
End Sub
```

If **mnuFileExit** is clicked, the program checks to see if the user really wants to exit. If not (the default response), the user is returned to the program:

```
Private Sub mnuFileExit_Click()
'Make sure user really wants to exit
Dim Response As Integer
Response = MsgBox("Are you sure you want to exit the note
   editor?", vbYesNo + vbCritical + vbDefaultButton2, "Exit
   Editor")
If Response = vbNo Then
  Exit Sub
Else
  End
End If
End Sub
```

If **mnuFmtBold** is clicked, the program toggles the current bold status:

```
Private Sub mnuFmtBold_Click()
'Toggle bold font status
mnuFmtBold.Checked = Not (mnuFmtBold.Checked)
txtEdit.FontBold = Not (txtEdit.FontBold)
End Sub
```

If **mnuFmtItalic** is clicked, the program toggles the current italic status:

```
Private Sub mnuFmtItalic_Click()
'Toggle italic font status
mnuFmtItalic.Checked = Not (mnuFmtItalic.Checked)
txtEdit.FontItalic = Not (txtEdit.FontItalic)
End Sub
```

If **mnuFmtUnderline** is clicked, the program toggles the current underline status:

```
Private Sub mnuFmtUnderline_Click()
'Toggle underline font status
mnuFmtUnderline.Checked = Not (mnuFmtUnderline.Checked)
txtEdit.FontUnderline = Not (txtEdit.FontUnderline)
End Sub
```

If either of the three size sub-menus is clicked, indicate the appropriate check mark location and change the font size:

```
Private Sub mnuFmtSizeSmall_Click()
'Set font size to small
mnuFmtSizeSmall.Checked = True
mnuFmtSizeMedium.Checked = False
mnuFmtSizeLarge.Checked = False
txtEdit.FontSize = 8
End Sub


Private Sub mnuFmtSizeMedium_Click()
'Set font size to medium
mnuFmtSizeSmall.Checked = False
mnuFmtSizeMedium.Checked = True
mnuFmtSizeLarge.Checked = False
txtEdit.FontSize = 12
End Sub


Private Sub mnuFmtSizeLarge_Click()
'Set font size to large
mnuFmtSizeSmall.Checked = False
mnuFmtSizeMedium.Checked = False
mnuFmtSizeLarge.Checked = True
txtEdit.FontSize = 18
End Sub
```

5. Save your application. We will use it again in Class 6 where we'll learn how to save and open text files created with the Note Editor. Test out all the options. Notice how the toggling of the check marks works. Try the shortcut keys.

Using Pop-Up Menus

*   **Pop-up menus** can show up anywhere on a form, usually being activated by a single or double-click of one of the two mouse buttons. Most Windows applications, and Windows itself, use pop-up menus. For example, using the right hand mouse button on almost any object in Windows 95 will display a pop-up menu. In fact, with the introduction of such pop-up menus with Windows 95, the need for adding such menus to Visual Basic applications has been reduced.

*   Adding pop-up menus to your Visual Basic application is a two step process. First, you need to create the menu using the **Menu Editor** (or, you can use any existing menu structure with at least one sub-menu). If creating a unique pop-up menu (one that normally does not appear on the menu bar), it's **Visible** property is set to be **False** at design time. Once created, the menu is displayed on a form using the **PopupMenu** method.

*   The PopupMenu method syntax is:

    ObjectName.**PopupMenu** MenuName, Flags, X, Y

    The ObjectName can be omitted if working with the current form. The arguments are:

MenuName	Full-name of the pop-up menu to display.
Flags	Specifies location and behavior of menu (optional).
X, Y	(X, Y) coordinate of menu in twips (optional; if either value is omitted, the current mouse coordinate is used).

*   The **Flags** setting is the sum of two constants. The first constant specifies location:

Value	Meaning	**Symbolic** Constant
0	Left side of menu is at X coordinate	vbPopupMenuLeftAlign
4	Menu is centered at X coordinate	vbPopupMenuCenterAlign
8	Right side of menu is at X coordinate	vbPopupMenuRightAlign

The second specifies behavior:

Value	Meaning	Symbolic Constant
0	Menu reacts only to left mouse button	vbPopupMenuLeftButton
2	Menu reacts to either mouse button	vbPopupMenuRightButton

- You need to decide where to put the code that displays the pop-up menu, that is the **PopupMenu** method. Usually, a pop-up menu is displayed in response to a **Click** event or **MouseDown** event. The standard (starting with Windows 95) seems to be leaning toward displaying a pop-up menu in response to a **right** mouse **button** click.

- Just like other menus, each item on your pop-up menu will need code for the corresponding **Click** event. Many times, though, the code is simply a call to an existing menu item's Click event.

Assigning Icons to Forms

- Notice that whenever you run an application, a small icon appears in the upper left hand corner of the form. This icon is also used to represent the form when it is minimized at run-time. The icon seen is the default Visual Basic icon for forms. Using the **Icon** property of a form, you can change this displayed icon.

- The idea is to assign a unique icon to indicate the form's function. To assign an icon, click on the Icon property in the Property Window for the form. Click on the ellipsis (...) and a window that allows selection of icon files will appear.

- The icon file you load must have the **.ico** filename extension and format. When you first assign an icon to a form (at design time), it will not appear on the form. It will only appear after you have run the application once.

Designing Your Own Icon with IconEdit

- Visual Basic offers a wealth of icon files from which you could choose an icon to assign to your form(s). But, it's also fun to design your own icon to add that personal touch.

- *PC Magazine* offers a free utility called **IconEdit** that allows you to design and save icons. Included with these notes is this program and other files (directory IconEdit). To install these files on your machine, copy the folder to your hard drive.

- To run IconEdit, click **Start** on the Windows 95 task bar, then click **Run**. Find the **IconEdit.exe** program (use Browse mode). You can also establish an shortcut to start IconEdit from your desktop, if desired. The following Editor window will appear:
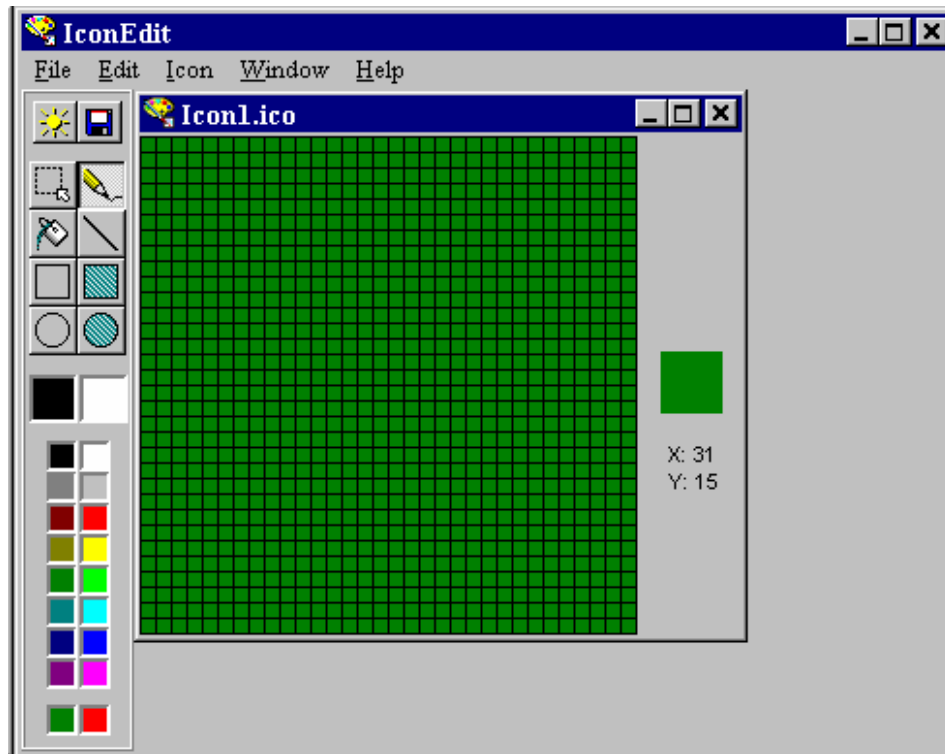
- The basic idea of IconEdit is to draw an icon in the 32 x 32 grid displayed. You can draw single points, lines, open rectangles and ovals, and filled rectangles and ovals. Various colors are available. Once completed, the icon file can be saved for attaching to a form.

- Another fun thing to do with IconEdit is to load in Visual Basic icon files and see how much artistic talent really goes into creating an icon.

- We won't go into a lot of detail on using the IconEdit program here - I just want you to know it exists and can be used to create and save icon files. Its use is fairly intuitive. Consult the on-line help of the program for details. And, there is a .**txt** file included that is very helpful.

Creating Visual Basic Executable Files

* Up to now, to run any of the applications created, we needed Visual Basic. The goal of creating an application is to let others (without Visual Basic) use it. This is accomplished by creating an **executable** version of the application.

* Before creating an executable, you should make sure your program is free of bugs and operating as desired. Save all forms, modules, and project files. Any later changes to the application will require re-creating the executable file.

* The executable file will have the extension **.exe**. To create an exe file for your application, select **Make [Project name] exe** from Visual Basic's **File** menu. This will display the Make EXE File dialog box, where you name the exe file. To open the Options box, click that button. The EXE Options dialog box will appear:
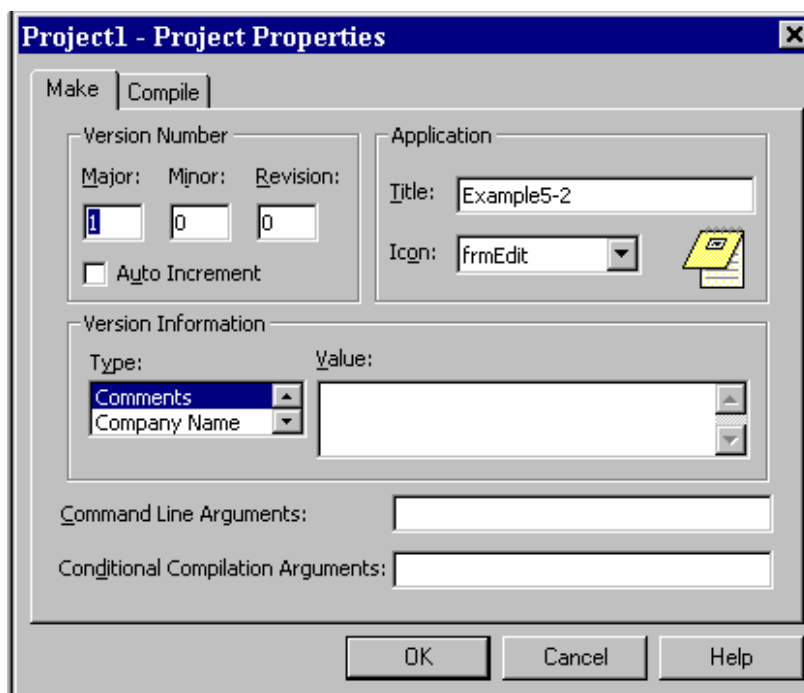
* We'll only concern ourselves with two pieces of information in this box: **Title** and **Icon**. The Title is the name you want to give your application. It does not have to be the same as the Project name. The Icon is selected from icons assigned to form(s) in your application. The selected icon is used to identify the application everywhere it is needed in Windows 95.

* Once you have selected these options, return to the Make EXE File dialog box, select a directory (best to use the same directory your application files are in) and name for your executable file. Click **OK** and the exe file is created.

- Use Windows Explorer to confirm creation of the file. And, while there, double-click the executable file name and the program will run!

Example 5-2

Note Editor - Building an Executable and Attaching an Icon

1. Open your Note Editor project. Attach an icon to the form by setting the **Icon** property. If you want to, open up the Image Viewer project from last class to take a look at icon files. The icon I used is **note.ico**

2. Create an executable version of your Note Editor. Confirm creation of the **exe** file and run it under the Windows Explorer.

3. Something you might want to try with your application is create a Windows 95 shortcut to run your program, that is, display a clickable icon. To get started, click the **Start** button on the taskbar, then **Settings**, then **Taskbar**. Here, you can add programs to those that show up when you select Start. The process is straightforward. Find your application, specify the folder you want the shortcut to appear in, and name your application. When done, the icon will appear in the specified location.

Using the Visual Basic Package & Deployment Wizard

- We were able to run the Note Editor executable file because Visual Basic is installed on our system. If you gave someone a copy of your exe file and they tried to run it, it wouldn't work (unless they have Visual Basic installed also). The reason it wouldn't run is that the executable file also needs some ancillary files (primarily, so-called dynamic link libraries) to run properly. These libraries provide most of the code associated with keeping things on a form working properly.

- So to allow others to run your application, you need to give them the executable file (exe) and at least two dynamic link libraries. Unfortunately, these dynamic link libraries take up over 1 Meg of disk space, so it's hard to move those around on a floppy disk.

- Visual Basic solves this 'distribution problem' by providing a very powerful tool called the Visual Basic **Package & Deployment Wizard**. This wizard is installed along with Visual Basic.

- The Package & Deployment Wizard prepares your application for distribution. It helps you determine which files to distribute, creates a Setup program (written in Visual Basic) that works like all other Windows Setup programs (**setup.exe**), compresses all required files to save disk space, and writes the compressed files to the distribution media of choice, usually floppy disk(s).

- To start the Package & Deployment Wizard, click the **Start** button in Windows, then find the **Visual Basic** program folder - click on **Visual Basic Tools**, then choose **Package & Deployment Wizard** The setup follows several steps. The directions provided with each step pertain to the simple applications we develop in class. For more complicated examples, you need to modify the directions, especially regarding what files you need to distribute with your application.

Step 1.   Initial Information. Enter the path and file name for your project file (.vbp). Click the ellipsis (...) to browse vbp files. If an executable (.exe) file does not exist, one will be created. Click the 'Package' button to continue. If you have previously saved a setup package for the selected project, it will load the package file created during that session.
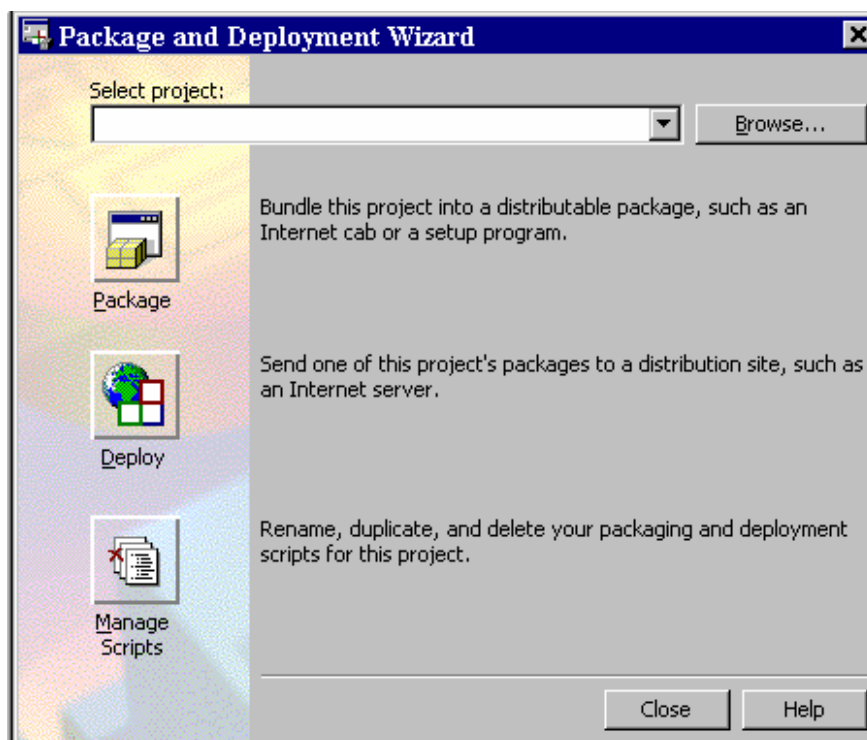
Step 2.   Package Type. Choose the Standard Setup Package (we want a standard setup program). Click Next to continue.

Step 3.   Package Folder. Select a directory where you want the application distribution package to be saved. Click Next to continue. Click Back to return to the previous step.

Step 4.   Included Files. The Package & Deployment Wizard will list all files it believes are required for your application to function properly. If your application requires any files not found by the wizard (for example, external data files you have created), you would add them to the setup list here (click Add). To continue, click Next. Click Back to return to the previous step.

Step 5.   Cab Options. Distribution files are called cab files (have a cab extension). You can choose a Single cab file written to your hard drive (if you use CD ROM distribution), or Multiple cab files (to allow distribution on floppy disks). If you choose, Multiple, you also specify the capacity of the disks you will use to write your distribution file(s). Make your choice. Click Next to Continue. Click Back to return to the previous step.

Step 6.   Installation Title. Enter a title you want your application to have. Click Next to Continue. Click Back to return to previous step.

Step 7.   Start Menu Items. This step determines where your installed application will be located on the user's Start menu. We will use the default choice. Click Next to Continue. Click Back to return to previous step.

Step 8.   Install Locations. The wizard gives you an opportunity to change the locations of installed files. Click Next to Continue. Click Back to return to previous step.

Step 9.   Shared Files. Some files in your application may be shared by other applications. Shared files will not be removed if the application is uninstalled. Decide if you have shared files. Click Next to Continue. Click Back to return to previous step.

Step 10.   Finished! Provide a name for saving the script for this wizard session (a file that saves answers to all the questions you just answered). Click Finish to Continue. Click Back to return to previous step. The wizard will create and write the cab files and tell you where they are saved. Click Close. You will be returned to the Package & Deployment Wizard opening window. Click Close.

Step 11.   Write Distribution Media. This is not a step in the wizard, but one you must take. The cab files (distribution files) the wizard wrote must now be copied to your distribution media. If you wrote a single cab file (for CD ROM), copy that file, the setup.exe file (the setup application), and the setup.lst file to your CD ROM). If you wrote multiple files (for floppy disks), copy the setup.exe, setup.lst, and first cab file (1 at end of file name) to floppy number 1. Copy the second cab file (2 at end of file name) to floppy number 2. Copy all subsequent cab files to as many floppies as needed. Properly label all floppies.

• To install the application using the distribution CD ROM or floppy disk(s), a user simply puts CD ROM or floppy number 1 in a drive. Then, through the Windows Explorer, run the **setup.exe** program. The user will be taken through the installation procedure step-by-step. The procedure is nearly identical to the installation process for all Microsoft products.

• The Package & Deployment Wizard is a very powerful tool. We've only looked at using it for simple applications. As your programming skills begin to include database access and other advanced topics, you will need to become familiar with other files that should be distributed with your applications.

Example 5-3

Note Editor - Creating a Distribution Disk

1. Open your Note Editor project again. Create a distribution disk using the Package & Deployment Wizard.

2. Try installing the application on your computer. Better yet, take the disk to another Windows 95/98/NT-based machine, preferably without Visual Basic installed. Install the application using the distribution disk and test its operation.

This page intentionally not left blank.

Exercise 5

US Capitals Quiz

Develop an application that quizzes a user on states and capitals in the United States.
Use a menu structure that allows the user to decide whether they want to name states or
capitals and whether they want mulitple choice or type-in answers. Throughly test your
application. Design an icon for your program using IconEdit or some other program
Create an executable file. Create a distribution disk using the Application Setup Wizard.
Give someone your application disk and have them install it on their computer and try out
your nifty little program.

My Solution:

Form:
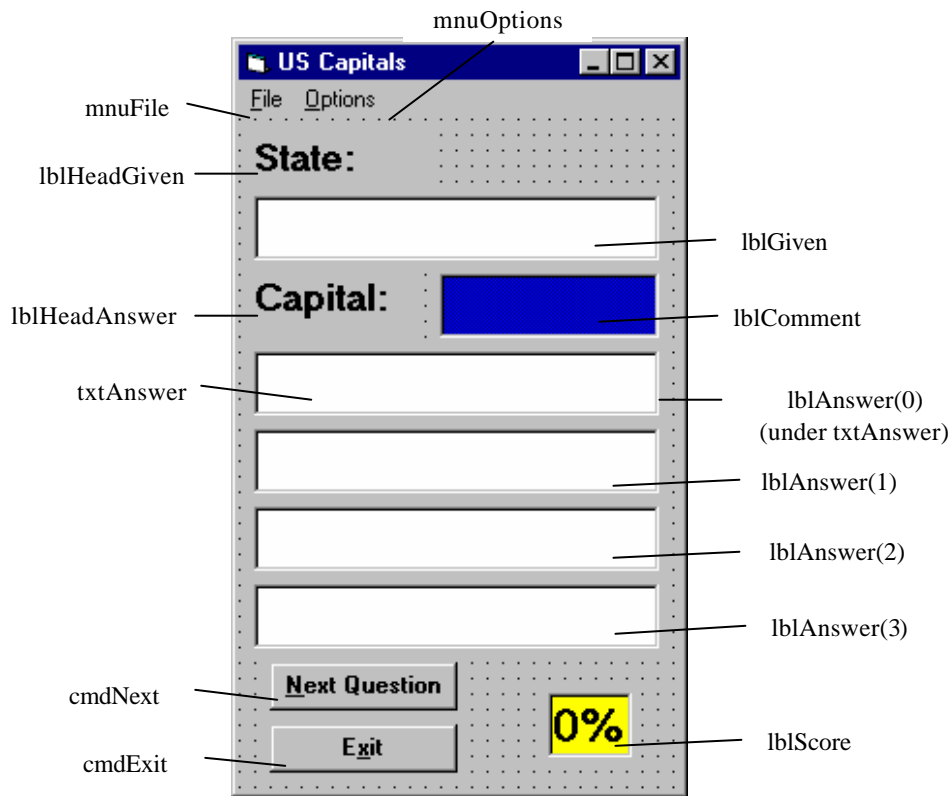
Properties:

Form **frmCapitals** :
>   BorderStyle = 1 - Fixed Single
>   Caption = US Capitals

CommandButton **cmdNext**:
>   Caption = &Next Question
>   Enabled = False

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

TextBox **txtAnswer**:
>   FontName = MS Sans Serif
>   FontSize = 13.2
>   Visible = False

Label **lblComment**:
>   Alignment = 2 - Center
>   BackColor = &H00C00000& (Blue)
>   BorderStyle = 1 - Fixed Single
>   FontName = MS Sans Serif
>   FontSize = 13.2
>   FontItalic = True
>   ForeColor = &H0000FFFF& (Yellow)

Label **lblScore** :
>   Alignment = 2 - Center
>   AutoSize = True
>   BackColor = &H0000FFFF& (Yellow)
>   BorderStyle = 1 - Fixed Single
>   Caption = 0%
>   FontName = MS Sans Serif
>   FontSize = 15.6
>   FontBold = True

Label **lblAnswer** (control array):
>   Alignment = 2 - Center
>   BackColor = &H00FFFFFF& (White)
>   BorderStyle = 1 - Fixed Single
>   FontName = MS Sans Serif
>   FontSize = 13.2
>   Index = 0, 1, 2, 3

Label **lblHeadAnswer**:
    Caption = Capital:
    FontName = MS Sans Serif
    FontSize = 13.2
    FontBold = True

Label **lblHeadGiven**:
    Caption = State:
    FontName = MS Sans Serif
    FontSize = 13.2
    FontBold = True

Menu **mnuFile**:
    Caption = &File

Menu **mnuFileNew**:
    Caption = &New

Menu **mnuFileBar**:
    Caption = -

Menu **mnuFileExit**:
    Caption = E&xit

Menu **mnuOptions**:
    Caption = &Options

Menu **mnuOptionsCapitals**:
    Captio n = Name &Capitals
    Checked = True

Menu **mnuOptionsState**:
    Caption = Name &State

Menu **mnuOptionsBar**:
    Caption = -

Menu **mnuOptionsMC**
    Caption = &Multiple Choice Answers
    Checked = True

Menu **mnuOptionsType** :
    Caption = &Type In Answers

Code:

General Declarations:

```
Option Explicit
Dim CorrectAnswer As Integer
Dim NumAns As Integer, NumCorrect As Integer
Dim Wsound(26) As Integer
Dim State(50) As String, Capital(50) As String
```

SoundEx General Function (this is a neat little function to check if spelling of two words is similar):

```
Private Function SoundEx(W As String, Wsound() As Integer)
As String
'Generates Soundex code for W
'Allows answers whose spelling is close, but not exact
Dim Wtemp As String, S As String
Dim L As Integer, I As Integer
Dim Wprev As Integer, Wsnd As Integer, Cindex As Integer
Wtemp = UCase(W)
L = Len(W)
If L <> 0 Then
  S = Left(Wtemp, 1)
  Wprev = 0
  If L > 1 Then
    For I = 2 To L
      Cindex = Asc(Mid(Wtemp, I, 1)) - 64
      If Cindex >= 1 And Cindex <= 26 Then
        Wsnd = Wsound(Cindex) + 48
        If Wsnd <> 48 And Wsnd <> Wprev Then S = S +
Chr(Wsnd)
        Wprev = Wsnd
      End If
    Next I
  End If
Else
  S = ""
End If
SoundEx = S
End Function
```

Update_Score General Procedure:

```
Private Sub Update_Score(Iscorrect As Integer)
Dim I As Integer
'Check if answer is correct
cmdNext.Enabled = True
cmdNext.SetFocus
If Iscorrect = 1 Then
  NumCorrect = NumCorrect + 1
  lblComment.Caption = "Correct!"
Else
  lblComment.Caption = "Sorry ..."
End If
'Display correct answer and update score
If mnuOptionsMC.Checked = True Then
  For I = 0 To 3
    If mnuOptionsCapitals.Checked = True Then
      If lblAnswer(I).Caption <> Capital(CorrectAnswer)
Then
        lblAnswer(I).Caption = ""
      End If
    Else
      If lblAnswer(I).Caption <> State(CorrectAnswer) Then
        lblAnswer(I).Caption = ""
      End If
    End If
  Next I
Else
  If mnuOptionsCapitals.Checked = True Then
    txtAnswer.Text = Capital(CorrectAnswer)
  Else
    txtAnswer.Text = State(CorrectAnswer)
  End If
End If
lblScore.Caption = Format(NumCorrect / NumAns, "##0%")
End Sub
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
'Exit program
Call mnuFileExit_Click
End Sub
```

cmdNext Click Event:

```
Private Sub cmdNext_Click()
'Generate the next question
cmdNext.Enabled = False
Call Next_Question(CorrectAnswer)
End Sub
```

Form Activate Event:

```
Private Sub Form_Activate()
Call mnufilenew_click
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
Randomize Timer
'Load soundex function array
Wsound(1) = 0: Wsound(2) = 1: Wsound(3) = 2: Wsound(4) = 3
Wsound(5) = 0: Wsound(6) = 1: Wsound(7) = 2: Wsound(8) = 0
Wsound(9) = 0: Wsound(10) = 2: Wsound(11) = 2: Wsound(12) =
4
Wsound(13) = 5: Wsound(14) = 5: Wsound(15) = 0: Wsound(16)
= 1
Wsound(17) = 2: Wsound(18) = 6: Wsound(19) = 2: Wsound(20)
= 3
Wsound(21) = 0: Wsound(22) = 1: Wsound(23) = 0: Wsound(24)
= 2
Wsound(25) = 0: Wsound(26) = 2
'Load state/capital arrays
State(1) = "Alabama": Capital(1) = "Montgomery"
State(2) = "Alaska": Capital(2) = "Juneau"
State(3) = "Arizona": Capital(3) = "Phoenix"
State(4) = "Arkansas": Capital(4) = "Little Rock"
State(5) = "California": Capital(5) = "Sacramento"
State(6) = "Colorado": Capital(6) = "Denver"
State(7) = "Connecticut": Capital(7) = "Hartford"
State(8) = "Delaware": Capital(8) = "Dover"
State(9) = "Florida": Capital(9) = "Tallahassee"
State(10) = "Georgia": Capital(10) = "Atlanta"
State(11) = "Hawaii": Capital(11) = "Honolulu"
State(12) = "Idaho": Capital(12) = "Boise"
State(13) = "Illinois": Capital(13) = "Springfield"
State(14) = "Indiana": Capital(14) = "Indianapolis"
```

```
State(15) = "Iowa": Capital(15) = "Des Moines"
State(16) = "Kansas": Capital(16) = "Topeka"
State(17) = "Kentucky": Capital(17) = "Frankfort"
State(18) = "Louisiana": Capital(18) = "Baton Rouge"
State(19) = "Maine": Capital(19) = "Augusta"
State(20) = "Maryland": Capital(20) = "Annapolis"
State(21) = "Massachusetts": Capital(21) = "Boston"
State(22) = "Michigan": Capital(22) = "Lansing"
State(23) = "Minnesota": Capital(23) = "Saint Paul"
State(24) = "Mississippi": Capital(24) = "Jackson"
State(25) = "Missouri": Capital(25) = "Jefferson City"
State(26) = "Montana": Capital(26) = "Helena"
State(27) = "Nebraska": Capital(27) = "Lincoln"
State(28) = "Nevada": Capital(28) = "Carson City"
State(29) = "New Hampshire": Capital(29) = "Concord"
State(30) = "New Jersey": Capital(30) = "Trenton"
State(31) = "New Mexico": Capital(31) = "Santa Fe"
State(32) = "New York": Capital(32) = "Albany"
State(33) = "North Carolina": Capital(33) = "Raleigh"
State(34) = "North Dakota": Capital(34) = "Bismarck"
State(35) = "Ohio": Capital(35) = "Columbus"
State(36) = "Oklahoma": Capital(36) = "Oklahoma City"
State(37) = "Oregon": Capital(37) = "Salem"
State(38) = "Pennsylvania": Capital(38) = "Harrisburg"
State(39) = "Rhode Island": Capital(39) = "Providence"
State(40) = "South Carolina": Capital(40) = "Columbia"
State(41) = "South Dakota": Capital(41) = "Pierre"
State(42) = "Tennessee": Capital(42) = "Nashville"
State(43) = "Texas": Capital(43) = "Austin"
State(44) = "Utah": Capital(44) = "Salt Lake City"
State(45) = "Vermont": Capital(45) = "Montpelier"
State(46) = "Virginia": Capital(46) = "Richmond"
State(47) = "Washington": Capital(47) = "Olympia"
State(48) = "West Virginia": Capital(48) = "Charleston"
State(49) = "Wisconsin": Capital(49) = "Madison"
State(50) = "Wyoming": Capital(50) = "Cheyenne"
End Sub
```

lblAnswer Click Event:

```
Private Sub lblAnswer_Click(Index As Integer)
'Check multiple choice answers
Dim Iscorrect As Integer
'If already answered, exit
If cmdNext.Enabled = True Then Exit Sub
Iscorrect = 0
If mnuOptionsCapitals.Checked = True Then
  If lblAnswer(Index).Caption = Capital(CorrectAnswer) Then
Iscorrect = 1
Else
  If lblAnswer(Index).Caption = State(CorrectAnswer) Then
Iscorrect = 1
End If
Call Update_Score(Iscorrect)
End Sub
```

mnuFileExit Click Event:

```
Private Sub mnuFileExit_Click()
'End the application
End
End Sub
```

mnuFileNew Click Event:

```
Private Sub mnufilenew_click()
'Reset the score and start again
NumAns = 0
NumCorrect = 0
lblScore.Caption = "0%"
lblComment.Caption = ""
cmdNext.Enabled = False
Call Next_Question(CorrectAnswer)
End Sub
```

mnuOptionsCapitals Click Event:

```
Private Sub mnuOptionsCapitals_Click()
'Set up for providing capital, given state
mnuOptionsState.Checked = False
mnuOptionsCapitals.Checked = True
lblHeadGiven.Caption = "State:"
lblHeadAnswer.Caption = "Capital:"
Call mnufilenew_click
End Sub
```

mnuOptionsMC Click Event:

```
Private Sub mnuOptionsMC_Click()
'Set up for multiple choice answers
Dim I As Integer
mnuOptionsMC.Checked = True
mnuOptionsType.Checked = False
For I = 0 To 3
  lblAnswer(I).Visible = True
Next I
txtAnswer.Visible = False
Call mnufilenew_click
End Sub
```

mnuOptionsState Click Event:

```
Private Sub mnuOptionsState_Click()
'Set up for providing state, given capital
mnuOptionsState.Checked = True
mnuOptionsCapitals.Checked = False
lblHeadGiven.Caption = "Capital:"
lblHeadAnswer.Caption = "State:"
Call mnufilenew_click
End Sub
```

mnuOptionsType Click Event:

```
Private Sub mnuOptionsType_Click()
'Set up for type in answers
Dim I As Integer
mnuOptionsMC.Checked = False
mnuOptionsType.Checked = True
For I = 0 To 3
   lblAnswer(I).Visible = False
Next I
txtAnswer.Visible = True
Call mnufilenew_click
End Sub
```

Next_Question General Procedure:

```
Private Sub Next_Question(Answer As Integer)
Dim VUsed(50) As Integer, I As Integer, J As Integer
Dim Index(3)
lblComment.Caption = ""
NumAns = NumAns + 1
'Generate the next question based on selected options
Answer = Int(Rnd * 50) + 1
If mnuOptionsCapitals.Checked = True Then
   lblGiven.Caption = State(Answer)
Else
   lblGiven.Caption = Capital(Answer)
End If
If mnuOptionsMC.Checked = True Then
'Multiple choice answers
'Vused array is used to see which states have
'been selected as possible answers
   For I = 1 To 50
     VUsed(I) = 0
   Next I
'Pick four different state indices (J) at random
'These are used to set up multiple choice answers
'Stored in the Index array
   I = 0
   Do
     Do
       J = Int(Rnd * 50) + 1
     Loop Until VUsed(J) = 0 And J <> Answer
     VUsed(J) = 1
     Index(I) = J
     I = I + 1
```

```
   Loop Until I = 4
'Now replace one index (at random) with correct answer
   Index(Int(Rnd * 4)) = Answer
'Display multiple choice answers in label boxes
   For I = 0 To 3
     If mnuOptionsCapitals.Checked = True Then
       lblAnswer(I).Caption = Capital(Index(I))
     Else
       lblAnswer(I).Caption = State(Index(I))
     End If
   Next I
Else
'Type-in answers
   txtAnswer.Locked = False
   txtAnswer.Text = ""
   txtAnswer.SetFocus
End If
End Sub
```

txtAnswer KeyPress Event:

```
Private Sub txtAnswer_KeyPress(KeyAscii As Integer)
'Check type in answer'
Dim Iscorrect As Integer
Dim YourAnswer As String, TheAnswer As String
'Exit if already answered
If cmdNext.Enabled = True Then Exit Sub
If (KeyAscii >= vbKeyA And KeyAscii <= vbKeyZ) _
Or (KeyAscii >= vbKeyA + 32 And KeyAscii <= vbKeyZ + 32) _
Or KeyAscii = vbKeySpace Or KeyAscii = vbKeyBack Or
KeyAscii = vbKeyReturn Then
'Acceptable keystroke
  If KeyAscii <> vbKeyReturn Then Exit Sub
'Lock text box once answer entered
  txtAnswer.Locked = True
  Iscorrect = 0
'Convert response and correct answers to all upper
'case for typing problems
  YourAnswer = UCase(txtAnswer.Text)
  If mnuOptionsCapitals.Checked = True Then
    TheAnswer = UCase(Capital(CorrectAnswer))
  Else
    TheAnswer = UCase(State(CorrectAnswer))
  End If
'Check for both exact and approximate spellings
  If YourAnswer = TheAnswer Or _
  SoundEx(YourAnswer, Wsound()) = SoundEx(TheAnswer,
Wsound()) Then Iscorrect = 1
  Call Update_Score(Iscorrect)
Else
'Unacceptable keystroke
  KeyAscii = 0
End If
End Sub
```

Learn Visual Basic 6.0

6. Error-Handling, Debugging and File Input/Output

Review and Preview

- In this class, we expand on our Visual Basic knowledge from past classes and examine a few new topics. We first look at handling errors in programs, using both run-time error trapping and debugging techniques. We then study input and output to disks using sequential files and random access files.

Error Types

- No matter how hard we try, **errors** do creep into our programs. These errors can be grouped into three categories:

    1. **Syntax** errors
    2. **Run-time** errors
    3. **Logic** errors

- **Syntax errors** occur when you mistype a command or leave out an expected phrase or argument. Visual Basic detects these errors as they occur and even provides help in correcting them. You cannot run a Visual Basic program until all syntax errors have been corrected.

- **Run-time errors** are usually beyond your program's control. Examples include: when a variable takes on an unexpected value (divide by zero), when a drive door is left open, or when a file is not found. Visual Basic allows you to trap such errors and make attempts to correct them.

- **Logic errors** are the most difficult to find. With logic errors, the program will usually run, but will produce incorrect or unexpected results. The Visual Basic debugger is an aid in detecting logic errors.

- Some ways to minimize errors:

  ⇒ Design your application carefully. More design time means less debugging time.
  ⇒ Use comments where applicable to help you remember what you were trying to do.
  ⇒ Use consistent and meaningful naming conventions for your variables, objects, and procedures.

Run-Time Error Trapping and Handling

- **Run-time errors** are trappable. That is, Visual Basic recognizes an error has occurred and enables you to trap it and take corrective action. If an error occurs and is not trapped, your program will usually end in a rather unceremonious manner.

- **Error trapping** is enabled with the **On Error** statement:

  On Error GoTo *errlabel*

  Yes, this uses the dreaded **GoTo** statement! Any time a run-time error occurs following this line, program control is transferred to the line labeled ***errlabel***. Recall a labeled line is simply a line with the label followed by a colon (:).

- The best way to explain how to use error trapping is to look at an outline of an example procedure with error trapping.

  ```
  Sub SubExample()
        .
        .  [Declare variables, ...]
        .
  On Error GoTo HandleErrors
        .
        .  [Procedure code]
        .
  Exit Sub
  HandleErrors:
        .
        .  [Error handling code]
        .
  End Sub
  ```

Once you have set up the variable declarations, constant definitions, and any other procedure preliminaries, the **On Error** statement is executed to enable error trapping. Your normal procedure code follows this statement. The error handling code goes at the end of the procedure, following the **HandleErrors** statement label. This is the code that is executed if an error is encountered anywhere in the Sub procedure. Note you must exit (with **Exit Sub**) from the code before reaching the HandleErrors line to avoid inadvertent execution of the error handling code.

* Since the error handling code is in the same procedure where an error occurs, all variables in that procedure are available for possible corrective action. If at some time in your procedure, you want to **turn off** error trapping, that is done with the following statement:

   On Error GoTo 0

* Once a run-time error occurs, we would like to know what the error is and attempt to fix it. This is done in the **error handling** code.

* Visual Basic offers help in identifying run-time errors. The **Err** object returns, in its **Number** property (Err.Number), the number associated with the current error condition. (The Err function has other useful properties that we won't cover here - consult on-line help for further information.) The **Error**() function takes this error number as its argument and returns a string description of the error. Consult on-line help for Visual Basic run-time error numbers and their descriptions.

* Once an error has been trapped and some action taken, control must be returned to your application. That control is returned via the **Resume** statement. There are three options:

Resume	Lets you retry the operation that caused the error. That is, control is returned to the line where the error occurred. This could be dangerous in that, if the error has not been corrected (via code or by the user), an infinite loop between the error handler and the procedure code may result.
Resume Next	Program control is returned to the line immediately following the line where the error occurred.
Resume *label*	Program control is returned to the line labeled *label.*

• Be careful with the Resume statement. When executing the error handling portion of the code and the end of the procedure is encountered before a Resume, an error occurs. Likewise, if a Resume is encountered outside of the error handling portion of the code, an error occurs.

General Error Handling Procedure

• Development of an adequate **error handling procedure** is application dependent. You need to know what type of errors you are looking for and what corrective actions must be taken if these errors are encountered. For example, if a 'divide by zero' is found, you need to decide whether to skip the operation or do something to reset the offending denominator.

• What we develop here is a generic framework for an error handling procedure. It simply informs the user that an error has occurred, provides a description of the error, and allows the user to Abort, Retry, or Ignore. This framework is a good starting point for designing custom error handling for your applications.

• The generic code (begins with label **HandleErrors**) is:

```
HandleErrors:
Select Case MsgBox(Error(Err.Number), vbCritical + vbAbortRetryIgnore, "Error
Number" + Str(Err.Number))
  Case vbAbort
    Resume ExitLine
  Case vbRetry
    Resume
  Case vbIgnore
    Resume Next
End Select
ExitLine:
Exit Sub
```

Let's look at what goes on here. First, this routine is only executed when an error occurs. A message box is displayed, using the Visual Basic provided error description [**Error(Err.Number)**] as the message, uses a **critical icon** along with the **Abort**, **Retry**, and **Ignore** buttons, and uses the error number [**Err.Number**] as the title. This message box returns a response indicating which button was selected by the user. If Abort is selected, we simply exit the procedure. (This is done using a **Resume** to the line labeled **ExitLine**. Recall all error trapping must be terminated with a Resume statement of some kind.) If Retry is selected, the offending program line is retried (in a real application, you or the user would have to change something here to correct the condition causing the error). If Ignore is selected, program operation continues with the line following the error causing line.

- To use this generic code in an existing procedure, you need to do three things:

  1. Copy and paste the error handling code into the end of your procedure.
  2. Place an **Exit Sub** line immediately preceding the **HandleErrors** labeled line.
  3. Place the line, **On Error GoTo HandleErrors** , at the beginning of your procedure.

  For example, if your procedure is the **SubExample** seen earlier, the modified code will look like this:

  ```
  Sub SubExample()
          .
          .  [Declare variables, ...]
          .
  On Error GoTo HandleErrors
          .
          .  [Procedure code]
          .
  Exit Sub
  HandleErrors:
  Select Case MsgBox(Error(Err.Number), vbCritical + vbAbortRetryIgnore, "Error
  Number" + Str(Err.Number))
    Case vbAbort
      Resume ExitLine
    Case vbRetry
      Resume
    Case vbIgnore
      Resume Next
  End Select
  ExitLine:
  Exit Sub
  End Sub
  ```

Again, this is a very basic error-handling routine. You must determine its utility in your applications and make any modifications necessary. Specifically, you need code to clear error conditions before using the Retry option.

- One last thing. Once you've written an error handling routine, you need to test it to make sure it works properly. But, creating run-time errors is sometimes difficult and perhaps dangerous. Visual Basic comes to the rescue! The Visual Basic **Err** object has a method (**Raise**) associated with it that simulates the occurrence of a run-time error. To cause an error with value **Number**, use:

Err.Raise Number

- We can use this function to completely test the operation of any error handler we write. Don't forget to remove the Raise statement once testing is completed, though! And, to really get fancy, you can also use **Raise** to generate your own 'application-defined' errors. There are errors specific to your application that you want to trap.

- To clear an error condition (any error, not just ones generated with the Raise method), use the method **Clear**:

    Err.Clear

Example 6-1

Simple Error Trapping

1. Start a new project. Add a text box and a command button.

2. Set the properties of the form and each control:

Form1:
BorderStyle	1-Fixed Single
Caption	Error Generator
Name	frmError

Command1:
Caption	Generate Error
Default	True
Name	cmdGenError

Text1:
Name	txtError
Text	[Blank]

The form should look something like this:
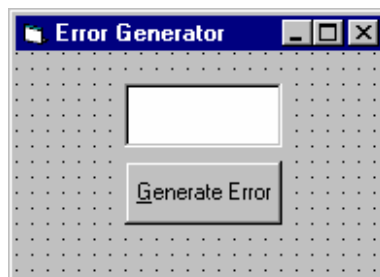
3. Attach this code to the **cmdGenError_Click** event.

```
Private Sub cmdGenError_Click()
On Error GoTo HandleErrors
Err.Raise Val(txtError.Text)
Err.Clear
Exit Sub
HandleErrors:
Select Case MsgBox(Error(Err.Number), vbCritical +
  vbAbortRetryIgnore, "Error Number" + Str(Err.Number))
  Case vbAbort
    Resume ExitLine
  Case vbRetry
    Resume
  Case vbIgnore
    Resume Next
End Select
ExitLine:
Exit Sub
End Sub
```

In this code, we simply generate an error using the number input in the text box. The generic error handler then displays a message box which you can respond to in one of three ways.

4. Save your application. Try it out using some of these typical error numbers (or use numbers found with on-line help). Notice how program control changes depending on which button is clicked.

Error Number	Error Description
6	Overflow
9	Subscript out of range
11	Division by zero
13	Type mismatch
16	Expression too complex
20	Resume without error
52	Bad file name or number
53	File not found
55	File already open
61	Disk full
70	Permission denied
92	For loop not initialized

Debugging Visual Basic Programs

* We now consider the search for, and elimination of, **logic errors**. These are errors that don't prevent an application from running, but cause incorrect or unexpected results. Visual Basic provides an excellent set of **debugging** tools to aid in this search.

* Debugging a code is an art, not a science. There are no prescribed processes that you can follow to eliminate all logic errors in your program. The usual approach is to eliminate them as they are discovered.

* What we'll do here is present the debugging tools available in the Visual Basic environment (several of which appear as buttons on the toolbar) and describe their use with an example. You, as the program designer, should select the debugging approach and tools you feel most comfortable with.

* The interface between your application and the debugging tools is via three different debug windows: the **Immediate Window**, the **Locals Window**, and the **Watch Window**. These windows can be accessed from the **View** menu (the Immediate Window can be accessed by pressing **Ctrl+G**). Or, they can be selected from the **Debug Toolbar** (accessed using the **Toolbars** option under the **View** menu):

*
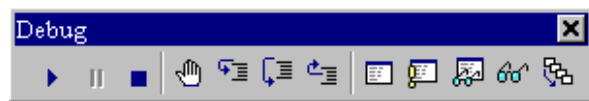
* All debugging using the debug windows is done when your application is in **break mode**. You can enter break mode by setting breakpoints, pressing **Ctrl+Break**, or the program will go into break mode if it encounters an untrapped error or a **Stop** statement.

* Once in break mode, the debug windows and other tools can be used to:

    $\Rightarrow$ Determine values of variables
    $\Rightarrow$ Set breakpoints
    $\Rightarrow$ Set watch variables and expressions
    $\Rightarrow$ Manually control the application
    $\Rightarrow$ Determine which procedures have been called
    $\Rightarrow$ Change the values of variables and properties

Example 6-2

Debugging Example

1. Unlike other examples, we'll do this one as a group. It will be used to demonstrate use of the debugging tools.

2. The example simply has a form with a single command button. The button is used to execute some code. We won't be real careful about proper naming conventions and such in this example.
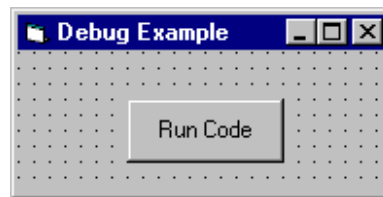
3. The code attached to this button's **Click** event is a simple loop that evaluates a function at several values.

```
Private Sub Command1_Click()
Dim X As Integer, Y As Integer
X = 0
Do
Y = Fcn(X)
X = X + 1
Loop While X <= 20
End Sub
```

This code begins with an X value of 0 and computes the Y value using the general integer function **Fcn**. It then increments X by 1 and repeats the Loop. It continues looping While X is less than or equal to 20. The function Fcn is computed using:

```
Function Fcn(X As Integer) As Integer
Fcn = CInt(0.1 * X ^ 2)
End Function
```

Admittedly, this code doesn't do much, especially without any output, but it makes a good example for looking at debugger use. Set up the application and get ready to try debugging.

Using the Debugging Tools

- There are several **debugging tools** available for use in Visual Basic. Access to these tools is provided with both menu options and buttons on the Debug toolbar. These tools include breakpoints, watch points, calls, step into, step over, and step out.

- The simplest tool is the use of direct prints to the immediate window.

- Printing to the Immediate Window:

  You can print directly to the **immediate window** while an application is running. Sometimes, this is all the debugging you may need. A few carefully placed print statements can sometimes clear up all logic errors, especially in small applications.

  To print to the immediate window, use the **Print** method:

  > Debug.Print [List of variables separated by commas or semi-colons]

- Debug.Print Example:

  1. Place the following statement in the **Command1_Click** procedure after the line calling the general procedure Fcn:

     `Debug.Print X; Y`

  and run the application.

  2. Examine the immediate window. Note how, at each iteration of the loop, the program prints the value of X and Y. You could use this information to make sure X is incrementing correctly and that Y values look acceptable.

  3. Remove the Debug.Print statement.

• Breakpoints:

In the above examples, the program ran to completion before we could look at the debug window. In many applications, we want to stop the application while it is running, examine variables and then continue running. This can be done with **breakpoints**.

A breakpoint is a line in the code where you want to stop (temporarily) the execution of the program, that is force the program into break mode. To set a breakpoint, put the cursor in the line of code you want to break on. Then, press **<F9>** or click the **Breakpoint** button on the toolbar or select **Toggle Breakpoint** from the **Debug** menu. The line will be highlighted.

When you run your program, Visual Basic will stop when it reaches lines with breakpoints and allow you to use the immediate window to check variables and expressions. To continue program operation after a breakpoint, press **<F5>**, click the **Run** button on the toolbar, or choose **Start** from the **Run** menu.

You can also change variable values using the immediate window. Simply type a valid Basic expression. This can sometimes be dangerous, though, as it may change program operation completely.

• Breakpoint Example:

1. Set a breakpoint on the **X = X + 1** line in the sample program. Run the program.

2. When the program stops, display the immediate window and type the following line:

Print X;Y

3. The values of these two variables will appear in the debug window. You can use a question mark (**?**) as shorthand for the command **Print**, if you'd like. Restart the application. Print the new variable values.

4. Try other breakpoints if you have time. Once done, all breakpoints can be cleared by **Ctrl**+**Shift**+**<F9>** or by choosing **Clear All Breakpoints** from the **Debug** menu. Individual breakpoints can be toggled using **<F9>** or the **Breakpoint** button on the toolbar.

• Viewing Variables in the Locals Window:

The **locals window** shows the value of any variables within the scope of the current procedure. As execution switches from procedure to procedure, the contents of this window changes to reflect only the variables applicable to the current procedure. Repeat the above example and notice the values of X and Y also appear in the locals window.

• Watch Expressions:

The **Add Watch** option on the **Debug** menu allows you to establish **watch expressions** for your application. Watch expressions can be variable values or logical expressions you want to view or test. Values of watch expressions are displayed in the **watch window**.

In break mode, you can use the **Quick Watch** button on the toolbar to add watch expressions you need. Simply put the cursor on the variable or expression you want to add to the watch list and click the Quick Watch button.

Watch expressions can be edited using the **Edit Watch** option on the **Debug** menu.

• Watch Expression Example:

1. Set a breakpoint at the **X = X + 1** line in the example.

2. Set a watch expression for the variable X. Run the application. Notice X appears in the watch window. Every time you re-start the application, the value of X changes.

3. At some point in the debug procedure, add a quick watch on Y. Notice it is now in the watch window.

4. Clear the breakpoint. Add a watch on the expression: X = Y. Set **Watch Type** to '**Break When Value Is True**.' Run the application. Notice it goes into break mode and displays the watch window whenever X = Y. Delete this last watch expression.

- Call Stack:

While at a breakpoint, you may execute your program one line at a time by pressing

Selecting the **Call Stack** button from the toolbar (or pressing **Ctrl**+**L** or selecting **Call Stack** from the **View** menu) will display all active procedures, that is those that have not been exited.

Call Stack helps you unravel situations with nested procedure calls to give you some idea of where you are in the application.

- Call Stack Example:

  1. Set a breakpoint on the **Fcn = Cint()** line in the general function procedure. Run the application. It will break at this line.

  2. Press the **Call Stack** button. It will indicate you are currently in the **Fcn** procedure which was called from the **Command1_Click** procedure. Clear the breakpoint.

- Single Stepping (Step Into):

While at a breakpoint, you may execute your program one line at a time by pressing **<F8>**, choosing the **Step Into** option in the **Debug** menu, or by clicking the **Step Into** button on the toolbar.

This process is **single stepping**. It allows you to watch how variables change (in the locals window) or how your form changes, one step at a time.

You may step through several lines at a time by using **Run To Cursor** option. With this option, click on a line below your current point of execution. Then press **Ctrl**+**<F8>** (or choose **Run To Cursor** in the **Debug** menu). the program will run through every line up to the cursor location, then stop.

- Step Into Example:

  1. Set a breakpoint on the **Do** line in the example. Run the application.

  2. When the program breaks, use the **Step Into** button to single step through the program.

  3. At some point, put the cursor on the **Loop While** line. Try the **Run To Cursor** option (press **Ctrl**+**<F8>**).

- Procedure Stepping (Step Over):

  While single stepping your program, if you come to a procedure call you know functions properly, you can perform **procedure stepping**. This simply executes the entire procedure at once, rather than one step at a time.

  To move through a procedure in this manner, press **Shift**+**<F8>**, choose **Step Over** from the **Debug** menu, or press the **Step Over** button on the toolbar.

- Step Over Example:

  1. Run the previous example. Single step through it a couple of times.

  2. One time through, when you are at the line calling the **Fcn** function, press the **Step Over** button. Notice how the program did not single step through the function as it did previously.

- Function Exit (Step Out):

  While stepping through your program, if you wish to complete the execution of a **function** you are in, without stepping through it line-by-line, choose the **Step Out** option. The function will be completed and you will be returned to the procedure accessing that function.

  To perform this step out, press **Ctrl**+**Shift**+**<F8>**, choose **Step Out** from the **Debug** menu, or press the **Step Out** button on the toolbar. Try this on the previous example.

Debugging Strategies

- We've looked at each debugging tool briefly. Be aware this is a cursory introduction. Use the on-line help to delve into the details of each tool described. Only through lots of use and practice can you become a proficient debugger. There are some guidelines to doing a good job, though.

- My first suggestion is: keep it **simple**. Many times, you only have one or two bad lines of code. And you, knowing your code best, can usually quickly narrow down the areas with bad lines. Don't set up some elaborate debugging procedure if you haven't tried a simple approach to find your error(s) first. Many times, just a few intelligently-placed **Debug.Print** statements or a few examinations of the immediate and locals windows can solve your problem.

- A tried and true approach to debugging can be called **Divide and Conquer**. If you're not sure where your error is, guess somewhere in the middle of your application code. Set a breakpoint there. If the error hasn't shown up by then, you know it's in the second half of your code. If it has shown up, it's in the first half. Repeat this division process until you've narrowed your search.

- And, of course, the best debugging strategy is to be careful when you first design and write your application to minimize searching for errors later.

Sequential Files

- In many applications, it is helpful to have the capability to read and write information to a disk file. This information could be some computed data or perhaps information loaded into a Visual Basic object.

- Visual Basic supports two primary file formats: sequential and random access. We first look at **sequential files**.

- A sequential file is a line-by-line list of data. You can view a sequential file with any text editor. When using sequential files, you must know the order in which information was written to the file to allow proper reading of the file.

- Sequential files can handle both text data and variable values. Sequential access is best when dealing with files that have lines with mixed information of different lengths. I use them to transfer data between applications.

Sequential File Output (Variables)

- We first look at **writing** values of **variables** to sequential files. The first step is to **Open** a file to write information to. The syntax for opening a sequential file for output is:

      Open SeqFileName For Output As #N

  where **SeqFileName** is the name of the file to open and **N** is an integer file number. The filename must be a complete path to the file.

- When done writing to the file, **Close** it using:

  Close N

  Once a file is closed, it is saved on the disk under the path and filename used to open the file.

- Information is written to a sequential file one line at a time. Each line of output requires a separate Basic statement.

- There are two ways to write variables to a sequential file. The first uses the **Write** statement:

  Write #N, [variable list]

  where the variable list has variable names delimited by commas. (If the variable list is omitted, a blank line is printed to the file.) This statement will write one line of information to the file, that line containing the variables specified in the variable list. The variables will be delimited by commas and any string variables will be enclosed in quotes. This is a good format for exporting files to other applications like Excel.

  Example

      Dim A As Integer, B As String, C As Single, D As Integer
          .
          .
      Open TestOut For Output As #1
      Write #1, A, B, C
      Write #1, D
      Close 1

  After this code runs, the file **TestOut** will have two lines. The first will have the variables A, B, and C, delimited by commas, with B (a string variable) in quotes. The second line will simply have the value of the variable D.

- The second way to write variables to a sequential file is with the **Print** statement:

  Print #N, [variable list]

  This statement will write one line of information to the file, that line containing the variables specified in the variable list. (If the variable list is omitted, a blank line will be printed.) If the variables in the list are separated with semicolons (;), they are printed with a single space between them in the file. If separated by commas (,), they are spaced in wide columns. Be careful using the Print statement with string variables. The Print statement does not enclose string variables in quotes, hence, when you read such a variable back in, Visual Basic may have trouble knowing where a string ends and begins. It's good practice to 'tack on' quotes to string variables when using Print.

Example

Dim A As Integer, B As String, C As Single, D As Integer

.
.

Open TestOut For Output As #1
Print #1, A; Chr(34) + B + Chr(34), C
Print #1, D
Close 1

After this code runs, the file **TestOut** will have two lines. The first will have the variables A, B, and C, delimited by spaces. B will be enclosed by quotes [Chr(34)]. The second line will simply have the value of the variable D.

Quick Example: Writing Variables to Sequential Files

1. Start a new project.

2. Attach the following code to the **Form_Load** procedure. This code simply writes a few variables to sequential files.

```
Private Sub Form_Load()
Dim A As Integer, B As String, C As Single, D As Integer
A = 5
B = "Visual Basic"
C = 2.15
D = -20
Open "Test1.Txt" For Output As #1
Open "Test2.Txt" For Output As #2
Write #1, A, B, C
Write #1, D
Print #2, A, B, C
Print #2, D
Close 1
Close 2
End Sub
```

3. Run the program. Use a text editor (try the Windows 95 **Notepad**) to examine the contents of the two files, **Test1.Txt** and **Test2.Txt**. They are probably in the Visual Basic main directory. Note the difference in the two files, especially how the variables are delimited and the fact that the string variable is not enclosed in quotes in Test2.Txt. Save the application, if you want to.

Sequential File Input (Variables)

- To **read variables** from a sequential file, we essentially reverse the write procedure. First, open the file using:

  Open SeqFileName For Input As #N

where **N** is an integer file number and **SeqFileName** is a complete file path. The file is closed using:

  Close N

- The **Input** statement is used to read in variables from a sequential file. The format is:

  Input #N, [variable list]

The variable names in the list are separated by commas. If no variables are listed, the current line in the file N is skipped.

- Note variables must be read in exactly the same manner as they were written. So, using our previous example with the variables A, B, C, and D, the appropriate statements are:

  Input #1, A, B, C
  Input #1, D

  These two lines read the variables A, B, and C from the first line in the file and D from the second line. It doesn't matter whether the data was originally written to the file using **Write** or **Print** (i.e. commas are ignored).

Quick Example: Reading Variables from Sequential Files

1. Start a new project or simply modify the previous quick example.

2. Attach the following code to the **Form_Load** procedure. This code reads in files created in the last quick example.

```
Private Sub Form_Load()
Dim A As Integer, B As String, C As Single, D As Integer
Open "Test1.Txt" For Input As #1
Input #1, A, B, C
Debug.Print "A="; A
Debug.Print "B="; B
Debug.Print "C="; C
Input #1, D
Debug.Print "D="; D
Close 1
End Sub
```

Note the **Debug.Print** statements and how you can add some identifiers (in quotes) for printed information.

3. Run the program. Look in the debug window and note the variable values. Save the application, if you want to.

4. Rerun the program using **Test2.Txt** as in the input file. What differences do you see? Do you see the problem with using Print and string variables? Because of this problem, I almost always use **Write** (instead of **Print**) for saving variable information to files. Edit the Test2.Txt file (in Notepad), putting quotes around the words **Visual Basic**. Rerun the program using this file as input - it should work fine now.

Writing and Reading Text Using Sequential Files

- In many applications, we would like to be able to save text information and retrieve it for later reference. This information could be a **text file** created by an application or the contents of a Visual Basic **text box**.

- Writing Text Files:

To **write** a sequential text file, we follow the simple procedure: open the file, write the file, close the file. If the file is a line-by-line text file, each line of the file is written to disk using a single **Print** statement:

Print #N, Line

where **Line** is the current line (a text string). This statement should be in a loop that encompasses all lines of the file. You must know the number of lines in your file, beforehand.

If we want to write the contents of the **Text** property of a text box named **txtExample** to a file, we use:

Print #N, txtExample.Text

Example

We have a text box named **txtExample**. We want to save the contents of the Text property of that box in a file named **MyText.ned** on the c: drive in the \MyFiles directory. The code to do this is:

Open "c:\MyFiles\MyText.ned" For Output As #1
Print #1, txtExample.Text
Close 1

The text is now saved in the file for later retrieval.

- Reading Text Files:

To read the contents of a previously-saved text file, we follow similar steps to the writing process: open the file, read the file, close the file. If the file is a text file, we read each individual line with the **Line Input** command:

Line Input #1, Line

This line is usually placed in a **Do/Loop** structure that is repeated untill all lines of the file are read in. The **EOF**() function can be used to detect an end-of-file condition, if you don't know, a prioiri, how many lines are in the file.

To place the contents of a file opened with number N into the **Text** property of a text box named **txtExample** we use the **Input** function:

txtExample.Text = Input(LOF(N), N)

This **Input** function has two arguments: **LOF**(N), the length of the file opened as N and **N**, the file number.

Example

We have a file named **MyText.ned** stored on the c: drive in the \MyFiles directory. We want to read that text file into the text property of a text box named **txtExample**. The code to do this is:

```
Open "c:\MyFiles\MyText.ned" For Input As #1
txtExample.Text = Input(LOF(1), 1)
Close 1
```

The text in the file will now be displayed in the text box.

Random Access Files

- Note that to access a particular data item in a sequential file, you need to read in all items in the file prior to the item of interest. This works acceptably well for small data files of unstructured data, but for large, structured files, this process is time-consuming and wasteful. Sometimes, we need to access data in nonsequential ways. Files which allow nonsequential access are **random access files**.

- To allow nonsequential access to information, a random access file has a very definite structure. A random access file is made up of a number of **records**, each record having the same length (measured in bytes). Hence, by knowing the length of each record, we can easily determine (or the computer can) where each record begins. The first record in a random access file is Record **1**, **not 0** as used in Visual Basic arrays. Each record is usually a set of variables, of different types, describing some item. The structure of a random access file is:

```
┌─────────────────┐
│    Record 1     │
│    N bytes      │
├─────────────────┤
│    Record 2     │
│    N bytes      │
├─────────────────┤
│    Record 3     │
│    N bytes      │
└─────────────────┘
        .
        .
        .
┌─────────────────┐
│   Record Last   │
│    N bytes      │
└─────────────────┘
```
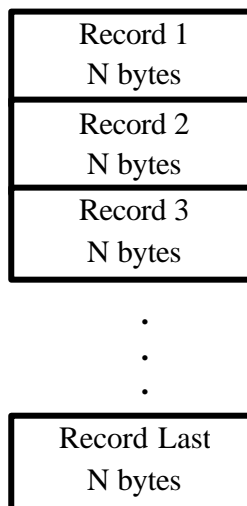
- A good analogy to illustrate the differences between sequential files and random access files are cassette music tapes and compact discs. To hear a song on a tape (a sequential device), you must go past all songs prior to your selection. To hear a song on a CD (a random access device), you simply go directly to the desired selection. One difference here though is we require all of our random access records to be the same length - not a good choice on CD's!

- To write and read random access files, we must know the **record length** in **bytes**. Some variable types and their length in bytes are:

Type	Length (Bytes)
Integer	2
Long	4
Single	4
Double	8
String	1 byte per character

So, for every variable that is in a file's record, we need to add up the individual variable length's to obtain the total record length. To ease this task, we introduce the idea of user-defined variables.

User-Defined Variables

- Data used with random access files is most often stored in **user-defined variables**. These data types group variables of different types into one assembly with a single, user-defined type associated with the group. Such types significantly simplify the use of random access files.

- The Visual Basic keyword **Type** signals the beginning of a user-defined type declaration and the words **End Type** signal the end. An example best illustrates establishing a user-defined variable. Say we want to use a variable that describes people by their name, their city, their height, and their weight. We would define a variable of **Type Person** as follows:

```
Type Person
    Name As String
    City As String
    Height As Integer
    Weight As Integer
End Type
```

These variable declarations go in the same code areas as normal variable declarations, depending on desired scope. At this point, we have not reserved any storage for the data. We have simply described to Visual Basic the layout of the data.

- To create variables with this newly defined type, we employ the usual **Dim** statement. For our **Person** example, we would use:

      Dim Lou As Person
      Dim John As Person
      Dim Mary As Person

  And now, we have three variables, each containing all the components of the variable type **Person**. To refer to a single component within a user-defined type, we use the dot-notation:

      VarName.Component

  As an example, to obtain Lou's **Age**, we use:

      Dim AgeValue as Integer
        .
        .
      AgeValue = Lou.Age

  Note the similarity to dot-notation we've been using to set properties of various Visual Basic tools.

Writing and Reading Random Access Files

- We look at **writing** and **reading random access files** using a user-defined variable. For other variable types, refer to Visual Basic on-line help. To open a random access file named **RanFileName**, use:

      Open RanFileName For Random As #N Len = RecordLength

  where **N** is an available file number and **RecordLength** is the length of each record. Note you don't have to specify an input or output mode. With random access files, as long as they're open, you can write or read to them.

- To **close** a random access file, use:

      Close N

- As mentioned previously, the record length is the sum of the lengths of all variables that make up a record. A problem arises with **String** type variables. You don't know their lengths ahead of time. To solve this problem, Visual Basic lets you declare fixed lengths for strings. This allows you to determine record length. If we have a string variable named **StrExample** we want to limit to **14** characters, we use the declaration:

      Dim StrExample As String * 14

  Recall each character in a string uses 1 byte, so the length of such a variable is 14 bytes.

- Recall our example user-defined variable type, **Person**. Let's revisit it, now with restricted string lengths:

      Type Person
          Name As String * 40
          City As String * 35
          Height As Integer
          Weight As Integer
      End Type

  The record length for this variable type is 79 bytes (40 + 35 +2 + 2). To open a file named **PersonData** as File **#1**, with such records, we would use the statement:

      Open PersonData For Random As #1 Len = 79

- The **Get** and **Put** statements are used to read from and write to random access files, respectively. These statements read or write one record at a time. The syntax for these statements is simple:

      Get #N, [RecordNumber], *variable*

      Put #N, [RecordNumber], *variable*

  The **Get** statement **reads** from the file and stores data in the *variable*, whereas the **Put** statement **writes** the contents of the specified *variable* to the file. In each case, you can optionally specifiy the record number. If you do not specify a record number, the next sequential position is used.

- The *variable* argument in the Get and Put statements is usually a single user-defined variable. Once read in, you obtain the component parts of this variable using **dot-notation**. Prior to writing a user-defined variable to a random access file, you 'load' the component parts using the same dot-notation.

- There's a lot more to using random access files; we've only looked at the basics. Refer to your Visual Basic documentation and on-line help for further information. In particular, you need to do a little cute programming when deleting records from a random access file or when 'resorting' records.

Using the Open and Save Common Dialog Boxes

- Note to both write and read sequential and random access files, we need a file name for the **Open** statement. To ensure accuracy and completeness, it is suggested that common dialog boxes (briefly studied in Class 4) be used to get this file name information from the user. I'll provide you with a couple of code segments that do just that. Both segments assume you have a **common dialog box** on your form named **cdlFiles**, with the **CancelError** property set equal to **True**. With this property True, an error is generated by Visual Basic when the user presses the **Cancel** button in the dialog box. By trapping this error, it allows an elegant exit from the dialog box when canceling the operation is desired.

- The code segment to obtain a file name (**MyFileName** with default extension **Ext**) for opening a file to **read** is:

      Dim MyFileName As String, Ext As String
          .
          .
      cdlFiles.Filter = "Files (*." + Ext + ")|*." + Ext
      cdlFiles.DefaultExt = Ext
      cdlFiles.DialogTitle = "Open File"
      cdlFiles.Flags = cdlOFNFileMustExist + cdlOFNPathMustExist
      On Error GoTo No_Open
      cdlFiles.ShowOpen
      MyFileName = cdlFiles.filename
          .
          .
      Exit Sub
      No_Open:
      Resume ExitLIne
      ExitLine:
      Exit Sub
      End Sub

  A few words on what's going on here. First, some properties are set such that only files with **Ext** (a three letter string variable) extensions are displayed (**Filter** property), the default extension is Ext (**DefaultExt** property), the title bar is set (**DialogTitle** property), and some **Flags** are set to insure the file and path exist (see Appendix II for more common dialog flags). Error trapping is enabled to trap the **Cancel** button. Finally, the common dialog box is displayed and the filename property returns with the desired name. That name is put in the string variable **MyFileName**. What you do after obtaining the file name depends on what type of file you are dealing with. For sequential files, you would open the file, read in the information, and close the file. For random access files, we just open the file here. Reading and writing to/from the file would be handled elsewhere in your coding.

• The code segment to retrieve a file name (**MyFileName**) for **writing** a file is:

```
Dim MyFileName As String, Ext As String
        .
        .
cdlFiles.Filter = "Files (*." + Ext + ")|*." + Ext
cdlFiles.DefaultExt = Ext
cdlFiles.DialogTitle = "Save File"
cdlFiles.Flags = cdlOFNOverwritePrompt + cdlOFNPathMustExist
On Error GoTo No_Save
cdlFiles.ShowSave
MyFileName = cdlFiles.filename
        .
        .
Exit Sub
No_Save:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

Note this code is essentially the same used for an Open file name. The **Flags** property differs slightly. The user is prompted if a previously saved file is selected for overwrite. After obtaining a valid file name for a sequential file, we would open the file for output, write the file, and close it. For a random access file, things are trickier. If we want to save the file with the same name we opened it with, we simply close the file. If the name is different, we must open a file (using a different number) with the new name, write the complete random access file, then close it. Like I said, it's trickier.

• We use both of these code segments in the final example where we write and read sequential files.

Example 6-3

Note Editor - Reading and Saving Text Files

1. We now add the capability to read in and save the contents of the text box in the Note Editor application from last class. Load that application. Add a common dialog box to your form. Name it **cdlFiles** and set the **CancelError** property to **True**.

2. Modify the File menu (use the Menu Editor and the **Insert** button) in your application, such that Open and Save options are included. The File menu should now read:

    File
        New
        Open
        Save
          ———
        Exit

    Properties for these new menu items should be:

Caption	Name	Shortcut
&Open	mnuFileOpen	[None]
&Save	mnuFileSave	[None]

3. The two new menu options need code. Attach this code to the **mnuFileOpen_Click**
    event. This uses a modified ve rsion of the code segment seen previously. We assign
    the extension **ned** to our note editor files.

```
Private Sub mnuFileOpen_Click()
cdlFiles.Filter = "Files (*.ned)|*.ned"
cdlFiles.DefaultExt = "ned"
cdlFiles.DialogTitle = "Open File"
cdlFiles.Flags = cdlOFNFileMustExist + cdlOFNPathMustExist
On Error GoTo No_Open
cdlFiles.ShowOpen
Open cdlFiles.filename For Input As #1
txtEdit.Text = Input(LOF(1), 1)
Close 1
Exit Sub
No_Open:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

And for the **mnuFileSave_Click** procedure, use this code. Much of this can be copied
    from the previous procedure.

```
Private Sub mnuFileSave_Click()
cdlFiles.Filter = "Files (*.ned)|*.ned"
cdlFiles.DefaultExt = "ned"
cdlFiles.DialogTitle = "Save File"
cdlFiles.Flags = cdlOFNOverwritePrompt +
   cdlOFNPathMustExist
On Error GoTo No_Save
cdlFiles.ShowSave
Open cdlFiles.filename For Output As #1
Print #1, txtEdit.Text
Close 1
Exit Sub
No_Save:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

Each of these procedures is similar. The dialog box is opened and, if a filename is returned, the file is read/written. If **Cancel** is pressed, no action is taken. These routines can be used as templates for file operations in other applications.

4. Save your application. Run it and test the **Open** and **Save** functions. Note you have to save a file before you can open one. Check for proper operation of the **Cancel** button in the common dialog box.

5. If you have the time, there is one major improvement that should be made to this application. Notice that, as written, only the text information is saved, not the formatting (bold, italic, underline, size). Whenever a file is opened, the text is displayed based on current settings. It would be nice to save formatting information along with the text. This can be done, but it involves a fair amount of reprogramming. Suggested steps:

A. Add lines to the **mnuFileSave_Click** routine that write the text box properties **FontBold**, **FontItalic**, **FontUnderline**, and **FontSize** to a separate sequential file. If your text file is named TxtFile.ned, I would suggest naming the formatting file TxtFile.fmt. Use string functions to put this name together. That is, chop the **ned** extension off the text file name and tack on the **fmt** extension. You'll need the **Len**() and **Left**() functions.

B. Add lines to the **mnuFileOpen_Click** routine that read the text box properties FontBold, FontItalic, FontUnderline, and FontSize from your format sequential file. You'll need to define some intermediate variables here because Visual Basic won't allow you to read properties directly from a file. You'll also need logic to set/reset any check marks in the menu structure to correspond to these input properties.

C. Add lines to the **mnuFileNew_Click** procedure that, when the user wants a new file, reset the text box properties FontBold, FontItalic, FontUnderline, and FontSize to their default values and set/reset the corresponding menu check marks.

D. Try out the modified application. Make sure every new option works as it should.

Actually, there are 'custom' tools (we'll look at custom tools in Class 10) that do what we are trying to do with this modification, that is save text box contents with formatting information. Such files are called 'rich text files' or **rtf** files. You may have seen these before when transferring files from one word processor to another.

6. Another thing you could try: Modify the message box that appears when you try to **Exit**. Make it ask if you wish to save your file before exiting - provide **Yes**, **No**, **Cancel** buttons. Program the code corresponding to each possible response. Use calls to existing procedures, if possible.

Exercise 6-1

Information Tracking

Design and develop an application that allows the user to enter (on a daily basis) some piece of information that is to be saved for future review and reference. Examples could be stock price, weight, or high temperature for the day. The input screen should display the current date and an input box for the desired information. all values should be saved on disk for future retrieval and update. A scroll bar should be available for reviewing all previously-stored values.

My Solution:

Form:
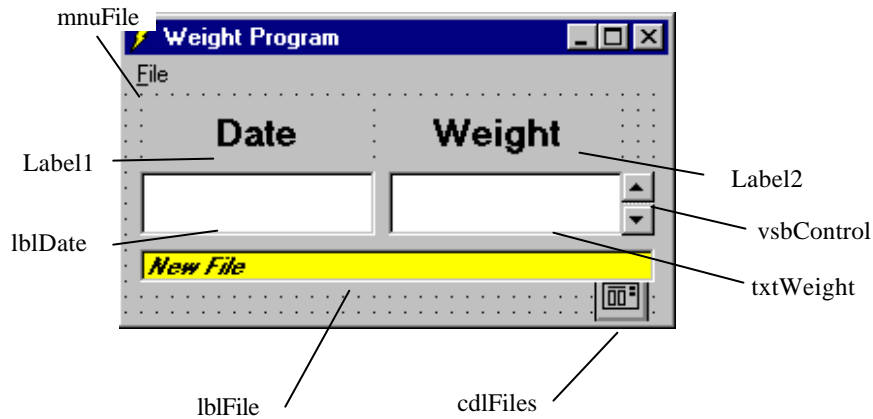
Properties:

Form **frmWeight**:
>   BorderStyle = 1 - Fixed Single
>   Caption = Weight Program

VScrollBar **vsbControl**:
>   Min = 1
>   Value = 1

TextBox **txtWeight**:
> Alignment = 2 - Center
> FontName = MS Sans Serif
> FontSize = 13.5

Label **lblFile**:
> BackColor = &H0000FFFF& (White)
> BorderStyle = 1 - Fixed Single
> Caption = New File
> FontName = MS Sans Serif
> FontBold = True
> FontItalic = True
> FontSize = 8.25

Label **lblDate**:
> Alignment = 2 - Center
> BackColor = &H00FFFFFF& (White)
> BorderStyle = 1 - Fixed Single
> FontName = MS Sans Serif
> FontSize = 13.5

Label **Label2**:
> Alignment = 2 - Center
> Caption = Weight
> FontName = MS Sans Serif
> FontSize = 13.5
> FontBold = True

Label **Label1**:
> Alignment = 2 - Center
> Caption = Date
> FontName = MS Sans Serif
> FontSize = 13.5
> FontBold = True

CommonDialog **cdlFiles**:
> CancelError = True

Menu **mnuFile**:
> Caption = &File

Menu **mnuFileNew**:
> Caption = &New

Menu **mnuFileOpen**:
      Caption = &Open

Menu **mnuFileSave**:
      Caption = &Save

Menu **mnuLine**:
      Caption = -

Menu **mnuFileExit**:
      Caption = E&xit

Code:

General Declarations:

```
Option Explicit
Dim Dates(1000) As Date
Dim Weights(1000) As String
Dim NumWts As Integer
```

Init General Procedure:

```
Sub Init()
NumWts = 1:  vsbControl.Value = 1: vsbControl.Max = 1
Dates(1) = Format(Now, "mm/dd/yy")
Weights(1) = ""
lblDate.Caption = Dates(1)
txtWeight.Text = Weights(1)
lblFile.Caption = "New File"
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
frmWeight.Show
Call Init
End Sub
```

mnufileExit Click Event:

```
Private Sub mnuFileExit_Click()
'Make sure user really wants to exit
Dim Response As Integer
Response = MsgBox("Are you sure you want to exit the weight
program?", vbYesNo + vbCritical + vbDefaultButton2, "Exit
Editor")
If Response = vbNo Then
  Exit Sub
Else
  End
End If
End Sub
```

mnuFileNew Click Event:

```
Private Sub mnuFileNew_Click()
'User wants new file
Dim Response As Integer
Response = MsgBox("Are you sure you want to start a new
file?", vbYesNo + vbQuestion, "New File")
If Response = vbNo Then
  Exit Sub
Else
  Call Init
End If
End Sub
```

mnuFileOpen Click Event:

```
Private Sub mnuFileOpen_Click()
Dim I As Integer
Dim Today As Date
Dim Response As Integer
Response = MsgBox("Are you sure you want to open a new
file?", vbYesNo + vbQuestion, "New File")
If Response = vbNo Then Exit Sub
cdlFiles.Filter = "Files (*.wgt)|*.wgt"
cdlFiles.DefaultExt = "wgt"
cdlFiles.DialogTitle = "Open File"
cdlFiles.Flags = cdlOFNFileMustExist + cdlOFNPathMustExist
On Error GoTo No_Open
cdlFiles.ShowOpen
Open cdlFiles.filename For Input As #1
```

```
lblFile.Caption = cdlFiles.filename
Input #1, NumWts
For I = 1 To NumWts
  Input #1, Dates(I), Weights(I)
Next I
Close 1
Today = Format(Now, "mm/dd/yy")
If Today <> Dates(NumWts) Then
  NumWts = NumWts + 1
  Dates(NumWts) = Today
  Weights(NumWts) = ""
End If
vsbControl.Max = NumWts
vsbControl.Value = NumWts
lblDate.Caption = Dates(NumWts)
txtWeight.Text = Weights(NumWts)
Exit Sub
No_Open:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

mnuFileSave Click Event:

```
Private Sub mnuFileSave_Click()
Dim I As Integer
cdlFiles.Filter = "Files (*.wgt)|*.wgt"
cdlFiles.DefaultExt = "wgt"
cdlFiles.DialogTitle = "Save File"
cdlFiles.Flags = cdlOFNOverwritePrompt +
cdlOFNPathMustExist
On Error GoTo No_Save
cdlFiles.ShowSave
Open cdlFiles.filename For Output As #1
lblFile.Caption = cdlFiles.filename
Write #1, NumWts
For I = 1 To NumWts
  Write #1, Dates(I), Weights(I)
Next I
Close 1
Exit Sub
No_Save:
Resume ExitLine
ExitLine:
Exit Sub
```

```
End Sub
```

txtWeight Change Event:

```
Private Sub txtWeight_Change()
Weights(vsbControl.Value) = txtWeight.Text
End Sub
```

txtWeight KeyPress Event:

```
Private Sub txtWeight_KeyPress(KeyAscii As Integer)
If KeyAscii >= vbKey0 And KeyAscii <= vbKey9 Then
  Exit Sub
Else
  KeyAscii = 0
End If
End Sub
```

vsbControl Change Event:

```
Private Sub vsbControl_Change()
lblDate.Caption = Dates(vsbControl.Value)
txtWeight.Text = Weights(vsbControl.Value)
txtWeight.SetFocus
End Sub
```

Exercise 6-2

'Recent Files' Menu Option

Under the File menu on nearly every application (that opens files) is a list of the four most recently-used files (usually right above the Exit option). Modify your information tracker to implement such a feature. This is not trivial -- there are lots of things to consider. For example, you'll need a file to store the last four file names. You need to open that file and initialize the corresponding menu entries when you run the application -- you need to rewrite that file when you exit the application. You need logic to re-order file names when a new file is opened or saved. You need logic to establish new menu items as new files are used. You'll need additional error-trapping in the open procedure, in case a file selected from the menu no longer exists. Like I said, a lot to consider here.

My Solution:

These new menu items immediately precede the existing **Exit** menu item:

Menu **mnuFileRecent**:
        Caption = [Blank]
        Index = 0, 1, 2, 3 (a control array)
        Visible = False

Menu **mnuFileBar**:
        Caption = -
        Visible = False

Code Modifications (new code is bold and italicized):

General Declarations:

```
Option Explicit
Dim Dates(1000) As Date
Dim Weights(1000) As String
Dim NumWts As Integer
Dim NFiles As Integer, RFile(3) As String, MenuOpen As
Integer, FNmenu As String
```

Rfile Update General Procedure:

```
Sub RFile_Update(NewFile As String)
'Routine to place newest file name in proper order
'in menu structure
Dim I As Integer, J As Integer, InList As Integer
'Convert name to all upper case letters
NewFile = UCase(NewFile)
'See if file is already in list
InList = 0
For I = 0 To NFiles - 1
  If RFile(I) = NewFile Then InList = 1: Exit For
Next I
'If file not in list, increment number of items with
'a maximum of 4.  Then, move others down, then place
'new name at top of list
If InList = 0 Then
  NFiles = NFiles + 1
  If NFiles > 4 Then
    NFiles = 4
  Else
    If NFiles = 1 Then mnuFileBar.Visible = True
    mnuFileRecent(NFiles - 1).Visible = True
  End If
  If NFiles <> 1 Then
    For I = NFiles - 1 To 1 Step -1
      RFile(I) = RFile(I - 1)
    Next I
  End If
  RFile(0) = NewFile
Else
'If file already in list, put name at top and shift
'others accordingly
  If I <> 0 Then
    For J = I - 1 To 0 Step -1
      RFile(J + 1) = RFile(J)
    Next J
    RFile(0) = NewFile
  End If
End If
'Set menu captions according to new list
For I = 0 To NFiles - 1
  mnuFileRecent(I).Caption = "&" + Format(I + 1, "# ") +
RFile(I)
Next I
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
Dim I As Integer
'Open .ini file and load in recent file names
Open "weight.ini" For Input As #1
NFiles = 0: MenuOpen = 0
For I = 0 To 3
  Input #1, RFile(I)
  If RFile(I) <> "" Then
    NFiles = NFiles + 1
    mnuFileBar.Visible = True
    mnuFileRecent(I).Caption = "&" + Format(I + 1, "# ") +
RFile(I)
    mnuFileRecent(I).Visible = True
  End If
Next I
Close 1
frmWeight.Show
Call Init
End Sub
```

mnuFileExit Click Event:

```
Private Sub mnuFileExit_Click()
'Make sure user really wants to exit
Dim Response As Integer, I As Integer
Response = MsgBox("Are you sure you want to exit the weight
program?", vbYesNo + vbCritical + vbDefaultButton2, "Exit
Editor")
If Response = vbNo Then
  Exit Sub
Else
  'Write out .ini file when done
  Open "weight.ini" For Output As #1
  For I = 0 To 3
    Write #1, RFile(I)
  Next I
  Close 1
  End
End If
End Sub
```

mnuFileOpen Click Event:

```
Private Sub mnuFileOpen_Click()
Dim I As Integer
Dim Today As Date
Dim Response As Integer
Dim File_To_Open As String
Response = MsgBox("Are you sure you want to open a new
file?", vbYesNo + vbQuestion, "New File")
If Response = vbNo Then Exit Sub
If MenuOpen = 0 Then
  cdlFiles.Filter = "Files (*.wgt)|*.wgt"
  cdlFiles.DefaultExt = "wgt"
  cdlFiles.DialogTitle = "Open File"
  cdlFiles.Flags = cdlOFNFileMustExist +
cdlOFNPathMustExist
  On Error GoTo No_Open
  cdlFiles.ShowOpen
  File_To_Open = cdlFiles.filename
Else
  File_To_Open = FNmenu
End If
MenuOpen = 0
On Error GoTo BadOpen
Open File_To_Open For Input As #1
lblFile.Caption = File_To_Open
Input #1, NumWts
For I = 1 To NumWts
  Input #1, Dates(I), Weights(I)
Next I
Close 1
Call RFile_Update(File_To_Open)
Today = Format(Now, "mm/dd/yy")
If Today <> Dates(NumWts) Then
  NumWts = NumWts + 1
  Dates(NumWts) = Today
  Weights(NumWts) = ""
End If
vsbControl.Max = NumWts
vsbControl.Value = NumWts
lblDate.Caption = Dates(NumWts)
txtWeight.Text = Weights(NumWts)
Exit Sub
No_Open:
Resume ExitLine
ExitLine:
Exit Sub
```

```
BadOpen:
Select Case MsgBox(Error(Err.Number), vbCritical +
vbRetryCancel, "File Open Error")
Case vbRetry
  Resume
Case vbCancel
  Resume No_Open
End Select
End Sub
```

mnuFileRecent Click Event:

```
Private Sub mnuFileRecent_Click(Index As Integer)
  FNmenu = RFile(Index): MenuOpen = 1
  Call mnuFileOpen_Click
End Sub
```

mnuFileSave Click Event:

```
Private Sub mnuFileSave_Click()
Dim I As Integer
cdlFiles.Filter = "Files (*.wgt)|*.wgt"
cdlFiles.DefaultExt = "wgt"
cdlFiles.DialogTitle = "Save File"
cdlFiles.Flags = cdlOFNOverwritePrompt +
cdlOFNPathMustExist
On Error GoTo No_Save
cdlFiles.ShowSave
Open cdlFiles.filename For Output As #1
lblFile.Caption = cdlFiles.filename
Write #1, NumWts
For I = 1 To NumWts
  Write #1, Dates(I), Weights(I)
Next I
Close 1
Call RFile_Update(cdlFiles.filename)
Exit Sub
No_Save:
Resume ExitLine
ExitLine:
Exit Sub
End Sub
```

This page intentionally not left blank.

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**:
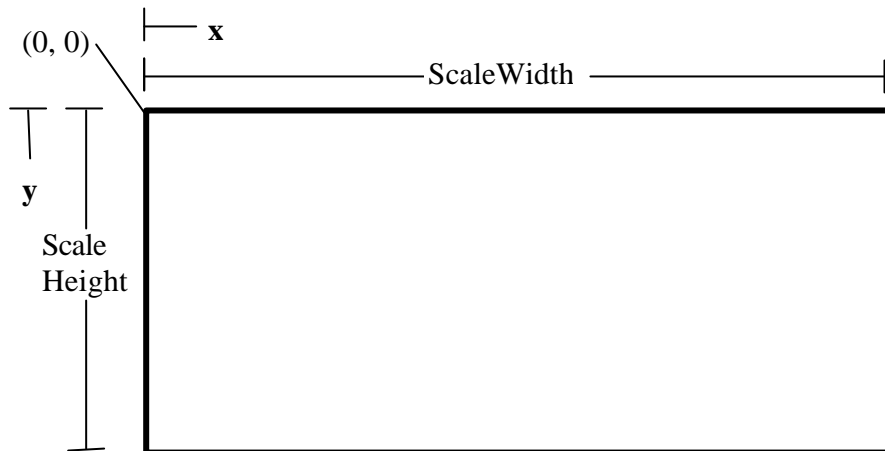
Note the **x** (horizontal) coordinate runs from left to right, starting at **0** and extending to **ScaleWidth - 1**. The **y** (vertical) coordinate goes from top to bottom, starting at **0** and ending at **ScaleHeight - 1**. Points in this coordinate system will always be referred to by a Cartesian 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:
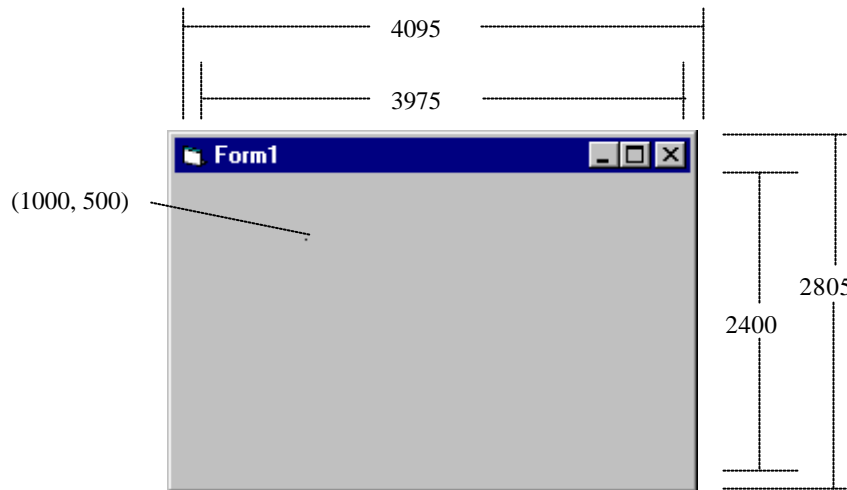
The marked point (in color **ForeColor,** black in this case) is pointed to by the Cartesian coordinate (1000, 500) - this marking, of course, does not appear on the form. If you want to try this example, and the other graphic methods, put the code in the Form_Click event. Run the project and click on 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:
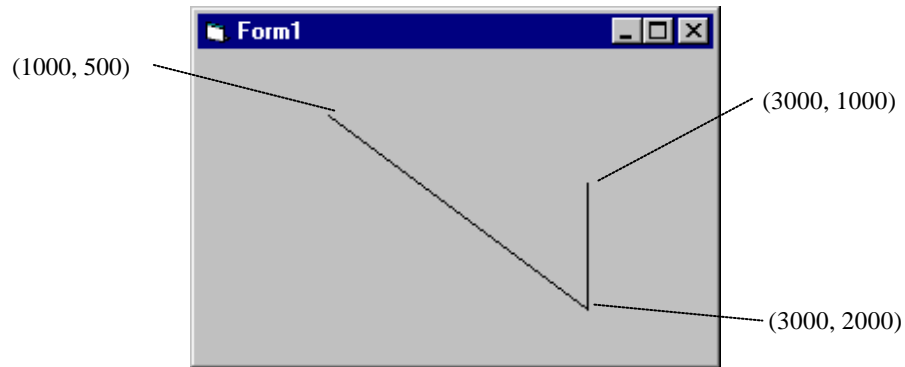
The command:

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

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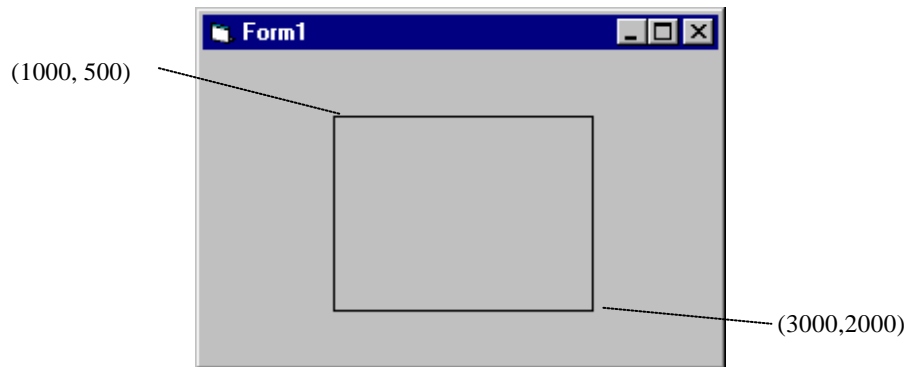
- Circle Method:

  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:
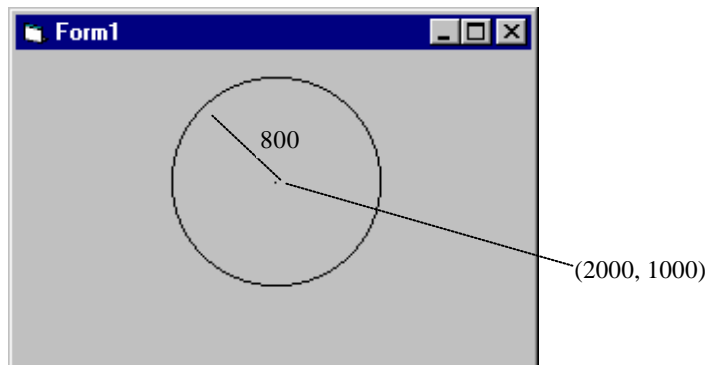
- Print Method:

  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):
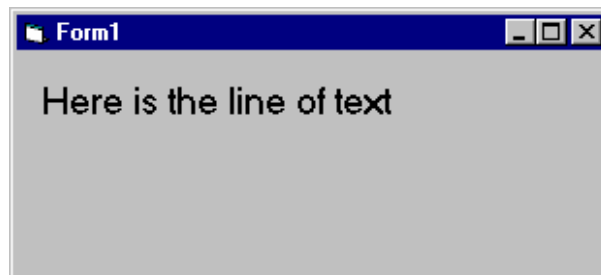
- Cls Method:

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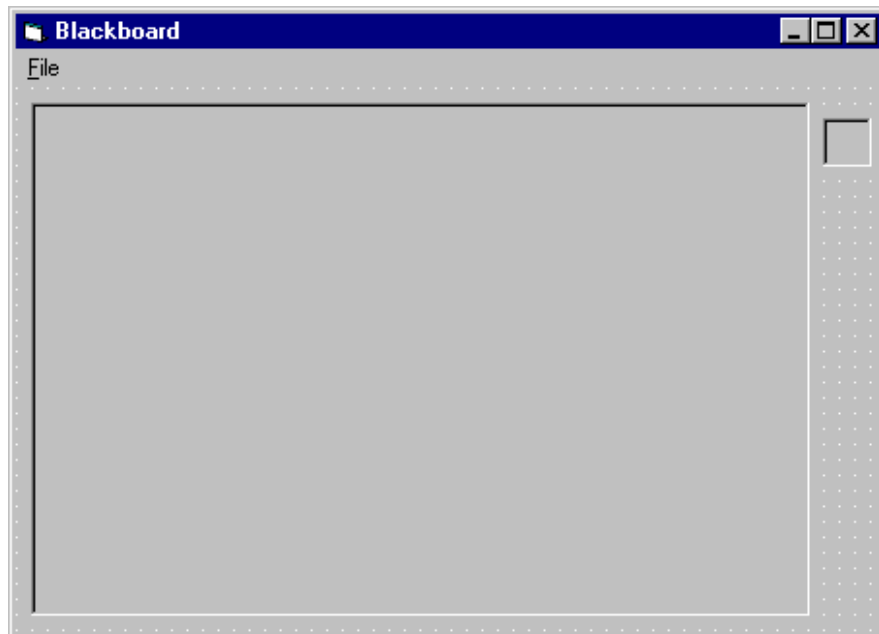
4. Now, copy and paste the label box (create a control array named **lblColor**) until there are eight boxes on the form, lined up vertically under the original box. When done, the form will look just as above, except there will be eight label boxes.

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:
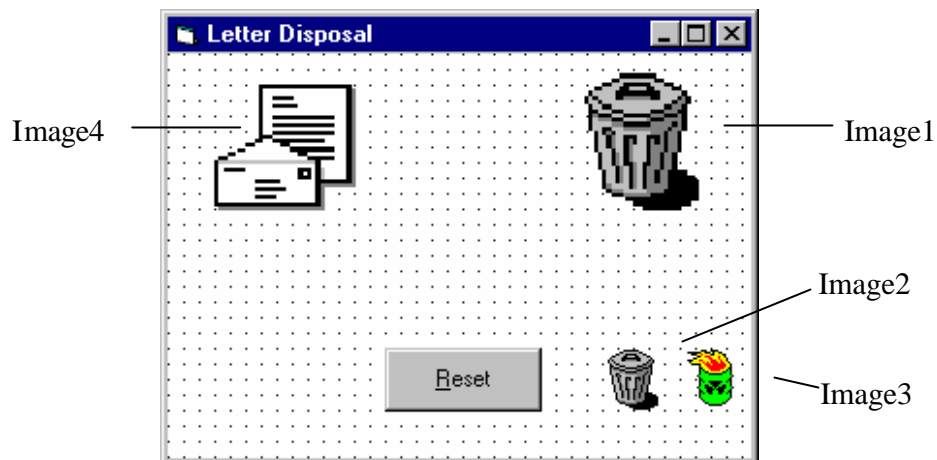
Some explanation about the images on this form is needed. The letter image is the control to be dragged and the trash can (at **Image1** location) is where it will be dragged to. The additional images (the other trash can and burning can) are not visible at run-time and are used to change the state of 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:
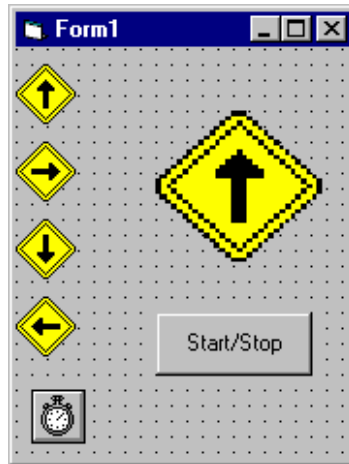
2. Attach this code to the **Command1_Click** procedure.

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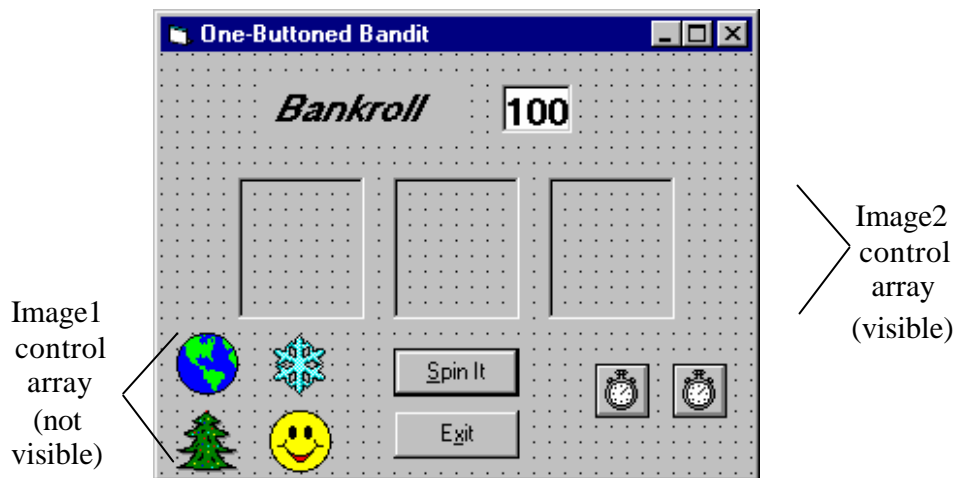
A few words on what we're doing. We will randomly fill the three large image boxes by choosing from the four choices in the non-visible image boxes. One timer (**timSpin**) will be used to flash pictures in the boxes. One timer (**timDone**) will be used to time the entire process.

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

(2,7) (5,11) (6,13) (0,2)
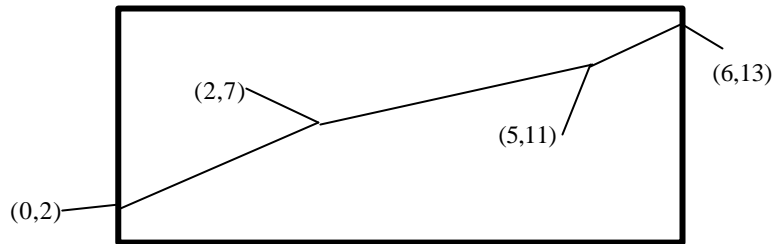
To draw such a plot, for each point, we would need to scale each (x, y) pair to fit within the dimensions of the form specified by the **ScaleWidth** and **ScaleHeight** properties. This is a straightforward, but tedious computation.

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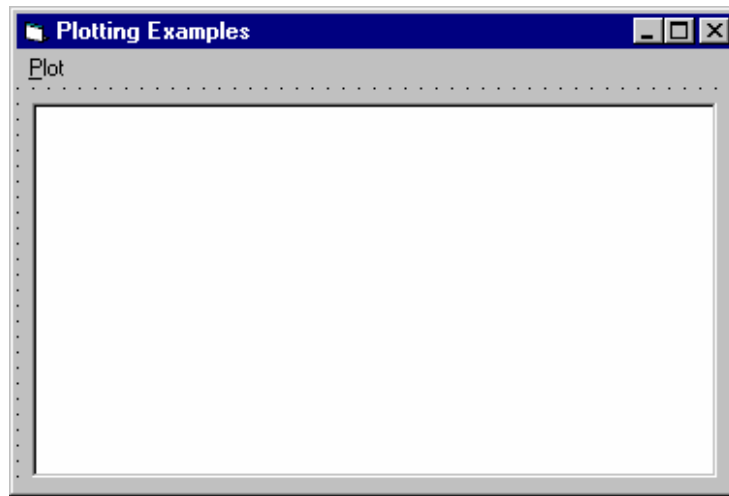
3. Place this code in the **general declarations** area. This makes the x and y arrays and the number of points global.

```
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:

lblResults

lblWinnings

imgSuit
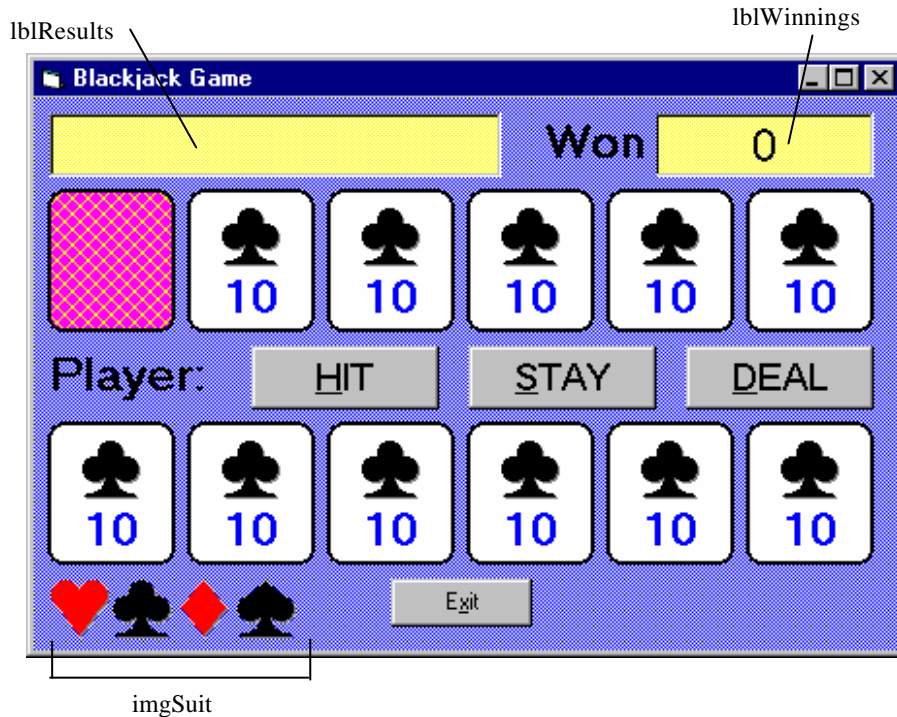
There are so many things here, I won't label them all. The button names are obvious.
The definition of the cards is not so obvious. Each card is made up of three different
objects (each a control array). The card itself is a shape (**shpDealer** for dealer cards,
shpPlayer for player cards), the number 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):
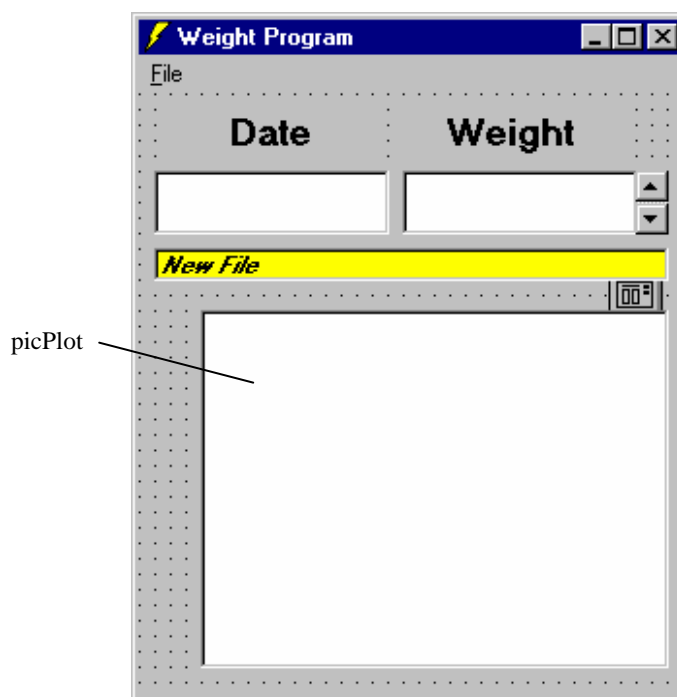
New Properties:

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.

Learn Visual Basic 6.0

8. Database Access and Management

Review and Preview

- In past classes, we've seen the power of the built-in Visual Basic tools. In this class, we look at one of the more powerful tools, the Data Control. Using this tool, in conjunction with associated 'data-aware' tools, allows us to access and manage databases. We only introduce the ideas of database access and management - these topics alone could easily take up a ten week course.

- A major change in Visual Basic, with the introduction of Version 6.0, is in its database management tools. New tools based on ActiveX Data Object (ADO) technology have been developed. These new tools will eventually replace the older database tools, called DAO (Data Access Object) tools. We will only discuss the ADO tools. Microsoft still includes the DAO tools for backward compatibility. You might want to study these on your own, if desired.

Database Structure and Terminology

- In simplest terms, a **database** is a collection of information. This collection is stored in well-defined **tables**, or matrices.

- The **rows** in a database table are used to describe similar items. The rows are referred to as database **records**. In general, no two rows in a database table will be alike.

- The **columns** in a database table provide characteristics of the records. These characteristics are called database **fields**. Each field contains one specific piece of information. In defining a database field, you specify the data type, assign a length, and describe other attributes.

- Here is a simple database example:

Field

ID No	Name	Date of Birth	Height	Weight
1	Bob Jones	01/04/58	72	170
2	Mary Rodgers	11/22/61	65	125
3	Sue Williams	06/11/57	68	130

Record

Table

In this database **table**, each **record** represents a single individual. The **fields** (descriptors of the individuals) include an identification number (ID No), Name, Date of Birth, Height, and Weight.

- Most databases use **indexes** to allow faster access to the information in the database. Indexes are sorted lists that point to a particular row in a table. In the example just seen, the **ID No** field could be used as an index.

- A database using a single table is called a **flat database**. Most databases are made up of many tables. When using multiple tables within a database, these tables must have some common fields to allow cross-referencing of the tables. The referral of one table to another via a common field is called a **relation** Such groupings of tables are called **relational databases**.

- In our first example, we will use a sample database that comes with Visual Basic. This database (**BIBLIO.MDB**) is found in the main Visual Basic directory (try c:\Program Files\Microsoft Visual Studio\VB98). It is a database of books about computers. Let's look at its relational structure. The BIBLIO.MDB database is made up of four tables:

Authors Table (6246 Records, 3 Fields)

Au_ID	Author	Year Born

Publishers Table (727 Records, 10 Fields)

PubID	Name	Company		Fax	Comments

Title Author Table (16056 Records, 2 Fields)

ISBN	Au_ID

Titles Table (8569 Records, 8 Fields)

Title	Year Pub	ISBN	PubID		Comments

The **Authors** table consists of author identification numbers, the author's name, and the year born. The **Publishers** table has information regarding book publishers. Some of the fields include an identification number, the publisher name, and pertinent phone numbers. The **Title Author** table correlates a book's ISBN (a universal number assigned to books) with an author's identification number. And, the **Titles** table has several fields describing each individual book, including title, ISBN, and publisher identification.

Note each table has two types of information: **source** data and **relational** data. Source data is actual information, such as titles and author names. Relational data are references to data in other tables, such as Au_ID and PubID. In the Authors, Publishers and Title Author tables, the first column is used as the table **index**. In the Titles table, the ISBN value is the **index**.

• Using the relational data in the four tables, we should be able to obtain a complete description of any book title in the database. Let's look at one example:

Titles   **Publishers**

Title		ISBN	PubID		PubID	Publisher
Step-by-step dBase IV		0-0280095-2-5	52		52	McGraw-Hill

Title Author   **Authors**

ISBN	Au_ID		Au_ID	Author
0-0280095-2-5	171		171	Wraye, Toby

Here, the book in the **Titles** table, entitled "Step-by-step dBase IV," has an ISBN of 0-0280095-2-5 and a PubID of 52. Taking the PubID into the **Publishers** table, determines the book is published by McGraw-Hill and also allows us to access all other information concerning the publisher. Using the ISBN in the **Title Author** table provides us with the author identification (Au_ID) of 171, which, when used in the **Authors** table, tells us the book's author is Toby Wraye.

- We can form alternate tables from a database's inherent tables. Such **virtual tables**, or **logical views**, are made using queries of the database. A **query** is simply a request for information from the database tables. As an example with the BIBLIO.MDB database, using pre-defined query languages, we could 'ask' the database to form a table of all authors and books published after 1992, or provide all author names starting with B. We'll look briefly at queries.

- Keeping track of all the information in a database is handled by a **database management system** (DBMS). They are used to create and maintain databases. Examples of commercial DBMS programs are Microsoft Access, Microsoft FoxPro, Borland Paradox, Borland dBase, and Claris FileMaker. We can also use Visual Basic to develop a DBMS. Visual Basic shares the same 'engine' used by Microsoft **Access**, known as the **Jet** engine. In this class, we will see how to use Visual Basic to access data, display data, and perform some elementary management operations.

ADO Data Control

- The **ADO** (ActiveX Data Object) **data control** is the primary interface between a Visual Basic application and a database. It can be used without writing any code at all! Or, it can be a central part of a complex database management system. This icon may not appear in your Visual Basic toolbox. If it doesn't, select **Project** from the main menu, then click **Components**. The Components window will appear. Select **Microsoft ADO Data Control**, then click **OK**. The control will be added to your toolbox.

- As mentioned in Review and Preview, previous versions of Visual Basic used another data control. That control is still included with Visual Basic 6.0 (for backward compatibility) and has as its icon:

Make sure you are <u>not</u> using this data control for the work in this class. This control is suitable for small databases. You might like to study it on your own.

- The data control (or tool) can access databases created by several other programs besides Visual Basic (or Microsoft Access). Some other formats supported include Btrieve, dBase, FoxPro, and Paradox databases.

- The data control can be used to perform the following tasks:

    1. Connect to a database.
    2. Open a specified database table.
    3. Create a virtual table based on a database query.
    4. Pass database fields to other Visual Basic tools, for display or editing.
       Such tools are **bound** tools (controls), or data aware.
    5. Add new records or update a database.
    6. Trap any errors that may occur while accessing data.
    7. Close the database.

- Data Control Properties:

Align	Determines where data control is displayed.
Caption	Phrase displayed on the data control.
ConnectionString	Contains the information used to establish a connection to a database.
LockType	Indicates the type of locks placed on records during editing (default setting makes databases read-only).
Recordset	A set of records defined by a data control's ConnectionString and RecordSource properties. Run-time only.
RecordSource	Determines the table (or virtual table) the data control is attached to.

- As a rule, you need one data control for every database table, or virtual table, you need access to. One row of a table is accessible to each data control at any one time. This is referred to as the **current record**.

- When a data control is placed on a form, it appears with the assigned caption and four arrow buttons:

The arrows are used to navigate through the table rows (records). As indicated, the buttons can be used to move to the beginning of the table, the end of the table, or from record to record.

Data Links

- After placing a data control on a form, you set the **ConnectionString** property. The ADO data control can connect to a variety of database types. There are three ways to connect to a database: using a data link, using an ODBC data source, or using a connection string. In this class, we will look only at connection to a Microsoft Access database using a **data link**. A data link is a file with a **UDL** extension that contains information on database type.

- If your database does not have a data link, you need to create one. This process is best illustrated by example. We will be using the BIBLIO.MDB database in our first example, so these steps show you how to create its data link:

  1. Open Windows **Explorer**.
  2. Open the folder where you will store your data link file.
  3. Right-click the right side of Explorer and choose **New**. From the list of files, select **Microsoft Data Link**.
  4. Rename the newly created file **BIBLIO.UDL**
  5. Right-click this new UDL file and click **Properties**.
  6. Choose the **Provider** tab and select **Microsoft Jet 3.51 OLE DB Provider** (an Access database).
  7. Click the **Next** button to go to the **Connection** tab.
  8. Click the ellipsis and use the **Select Access Database** dialog box to choose the **BIBLIO.MDB** file which is in the Visual Basic main folder. Click **Open**.
  9. Click **Test Connection**. Then, click **OK** (assuming it passed). The UDL file is now created and can be assigned to **ConnectionString**, using the steps below.

- If a data link has been created and exists for your database, click the ellipsis that appears next to the **ConnectionString** property. Choose **Use Data Link File**. Then, click **Browse** and find the file. Click **Open**. The data link is now assigned to the property. Click **OK**.

Assigning Tables

- Once the ADO data control is connected to a database, we need to assign a table to that control. Recall each data control is attached to a single table, whether it is a table inherent to the database or the virtual table we discussed. Assigning a table is done via the **RecordSource** property.

- Tables are assigned by making queries of the database. The language used to make a query is **SQL** (pronounced 'sequel,' meaning structured query language). SQL is an English-like language that has evolved into the most widely used database query language. You use SQL to formulate a question to ask of the database. The data base 'answers' that question with a new table of records and fields that match your criteria.

- A table is assigned by placing a valid SQL statement in the **RecordSource** property of a data control. We won't be learning any SQL here. There are many texts on the subject - in fact, many of them are in the BIBLIO.MDB database we've been using. Here we simply show you how to use SQL to have the data control 'point' to an inherent database table.

- Click on the ellipsis next to **RecordSource** in the property box. A **Property Pages** dialog box will appear. In the box marked **Command Text** (**SQL**), type this line:

<div align="center">

SELECT * FROM TableName

</div>

  This will select all fields (the * is a wildcard) from a table named **TableName** in the database. Click **OK**.

- Setting the **RecordSource** property also establishes the **Recordset** property, which we will see later is a very important property.

- In summary, the relationship between the **data control** and its two primary properties (**ConnectionString** and **RecordSource**) is:
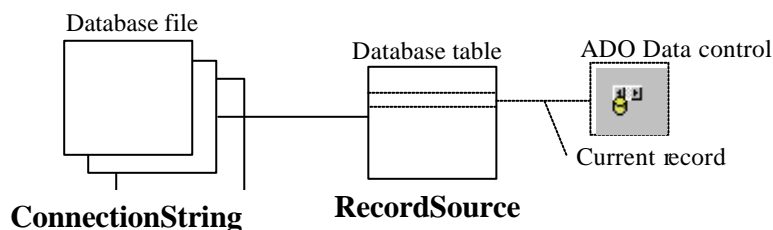
Bound Data Tools

- Most of the Visual Basic tools we've studied can be used as **bound**, or **data-aware**, tools (or controls). That means, certain tool properties can be tied to a particular database field. To use a bound control, one or more data controls must be on the form.

- Some bound data tools are:

Label	Can be used to provide display-only access to a specified text data field.
Text Box	Can be used to provide read/write access to a specified text data field. Probably, the most widely used data bound tool.
Check Box	Used to provide read/write access to a Boolean field.
Combo Box	Can be used to provide read/write access to a text data field.
List Box	Can be used to provide read/write access to a text data field.
Picture Box	Used to display a graphical image from a bitmap, icon, or metafile on your form. Provides read/write access to a image/binary data field.
Image Box	Used to display a graphical image from a bitmap, icon, or metafile on your form (uses fewer resources than a picture box). Provides read/write access to a image/binary data field.

- There are also three 'custom' data aware tools, the **DataCombo** (better than using the bound combo box), **DataList** (better than the bound list box), and **DataGrid** tools, we will look at later.

- Bound Tool Properties:

DataChanged	Indicates whether a value displayed in a bound control has changed.
DataField	Specifies the name of a field in the table pointed to by the respective data control.
DataSource	Specifies which data control the control is bound to.

If the data in a data-aware control is changed and then the user changes focus to another control or tool, the database will automatically be updated with the new data (assuming LockType is set to allow an update).

- To make using bound controls easy, follow these steps (in order listed) in placing the controls on a form:

    1. Draw the bound control on the same form as the data control to which it will be bound.
    2. Set the **DataSource** property. Click on the drop-down arrow to list the data controls on your form. Choose one.
    3. Set the **DataField** property. Click on the drop-down arrow to list the fields associated with the selected data control records. Make your choice.
    4. Set all other properties, as required.

    By following these steps in order, we avoid potential data access errors.

- The relationships between the bound data control and the data control are:
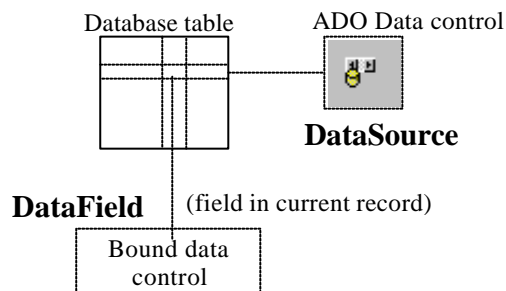
Example 8-1

Accessing the Books Database

1. Start a new application. We'll develop a form where we can skim through the books database, examining titles and ISBN values. Place an ADO data control, two label boxes, and two text boxes on the form.

2. If you haven't done so, create a data link for the BIBLIO.MDB database following the steps given under Data Links in these notes.

3. Set the following properties for each control. For the data control and the two text boxes, make sure you set the properties in the order given.

Form1:

BorderStyle	1-Fixed Single
Caption	Books Database
Name	frmBooks

Adodc1:

Caption	Book Titles
ConnectionString	BIBLIO.UDL (in whatever folder you saved it in - select, don't type)
RecordSource	SELECT * FROM Titles
Name	dtaTitles

Label1:

Caption	Title

Label2:

Caption	ISBN

Text1:

DataSource	dtaTitles (select, don't type)
DataField	Title (select, don't type)
Locked	True
MultiLine	True
Name	txtTitle
Text	[Blank]

Text2:

DataSource   dtaTitles (select, don't type)
DataField   ISBN (select, don't type)
        Locked   True
Name   txtISBN
Text   [Blank]

When done, the form will look something like this (try to space your controls as shown; we'll use all the blank space as we continue with this example):
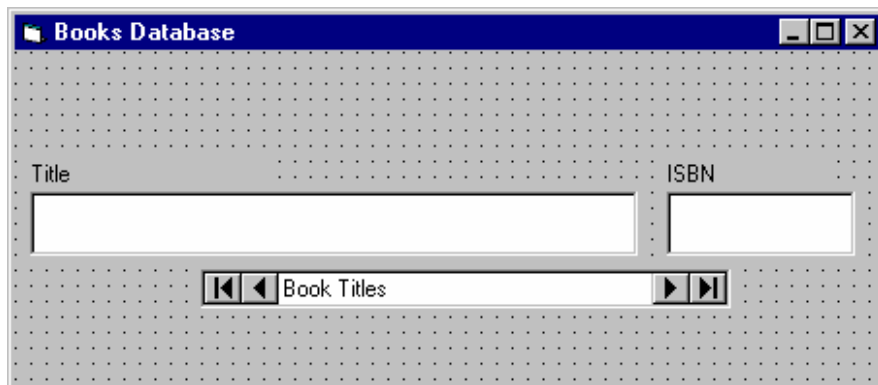
4. Save the application. Run the application. Cycle through the various book titles using the data control. Did you notice something? You didn't have to write one line of Visual Basic code! This indicates the power behind the data tool and bound tools.

Creating a Virtual Table

- Many times, a database table has more information than we want to display. Or, perhaps a table does not have all the information we want to display. For instance, in Example 8-1, seeing the Title and ISBN of a book is not real informative - we would also like to see the Author, but that information is not provided by the Titles table. In these cases, we can build our own **virtual table**, displaying only the information we want the user to see.

- We need to form a different SQL statement in the RecordSource property. Again, we won't be learning SQL here. We will just give you the proper statement.

Quick Example: Forming a Virtual Table

1. We'll use the results of Example 8-1 to add the **Author** name to the form. Replace the **RecordSource** property of the **dtaTitles** control with the following SQL statement:

   SELECT Author,Titles.ISBN,Title FROM Authors,[Title Author],Titles
       WHERE Authors.Au_ID=[Title Author].Au_ID AND Titles.ISBN=[Title
       Author].ISBN ORDER BY Author

This must be typed as a single line in the Command Text (SQL) area that appears when you click the ellipsis by the RecordSource property. Make sure it is typed in exactly as shown. Make sure there are spaces after 'SELECT', after 'Author,Titles.ISBN,Title', after 'FROM', after 'Authors,[Title Author],Titles', after 'WHERE', after 'Authors.Au_ID=[Title Author].Au_ID', after 'AND', after 'Titles.ISBN=[Title Author].ISBN', and separating the final three words 'ORDER BY Author'. The program will tell you if you have a syntax error in the SQL statement, but will give you little or no help in telling you what's wrong.

Here's what this statement does: It **selects** the Author, Titles.ISBN, and Title fields **from** the Authors, Title Author, and Titles tables, **where** the respective Au_ID and ISBN fields match. It then **orders** the resulting virtual table, using authors as an index.

2. Add a label box and text box to the form, for displaying the author name. Set the control properties.

> **Label3:**
> Caption   Author
>
> **Text1:**
> DataSource   dtaTitles (select, don't type)
> DataField   Author (select, don't type)
>   Locked   True
> Name   txtAuthor
> Text   [Blank]

When done, the form should resemble this:
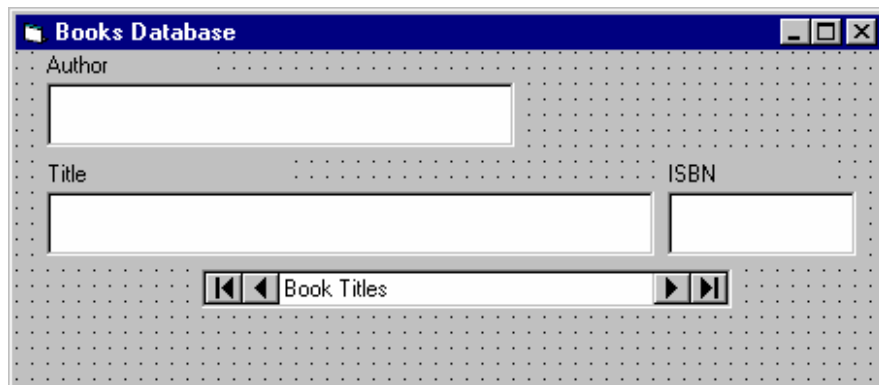
3. Save, then rerun the application. The author's names will now appear with the book titles and ISBN values. Did you notice you still haven't written any code? I know you had to type out that long SQL statement, but that's not code, technically speaking. Notice how the books are now ordered based on an alphabetical listing of authors' last names.

Finding Specific Records

- In addition to using the data control to move through database records, we can write Visual Basic code to accomplish the same, and other, tasks. This is referred to as **programmatic control**. In fact, many times the data control **Visible** property is set to **False** and all data manipulations are performed in code. We can also use programmatic control to find certain records.

- There are four methods used for moving in a database. These methods replicate the capabilities of the four arrow buttons on the data control:

MoveFirst	Move to the first record in the table.
MoveLast	Move to the last record in the table.
MoveNext	Move to the next record (with respect to the current record) in the table.
MovePrevious	Move to the previous record (with respect to the current record) in the table.

- When moving about the database programmatically, we need to test the **BOF** (beginning of file) and **EOF** (end of file) properties. The BOF property is True when the current record is positioned before any data. The EOF property is True when the current record has been positioned past the end of the data. If either property is True, the current record is invalid. If both properties are True, then there is no data in the database table at all.

- These properties, and the programmatic control methods, operate on the **Recordset** property of the data control. Hence, to move to the first record in a table attached to a data control named **dtaExample**, the syntax is:

  dtaExample.Recordset.MoveFirst

- There is a method used for searching a database:

Find	Find a record that meets the specified search criteria.

  This method also operates on the **Recordset** property and has three arguments we will be concerned with. To use **Find** with a data control named **dtaExample**:

  dtaExample.Recordset.Find   Criteria,NumberSkipped,SearchDirection

- The search **Criteria** is a string expression like a **WHERE** clause in SQL. We won't go into much detail on such criteria here. Simply put, the criteria describes what particular records it wants to look at. For example, using our book database, if we want to look at books with titles (the **Title** field) beginning with S, we would use:

Criteria = "Title >= 'S'"

Note the use of single quotes around the search letter. Single quotes are used to enclose strings in Criteria statements. Three logical operators can be used: equals (=), greater than (>), and less than (<).

- The **NumberSkipped** argument tells how many records to skip before beginning the Find. This can be used to exclude the current record by setting NumberSkipped to 1.

- The **SearchDirection** argument has two possible values: **adSearchForward** or **adSearchBackward**. Note, in conjunction with the four Move methods, the SearchDirection argument can be used to provide a variety of search types (search from the top, search from the bottom, etc.)

- If a search fails to find a record that matches the criteria, the Recordset's **EOF** or **BOF** property is set to True (depending on search direction). Another property used in searches is the **Bookmark** property. This allows you to save the current record pointer in case you want to return to that position later. The example illustrates its use.

Example 8-2

'Rolodex' Searching of the Books Database

1. We expand the book database application to allow searching for certain author names. We'll use a 'rolodex' approach where, by pressing a particular letter button, books with author last names corresponding to that button appear on the form.

2. We want a row of buttons starting at 'A' and ending at 'Z' to appear on the lower part of our form. Drawing each one individually would be a big pain, so we'll let Visual Basic do all the work in the **Form_Load** procedure. What we'll do is create one command button (the 'A'), make it a control array, and then dynamically create 25 new control array elements at run-time, filling each with a different letter. We'll even let the code decide on proper spacing.

So, add one command button to the previous form. **Name** it **cmdLetter** and give it a **Caption** of **A**. Set its **Index** property to **0** to make it a control array element. On my form, things at this point look like this:
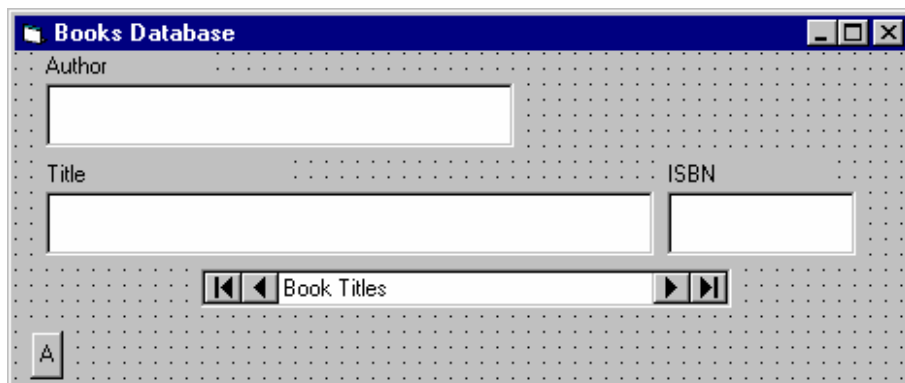
3. Attach this code to the **Form_Load** procedure. This code sets up the rolodex control array and draws the additional 25 letter buttons on the form. (Sorry, you have to type some code now!)

```
Private Sub Form_Load()
Dim I As Integer
'Size buttons
cmdLetter(0).Width = (frmBooks.ScaleWidth - 2*
   cmdLetter(0).Left) / 26
For I = 1 To 25
Load cmdLetter(I) ' Create new control array element
'Position new letter next to previous one
cmdLetter(I).Left = cmdLetter(I - 1).Left +
   cmdLetter(0).Width
'Set caption and make visible
cmdLetter(I).Caption = Chr(vbKeyA + I)
cmdLetter(I).Visible = True
Next I
End Sub
```

At this point, even though all the code is not in place, you could run your application to check how the letter buttons look. My finished form (at run-time) looks like this:
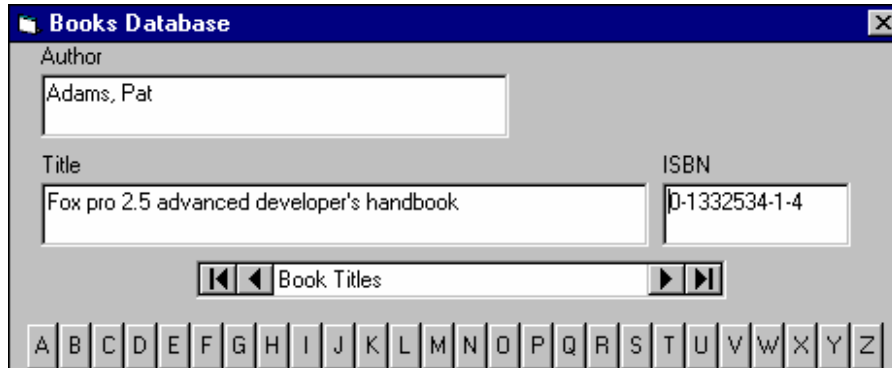
Notice how Visual Basic adjusted the button widths to fit nicely on the form.

4. Attach this code to the **cmdLetter_Click** procedure. In this procedure, we use a search criteria that finds the first occurrence of an author name that begins with the selected letter command button. If the search fails, the record displayed prior to the search is retained (using the **Bookmark** property).

```
Private Sub cmdLetter_Click(Index As Integer)
Dim BookMark1 As Variant
'Mark your place in case no match is found
BookMark1 = dtaTitles.Recordset.Bookmark
'Move to top of table to start search
dtaTitles.Recordset.MoveFirst
dtaTitles.Recordset.Find "Author >= '" +
   cmdLetter(Index).Caption + "'", 0, adSearchForward
If dtaTitles.Recordset.EOF = True Then
  dtaTitles.Recordset.Bookmark = BookMark1
End If
txtAuthor.SetFocus
End Sub
```

Let's look at the search a little closer. We move to the top of the database using **Move First**. Then, the **Find** is executed (notice the selected letter is surrounded by single quotes). If **EOF** is True after the Find, it means we didn't find a match to the Criteria and **Bookmark** is returned to its saved value.

5. Save your application. Test its operation. Note once the program finds the first occurrence of an author name beginning with the selected letter (or next highest letter if there is no author with the pressed letter), you can use the data control navigation buttons (namely the right arrow button) to find other author names beginning with that letter.

Data Manager

- At this point, we know how to use the data control and associated data bound tools to access a database. The power of Visual Basic lies in its ability to manipulate records in code. Such tasks as determining the values of particular fields, adding records, deleting records, and moving from record to record are easily done. This allows us to build a complete database management system (**DBMS**).

- We don't want to change the example database, BIBLIO.MDB. Let's create our own database to change. Fortunately, Visual Basic helps us out here. The **Visual Data Manager** is a Visual Basic Add-In that allows the creation and management of databases. It is simple to use and can create a database compatible with the Microsoft Jet (or Access) database engine.

- To examine an existing database using the Data Manager, follow these steps:

  1. Select **Visual Data Manager** from Visual Basic's **Add-In** menu (you may be asked if you want to add SYSTEM.MDA to the .INI file - answer No.)
  2. Select **Open Database** from the Data Manager **File** menu.
  3. Select the database type and name you want to examine.

  Once the database is opened, you can do many things. You can simply look through the various tables. You can search for particular records. You can apply SQL queries. You can add/delete records. The Data Manager is a DBMS in itself. You might try using the Data Manager to look through the BIBLIO.MDB example database.

- To create a new database, follow these steps:

  1. Select **Visual Data Manager** from Visual Basic's **Add-In** menu (you may be asked if you want to add SYSTEM.MDA to the .INI file - answer No.)
  2. Select **New** from the Data Manager **File** menu. Choose database type (Microsoft Access, Version 7.0), then select a directory and enter a name for your database file. Click **OK**.
  3. The Database window will open. Right click the window and select **New Table**. In the **Name** box, enter the name of your table. Then define the table's fields, one at a time, by clicking **Add Field**, then entering a field name, selecting a data type, and specifying the size of the field, if required. Once the field is defined, click the **OK** button to add it to the field box. Once all fields are defined, click the **Build the Table** button to save your table.

Example 8-3

Phone Directory - Creating the Database
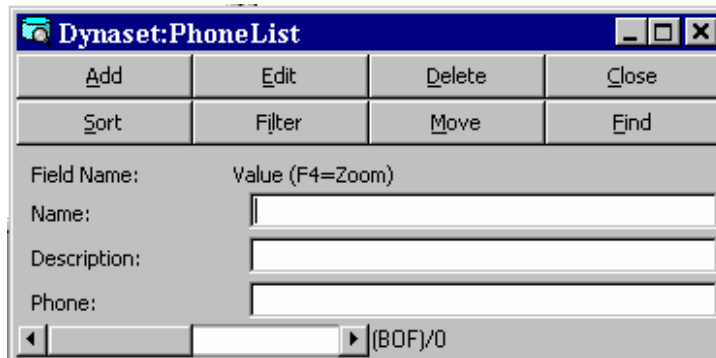
1. With this example, we begin the development of a simple phone directory. In the directory, we will keep track of names and phone numbers. We'll be able to edit, add and delete names and numbers from the directory. And, we'll be able to search the directory for certain names. In this first step, we'll establish the structure for the database we'll use. The directory will use a single table, with three fields: **Name**, **Description**, and **Phone**. Name will contain the name of the person or company, Description will contain a descriptive phrase (if desired) of who the person or company is, and Phone will hold the phone number.

2. Start the Data Manager. Use the previously defined steps to establish a new database (this is a Microsoft Access, Version 7.0 database). Use **PhoneList** as a Name for your database table. Define the three fields. Each should be a **Text** data type. Assign a size of **40** to the **Name** and **Description** fields, a size of **15** to the **Phone** field. When all fields have been defined, the screen should look like this:

When done with the field definitions, click **Build the Table** to save your new
table. You will be returned to the Database Tables window.

3. We're now ready to enter some data in our database. From the Database Tables
window, right click the **PhoneList** table and select **Open**. The following window
will appear:

At this point, add several (at least five - make them up or whatever) records to your
database. The steps for each record are: (1) click **Add** to add a record, (2) fill in the
three fields (or, at least the Name and Phone fields), and (3) click **Update** to save the
contents.

You can also **Delete** records and **Find** records, if desired. You can move through
the records using the scroll bar at the bottom of the screen. When done
entering records, click **Close** to save your work. Select **Exit** from the Data
Manager File menu. Your database has been created.

Database Management

- The Data Manager is a versatile utility for creating and viewing databases. However, its interface is not that pretty and its use is somewhat cumbersome. We would not want to use it as a **database management system** (**DBMS**). Nor, would we expect users of our programs to have the Data Manager available for their use. The next step in our development of our database skills is to use Visual Basic to manage our databases, that is develop a DBMS.

- We will develop a simple DBMS. It will allow us to **view** records in an existing database. We will be able to **edit** records, **add** records, and **delete** records. Such advanced tasks as adding tables and fields to a database and creating a new database can be done with Visual Basic, but are far beyond the scope of the discussion here.

- To create our DBMS, we need to define a few more programmatic control methods associated with the data control **Recordset** property. These methods are:

AddNew	A new record is added to the table. All fields are set to Null and this record is made the current record.
Delete	The current record is deleted from the table. This method must be immediately followed by one of the Move methods because the current record is invalid after a Delete.
Update	Saves the current contents of all bound tools.

- To **edit** an existing record, you simply display the record and make any required changes. The **LockType** property should be set to **adLockPessimistic** (locks each record as it is edited). Then, when you move off of that record, either with a navigation button or through some other action, Visual Basic will automatically update the record. If desired, or needed, you may invoke the **Update** method to force an update (use **LockType** = **asLockOptimistic**). For a data control named **dtaExample**, the syntax for this statement is:

      dtaExample.Recordset.Update

- To **add** a record to the database, we invoke the **AddNew** method. The syntax for our example data control is:

      dtaExample.Recordset.AddNew

  This statement will blank out any bound data tools and move the current record to the end of the database. At this point, you enter the new values. When you move off of this record, the changes are automatically made to the database. Another way to update the database with the changes is via the Update method.

After adding a record to a database, you should invoke the **Refresh** property of the data control to insure proper sorting (established by RecordSource SQL statement) of the new entry. The format is:

>   dtaExample.Refresh

- To **delete** a record from the database, make sure the record to delete is the current record. Then, we use the **Delete** method. The syntax for the example data control is:

>   dtaExample.Recordset.Delete

Once we execute a Delete, we must move (using one of the 'Move' methods) off of the current record because it no longer exists and an error will occur if we don't move. This gets particularly tricky if deleting the last record (check the **EOF** property). If EOF is true, you must move to the top of the database (**MoveFirst**). You then must make sure there is a valid record there (check the **BOF** property). The example code demonstrates proper movement.

Example 8-4

Phone Directory - Managing the Database

1. Before starting, make a copy of your phone database file using the Windows Explorer. That way, in case we mess up, you still have a good copy. And, create a data link to the database. Here, we develop a simple DBMS for our phone number database. We will be able to display individual records and edit them. And, we will be able to add or delete records. Note this is a simple system and many of the fancy 'bells and whistles' (for example, asking if you really want to delete a record) that should really be here are not. Adding such amenities is left as an exercise to the student.

2. Load your last Books Database application (Example 8-2 - the one with the 'Rolodex' search). We will modify this application to fit the phone number DBMS. Resave your form and project with different names. Add three command buttons to the upper right corner of the form. Modify/set the following properties for each tool. For the data control and text boxes, make sure you follow the order shown.

    frmBooks (this is the old name)**:**
Caption	Phone List
Name	frmPhone

    dtaTitles (this is the old name)**:**
Caption	Phone Numbers
ConnectionString	[your phone database data link] (select, don't type)
RecordSource	SELECT * FROM PhoneList ORDER BY Name (the ORDER keyword sorts the database by the given field)
Name	dtaPhone
LockType	adLockOptimistic

    Label1:
Caption	Description

    Label2:
Caption	Phone

    Label3:
Caption	Name

txtAuthor (this is the old name)**:**

      DataSource   dtaPhone (select, don't type)

      DataField   Name (select, don't type)

      Locked   False

      Name   txtName

MaxLength   40

      TabIndex   1

txtISBN (this is the old name)**:**

DataSource   dtaPhone (select, don't type)

DataField   Phone (select, don't type)

Locked   False

Name   txtPhone

MaxLength   15

TabIndex   3

txtTitle (this is the old name)**:**

      DataSource   dtaPhone (select, don't type)

      DataField   Description (select, don't type)

      Locked   False

      Name   txtDesc

MaxLength   40

TabIndex   2

      Command1:

      Caption   &&Add

      Name   cmdAdd

      Command2:

      Caption   &&Save

      Enabled   False

      Name   cmdSave

      Command3:

      Caption   &&Delete

      Name   cmdDelete

When done, my form looked like this:
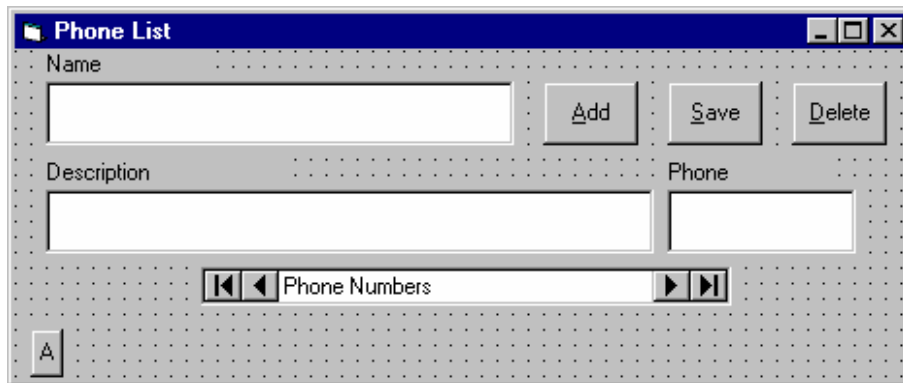
At this point, you can run your application and you should be able to navigate through your phone database using the data control. Don't try any other options, though. We need to do some coding.

3. In **Form_Load**, replace the word **frmBooks** with **frmPhone** . This will allow the letter keys to be displayed properly.

4. In the **cmdLetter_Click** procedure, replace all occurrences of the word **dtaTitles** with **dtaPhone** . Replace all occurrences of **Author** with **Name** . The modified code will be:

```
Private Sub cmdLetter_Click(Index As Integer)
Dim BookMark1 As Variant
'Mark your place in case no match is found
BookMark1 = dtaPhone.Recordset.Bookmark
dtaPhone.Recordset.MoveFirst
dtaPhone.Recordset.Find "Name >= '" +
   cmdLetter(Index).Caption + "'"
If dtaPhone.Recordset.EOF = True Then
  dtaPhone.Recordset.Bookmark = BookMark1
End If
txtName.SetFocus
End Sub
```

5. Attach this code to the **cmdAdd_Click** procedure. This code invokes the code
    needed to add a record to the database. The **Add** and **Delete** buttons are disabled.
    Click the **Save** button when done adding a new record.

```
Private Sub cmdAdd_Click()
cmdAdd.Enabled = False
cmdSave.Enabled = True
cmdDelete.Enabled = False
dtaPhone.Recordset.AddNew
txtName.SetFocus
End Sub
```

6. Add this code to the **cmdSave_Click** procedure. When done entering a new record,
    the command button status's are toggled, the Recordset updated, and the data control
    Refresh method invoked to insure proper record sorting.

```
Private Sub cmdSave_Click()
dtaPhone.Recordset.Update
dtaPhone.Refresh
cmdAdd.Enabled = True
cmdSave.Enabled = False
cmdDelete.Enabled = True
txtName.SetFocus
End Sub
```

7. Attach this code to the **cmdDelete_Click** procedure. This deletes the current record
    and moves to the next record. If we bump into the end of file, we need to check if
    there are no records remaining. If no records remain in the table, we display a
    message box. If records remain, we move around to the first record.

```
Private Sub cmdDelete_Click()
dtaPhone.Recordset.Delete
dtaPhone.Recordset.MoveNext
If dtaPhone.Recordset.EOF = True Then
  dtaPhone.Refresh
  If dtaPhone.Recordset.BOF = True Then
    MsgBox "You must add a record.", vbOKOnly +
  vbInformation, "Empty file"
    Call cmdAdd_Click
  Else
    dtaPhone.Recordset.MoveFirst
  End If
End If
txtName.SetFocus
End Sub
```

8. Save the application. Try running it. Add records, delete records, edit records. If you're really adventurous, you could add a button that dials your phone (via modem) for you! Look at the custom communications control.

Custom Data Aware Controls

- As mentioned earlier, there are three **custom** data aware **tools**, in addition to the standard Visual Basic tools: the **DataList**, **DataCombo**, and **DataGrid** ADO tools. We'll present each of these, giving their suggested use, some properties and some events. If the icons for these tools are not in the toolbox, select **Project** from the main menu, then click **Components**. Select **Microsoft DataList Controls 6.0 (OLEDB)** and **Microsoft DataGrid 6.0 (OLEDB)** in the Components window. Click **OK** - the controls will appear.

- Like the data control, previous versions of Visual Basic used DAO versions of the list, combo, and grid controls, named DBList, DBCombo, and DBGrid. Make sure you are not using these tools.

- DataList Box:

The first bound data custom tool is the **DataList Box**. The list box is automatically filled with a field from a specified data control. Selections from the list box can then be used to update another field from the same data control or, optionally, used to update a field from another data control.

Some properties of the DataList box are:

DataSource	Name of data control that is updated by the selection.
DataField	Name of field updated in Recordset specified by DataSource.
RowSource	Name of data control used as source of items in list box.
ListField	Name of field in Recordset specified by RowSource used to fill list box.
BoundColumn	Name of field in Recordset specified by RowSource to be passed to DataField, once selection is made. This is usually the same as ListField.
BoundText	Text value of BoundColumn field. This is the value passed to DataField property.
Text	Text value of selected item in list. Usually the same as BoundText.

The most prevalent use of the DataList box is to fill the list from the database, then allow selections. The selection can be used to fill any tool on a form, whether it is data aware or not.

As a quick example, here is a **DataList** box filled with the **Title** (**ListField**) field from the **dtaExample** (**RowSource**) data control. The data control is bound to the **Titles** table in the **BIBLIO.MDB** database.
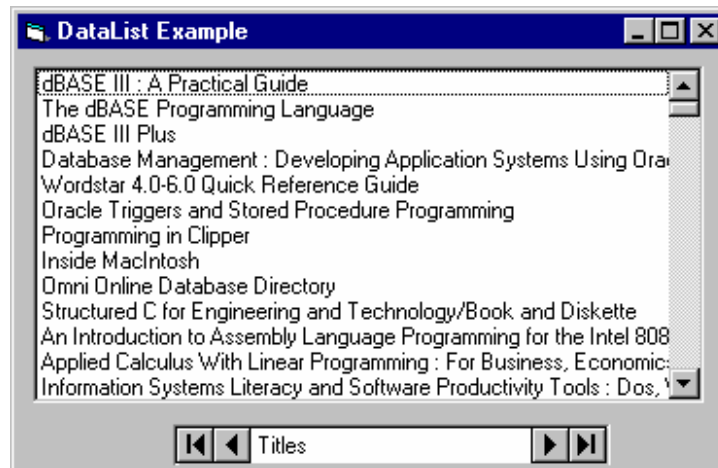
• DataCombo Box:

The **DataCombo Box** is nearly identical to the DataList box, hence we won't look at a separate set of properties. The only differences between the two tools is that, with the DataCombo box, the list portion appears as a drop-down box and the user is given the opportunity to change the contents of the returned **Text** property.
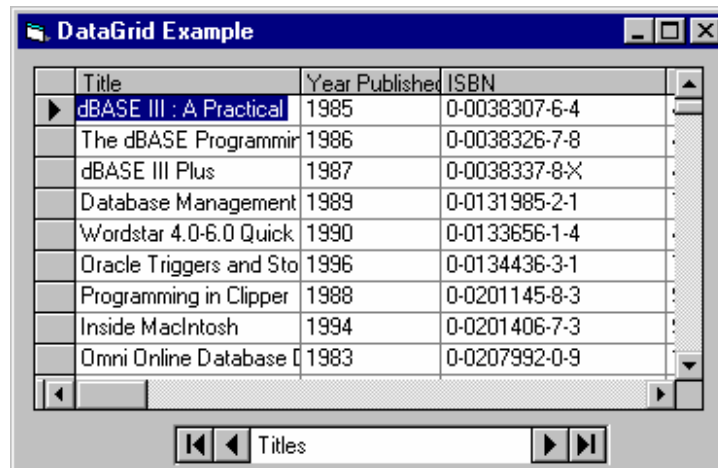
• DataGrid Tool:

The **DataGrid** tool is, by far, the most useful of the custom data bound tools. It can display an entire database table, referenced by a data control. The table can then be edited as desired.

The DataGrid control is in a class by itself, when considering its capabilities. It is essentially a separate, highly functional program. The only property we'll be concerned with is the **DataSource** property, which, as always, identifies the table associated with the respective data control. Refer to the Visual Basic Programmer's Guide and other references for complete details on using the DataGrid control.

As an example of the power of the **DataGrid** control, here's what is obtained by simply setting the **DataSource** property to the **dtaExample** data control, which is bound to the **Titles** table in the **BIBLIO.MDB** database:

At this point, we can scroll through the table and edit any values we choose. Any changes are automatically reflected in the underlying database. Column widths can be changed at run-time! Multiple row and column selections are possible! Like we said, a very powerful tool.

Creating a Data Report

- Once you have gone to all the trouble of developing and managing a database, it is nice to have the ability to obtain printed or displayed information from your data. The process of obtaining such information is known as creating a **data report**.

- There are two steps to creating a data report. First, we need to create a **Data Environment**. This is designed within Visual Basic and is used to tell the data report what is in the database. Second, we create the **Data Report** itself. This, too, is done within Visual Basic. The Data Environment and Data Report files then become part of the Visual Basic project developed as a database management system.

- The Visual Basic 6.0 data report capabilities are vast and using them is a detailed process. The use of these capabilities is best demonstrated by example. We will look at the rudiments of report creation by building a tabular report for our phone database.

Example 8-5

Phone Directory - Building a Data Report

We will build a data report that lists all the names and phone numbers in our phone database. We will do this by first creating a Data Environment, then a Data Report. We will then reopen the phone database management project and add data reporting capabilities.

Creating a Data Environment

1.   Start a new **Standard EXE** project.

2.   On the **Project** menu, click **Add Data Environment**. If this item is not on the menu, click **Components**. Click the **Designers** tab, and choose **Data Environment** and click **OK** to add the designer to your menu.

3.   We need to point to our database. In the **Data Environment** window, right-click the **Connection1** tab and select **Properties**. In the **Data Link Properties** dialog box, choose **Microsoft Jet 3.51 OLE DB Provider**. Click **Next** to get to the **Connection** tab. Click the ellipsis button. Find your phone database (**mdb**) file. Click **OK** to close the dialog box.

4.   We now tell the Data Environment what is in our database. Right-click the **Connection1** tab and click **Rename**. Change the name of the tab to **Phone**. Right-click this newly named tab and click **Add Command** to create a **Command1** tab. Right-click this tab and choose **Properties**. Assign the following properties:

Command Name	PhoneList
Connection	Phone
DataBase Object	Table
ObjectName	PhoneList

5.   Click **OK**. All this was needed just to connect the environment to our database.

6. Display the properties window and give the data environment a name property of **denPhone**. Click **File** and **Save denPhone As**. Save the environment in an appropriate folder. We will eventually add this file to our phone database management system. At this point, my data environment window looks like this (I expanded the PhoneList tab by clicking the + sign):
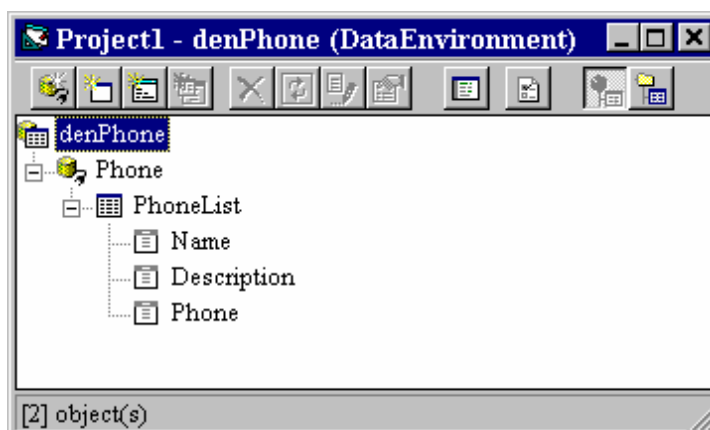
Creating a Data Report

Once the Data Environment has been created, we can create a Data Report. We will drag things out of the Data Environment onto a form created for the Data Report, so make sure your Data Environment window is still available.

1. On the **Project** menu, click **Add Data Report** and one will be added to your project. If this item is not on the menu, click **Components**. Click the **Designers** tab, and choose **Data Report** and click **OK** to add the designer to your menu.

2. Set the following properties for the report:

Name	rptPhone
Caption	Phone Directory
DataSource	denPhone (your phone data environment - choose, don't type)
DataMember	PhoneList (the table name - choose don't type)

3. Right-click the Data Report and click **Retrieve Structure**. This establishes a report format based on the Data Environment.

4. Note there are five sections to the data report: a **Report Header**, a **Page Header**, a **Detail** section, a **Page Footer**, and a **Report Footer**. The headers and footers contain information you want printed in the report and on each page. To place information in one of these regions, right-click the selected region, click **Add Control**, then choose the control you wish to place. These controls are called **data report controls** and

properties are established just like you do for usual controls. Try adding some headers.

5. The Detail section is used to layout the information you want printed for each record in your database. We will place two field listings (**Name**, **Phone**) there. Click on the Name tab in the Data Environment window and drag it to the Detail section of the Data Report. Two items should appear: a text box **Name** and a text box **Name (PhoneList).** The first text box is heading information. Move this text box into the Page Header section. The second text box is the actual value for Name from the PhoneList table. Line this text box up under the Name header. Now, drag the Phone tab from the Data Environment to the Data Report. Adjust the text boxes in the same manner. Our data report will have page headers Name and Phone. Under these headers, these fields for each record in our database will be displayed. When done, the form should look something like this:
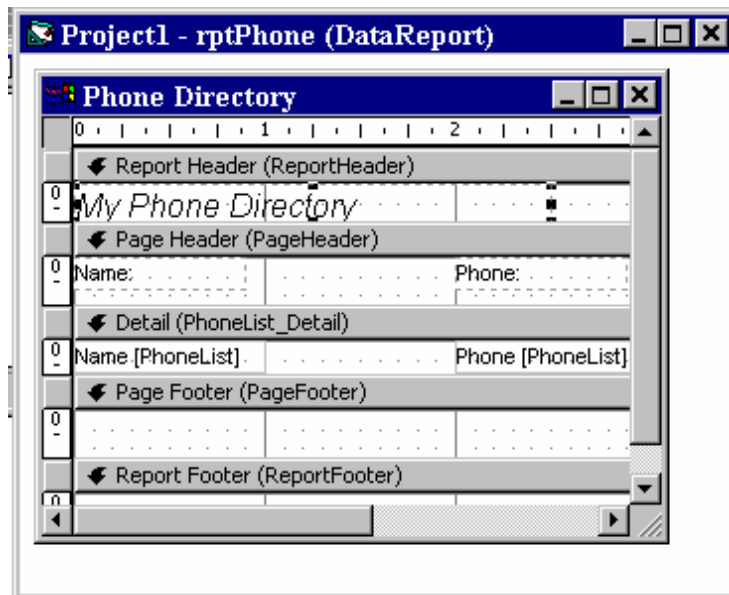
In this form, I've resized the labels a bit and added a Report Header. Also, make sure you close up the Detail section to a single line. Any space left in this section will be inserted after each entry.

6. Click **File** and **Save rptPhone As**. Save the environment in an appropriate folder. We will now reopen our phone database manager and attach this and the data environment to that project and add capabilities to display the report.

Accessing the Data Report

1. Reopen the phone directory project. Add a command button named **cmdReport** and give it a **Caption** of **Show Report**. (There may be two tabs in your toolbox, one named **General** and one named **DataReport**. Make sure you select from the General tools.)

2. We will now add the data environment and data report files to the project. Click the **Project** menu item, then click **Add File**. Choose **denPhone** and click **OK**. Also add **rptPhone**. Look at your Project Window. Those files should be listed under **Designers**.

3. Use this code in **cmdReport_Click**:

```
Private Sub cmdReport_Click()
rptPhone.Show
End Sub
```

4. This uses the **Show** method to display the data report.

5. Save the application and run it. Click the Show Report button and this should appear:

Phone Directory
My Phone Directory

Name:	Phone:
KIDware	(206) 721-2556
Santa Claus	777-7777
Holiday Travel	55-55555
Opposum Jones	111-1111
Henrietta Johnson	678-9054
UW Extension	444-4444
The President	999-9999
Alan's Plumbing	222-2222
Bob's Appliance	333-3333
Zebra Lodging	234-5657
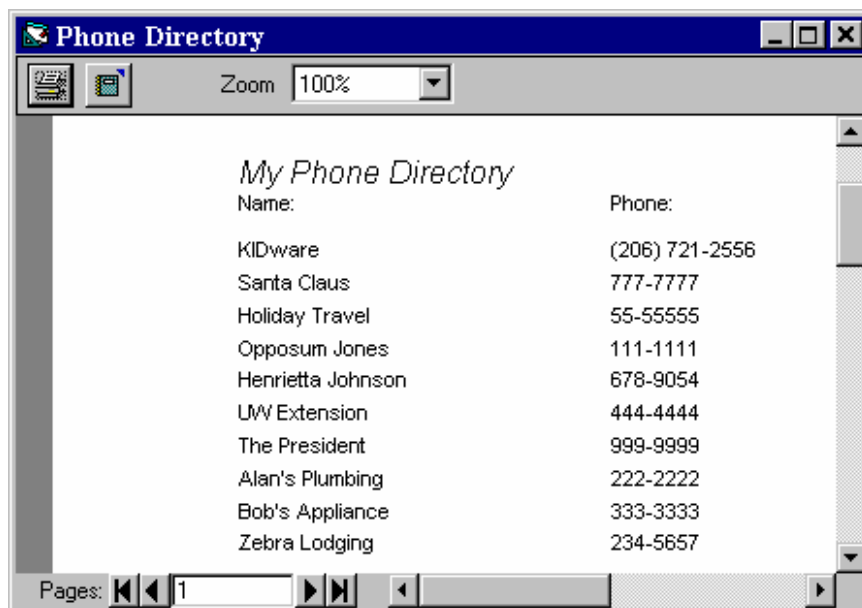
You now have a printable copy of the phone directory. Just click the Printer icon. Notice the relationship with this displayed report and the sections available in the Data Report designer.

This page intentionally not left blank.

Exercise 8

Home Inventory Database

Design and develop an application that manages a home inventory database. Add the option of obtaining a printed list of your inventoried property.

My Solution:

Database Design:

The first step is to design a database using **Data Manager** (or **Access**). My database is a single table (named **MYSTUFF**). Its specifications are:

Field Name	Field Type	Field Length
Item	Text	40
Serial Number	Text	20
Date Purchased	Text	20
New Value	Currency	<N/A>
Location	Text	40

This database is saved as file **HomeInv.mdb.** Create a data link to your database. The link is saved as **HomeInv.udl**.

Report Design:

The second step is to use the Data Environment and Data Report designers to setup how you want the printed home inventory to appear. Use your discretion here. My final report design is saved in **denHomeInv** and **rptHomeInv**. We will access this report from our Visual Basic application. My Data Report design looks like this:
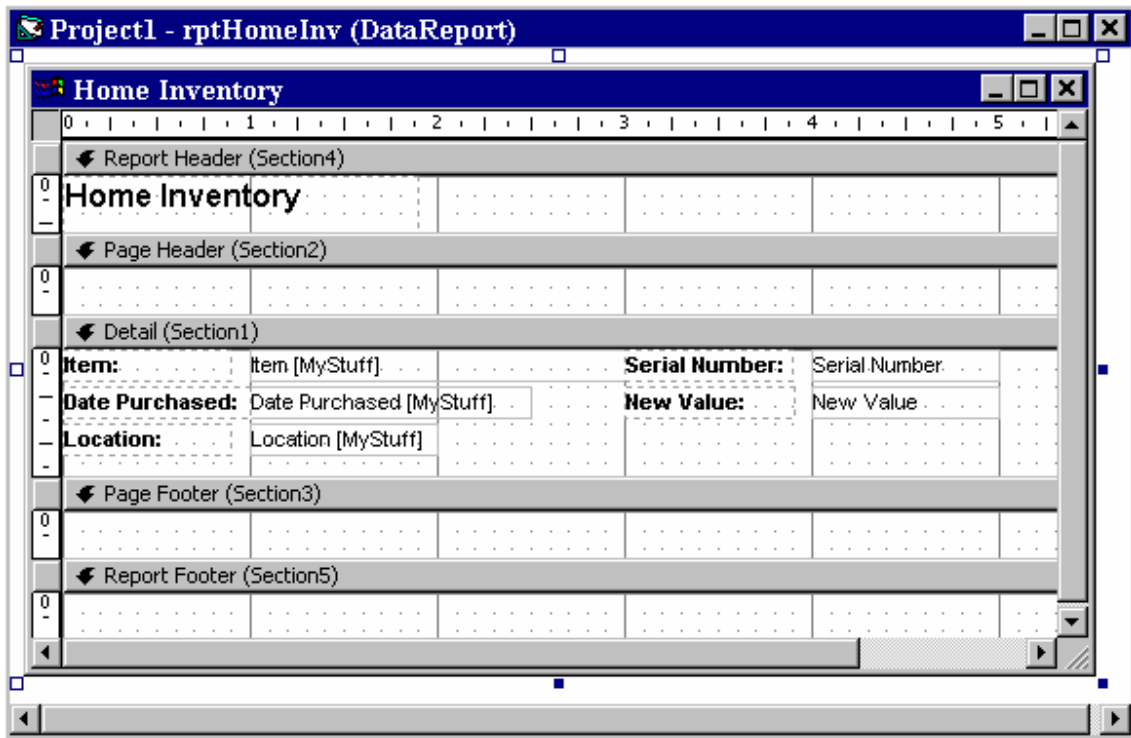
Project Design:

Form:
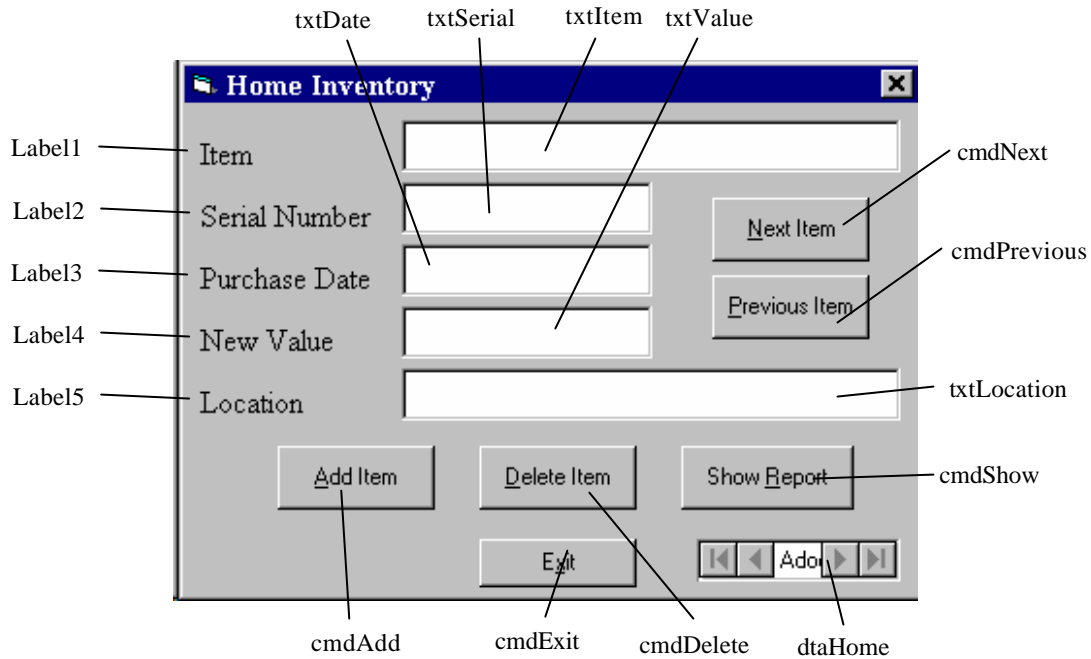
Properties:

Form **frmHome** :
        BorderStyle = 1 - Fixed Single
        Caption = Home Inventory

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

ADO Data Control **dtaHome:**
        Caption = Book Titles
        ConnectionString = HomeInv.udl (in whatever folder you saved it in -
                      select, don't type)
        RecordSource = SELECT * FROM MyStuff
        Visible = False

CommandButton **cmdShow**:
        Caption = Show &Report

CommandButton **cmdPrevious** :
        Caption = &Previous Item

CommandButton **cmdNext**:
        Caption = &Next Item

CommandButton **cmdDelete**:
        Caption = &Delete Item

CommandButton **cmdAdd**:
        Caption = &Add Item

TextBox **txtLocation**:
        DataField = Location
        DataSource = dtaHome
        FontName = MS Sans Serif
        FontSize = 9.75
        MaxLength = 40

TextBox **txtValue**:
        DataField = New Value
        DataSource = dtaHome
        FontName = MS Sans Serif
        FontSize = 9.75

TextBox **txtDate**:
        DataField = Date Purchased
        DataSource = dtaHome
        FontName = MS Sans Serif
        FontSize = 9.75
        MaxLength = 20

TextBox **txtSerial**:
        DataField = Serial Number
        DataSource = dtaHome
        FontName = MS Sans Serif
        FontSize = 9.75
        MaxLength = 20

TextBox **txtItem**:
        DataField = Item
        DataSource = dtaHome
        FontName = MS Sans Serif
        FontSize = 9.75
        MaxLength = 40

Label **Label5**:
      Caption = Location
      FontName = Times New Roman
      FontSize = 12

Label **Label4**:
      Caption = New Value
      FontName = Times New Roman
      FontSize = 12

Label **Label3**:
      Caption = Purchase Date
      FontName = Times New Roman
      FontSize = 12

Label **Label2**:
      Caption = Serial Number
      FontName = Times New Roman
      FontSize = 12

Label **Label1**:
      Caption = Item
      FontName = Times New Roman
      FontSize = 12

Code:

General Declarations:

```
Option Explicit
```

cmdAdd Click Event:

```
Private Sub cmdAdd_Click()
'Add new item to database
dtaHome.Recordset.AddNew
txtItem.SetFocus
End Sub
```

cmdDelete Click Event:

```
Private Sub cmdDelete_Click()
'Delete item from database
Dim Rvalue As Integer
Rvalue = MsgBox("Are you sure you want to delete this
item?", vbQuestion + vbYesNo, "Delete Item")
If Rvalue = vbNo Then Exit Sub
dtaHome.Recordset.Delete
dtaHome.Recordset.MoveNext
If dtaHome.Recordset.EOF Then
  If dtaHome.Recordset.BOF Then
    MsgBox "You must add an item.", vbOKOnly +
vbInformation, "Empty Database"
    Call cmdAdd_Click
  Else
    dtaHome.Recordset.MoveFirst
  End If
End If
txtItem.SetFocus
End Sub
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
End
End Sub
```

cmdNext Click Event:

```
Private Sub cmdNext_Click()
'Move to next item - if at end-of-file, backup one item
dtaHome.Recordset.MoveNext
If dtaHome.Recordset.EOF Then
dtaHome.Recordset.MovePrevious
txtItem.SetFocus
End Sub
```

cmdPrevious Click Event:

```
Private Sub cmdPrevious_Click()
'Move to previous item - if at beginning-of-file, go down
one item
dtaHome.Recordset.MovePrevious
If dtaHome.Recordset.BOF Then dtaHome.Recordset.MoveNext
txtItem.SetFocus
End Sub
```

cmdShow Click Event:

```
Private Sub cmdShow_Click()
rptHomeInv.Show
End Sub
```

This page intentionally not left blank.

Learn Visual Basic 6.0

9. Dynamic Link Libraries and the Windows API

Review and Preview

*   In our last class, we saw how using the data control and bound data tools allowed us to develop a simple database management system. Most of the work done by that DBMS, though, was done by the underlying Jet database engine, not Visual Basic. In this class, we learn how to interact with another underlying set of code by programming the Windows applications interface (API) using dynamic link libraries (DLL). Alphabet soup!

Dynamic Link Libraries (DLL)

*   All Windows applications at their most basic level (even ones written using Visual Basic) interact with the computer environment by using calls to **dynamic link libraries** (**DLL**). DLL's are libraries of routines, usually written in C, C++, or Pascal, that you can link to and use at run-time.

*   Each DLL usually performs a specific function. By using DLL routines with Visual Basic, you are able to extend your application's capabilities by making use of the

many hundreds of functions that make up the Windows Application Programming Interface (**Windows API**). These functions are used by virtually every application to perform functions like displaying windows, file manipulation, printer control, menus and dialog boxes, multimedia, string manipulation, graphics, and managing memory.

*   The advantage to using DLL's is that you can use available routines without having to duplicate the code in Basic. In many cases, there isn't even a way to do a function in Basic and calling a DLL routine is the only way to accomplish the task. Or, if there is an equivalent function in Visual Basic, using the corresponding DLL routine may be faster, more efficient, or more adaptable. Reference material on DLL calls and the API run thousands of pages - we'll only scratch the surface here. A big challenge is just trying to figure out what DLL procedures exist, what they do, and how to call them.

- There is a price to pay for access to this vast array of code. Once you leave the protective surroundings of the Visual Basic environment, as you must to call a DLL, you get to taunt and tease the dreaded general protection fault (**GPF**) monster, which can bring your entire computer system to a screeching halt! So, be careful. And, if you don't have to use DLL's, don't.

Accessing the Windows API With DLL

- Using a DLL procedure from Visual Basic is not much different from calling a general basic function or procedure. Just make sure you pass it the correct number and correct type of arguments. Say **DLLFcn** is a DLL function and **DLLProc** is a DLL procedure. Proper syntax to invoke these is, respectively (ignoring arguments for now):

      ReturnValue = DLLFcn()
      Call DLLProc()

- Before you call a DLL procedure, it must be declared in your Visual Basic program using the **Declare** statement. Declare statements go in the **general declarations** area of form and code modules. The Declare statement informs your program about the name of the procedure, and the number and type of arguments it takes. This is nearly identical to function prototyping in the C language. For a DLL function (**DLLFcn**), the syntax is:

      Declare Function DLLFcn Lib *DLLname* [(argument list)] As *type*

  where ***DLLname*** is a string specifying the name of the DLL file that contains the procedure and ***type*** is the returned value type.

  For a procedure (**DLLProc**), use:

      Declare Sub DLLProc Lib *DLLname* [(argument list)]

  In code modules, you need to preface the Declare statements with the keywords **Public** or **Private** to indicate the procedure scope. In form modules, preface the Declare statement with **Private**, the default (and only possible) scope in a form module.

- Nearly all arguments to DLL procedures are passed by value (use the **ByVal** keyword), so the argument list has the syntax:

      ByVal *argname1* As *type*, ByVal *argname2* As *type*, ...

Again, it is very important, when calling DLL procedures, that the argument lists be correct, both regarding number and type. If the list is not correct, very bad things can happen.

• And, it is critical that the **Declare** statement be exactly correct or very bad things can happen. Fortunately, there is a program included with Visual Basic called the **API Text Viewer**, which provides a complete list of Declare statements for all API procedures. The viewer is available from the Start Menu folder for Visual Basic 6.0 (choose Visual Basic 6.0 Tools folder, then API Text Viewer). Most of the Declare statements are found in a file named **win32api.txt** (load this from the **File** menu).
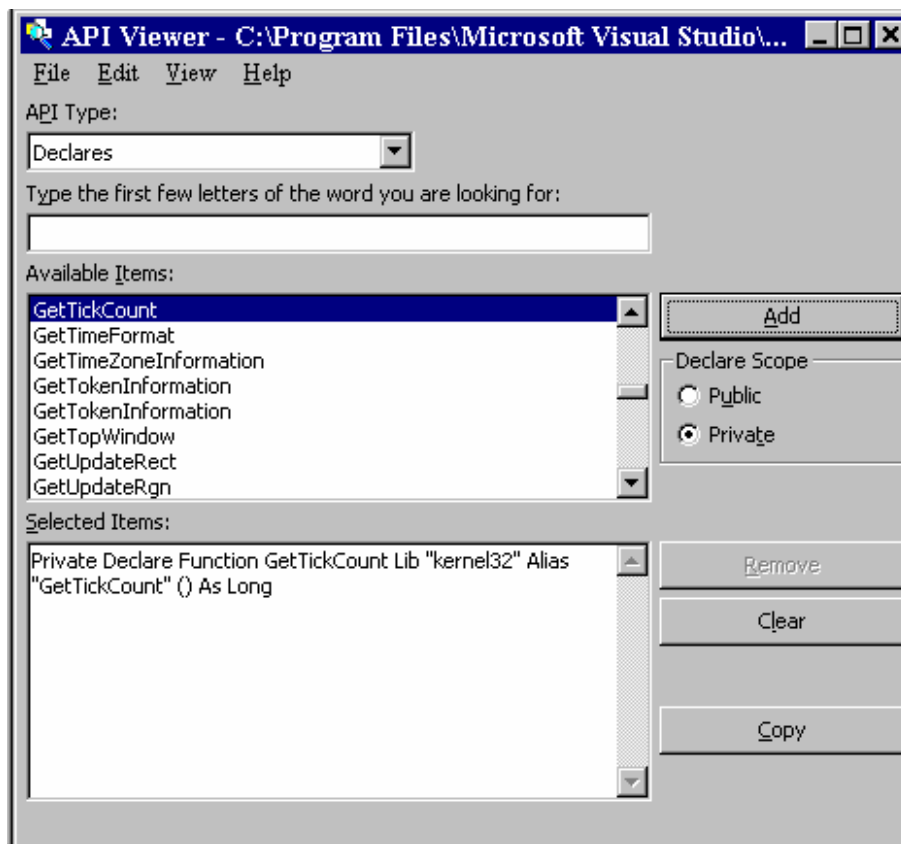
Always use this program to establish Declare statements for your DLL calls. The procedure is simple. Scroll through the listed items and highlight the desired routine. Choose the scope (**Public** or **Private**). Click **Add** to move it to the **Selected Items** area. Once all items are selected, click **Copy**. This puts the desired Declare statements in the Windows clipboard area. Then move to the General Declarations area of your application and choose **Paste** from the **Edit** menu. The Declare statements will magically appear. The API Text Viewer can also be used to obtain any constants your DLL routine may need.

- To further confuse things, unlike Visual Basic routine names, DLL calls are **case-sensitive**, we must pay attention to proper letter case when accessing the API.

- Lastly, **always, always, always save** your Visual Basic application before testing any DLL calls. More good code has gone down the tubes with GPF's - they are very difficult to recover from. Sometimes, the trusty on-off switch is the only recovery mechanism.

Timing with DLL Calls

- Many times you need some method of timing within an application. You may want to know how long a certain routine (needed for real-time simulations) takes to execute or want to implement some sort of delay in your code. The DLL function **GetTickCount** is very useful for such tasks.

- The DLL function **GetTickCount** is a measure of the number of milliseconds that have elapsed since Windows was started on your machine. GetTickCount is 85 percent faster than the Visual Basic **Timer** or **Now** functions. The GetTickCount function has no arguments. The returned value is a **long** integer. The usage syntax is:

      Dim TickValue as Long
         .
         .
      TickValue = GetTickCount()

  Let's look at a couple of applications of this function.

Quick Example 1: Using GetTickCount to Build a Stopwatch

Remember way back in Class 1, where we built a little stop watch. We'll modify that example here using **GetTickCount** to do our timing.

1. Load Example 1-3 from long, long ago.

2. Use the API Text Viewer to obtain the Declare statement for the **GetTickCount** function. Choose **Private** scope. Copy and paste it into the applications **General Declarations** area (new code is italicized).

```
Option Explicit
Dim StartTime As Variant
Dim EndTime As Variant
Dim ElapsedTime As Variant
Private Declare Function GetTickCount Lib "kernel32" () As
   Long
```

3. Modify the **cmdStart_Click** procedure as highlighted:

```
Private Sub cmdStart_Click()
'Establish and print starting time
StartTime = GetTickCount() / 1000
lblStart.Caption = Format(StartTime, "#########0.000")
lblEnd.Caption = ""
lblElapsed.Caption = ""
End Sub
```

4. Modify the **cmdEnd_Click** procedure as highlighted:

```
Private Sub cmdEnd_Click()
'Find the ending time, compute the elapsed time
'Put both values in label boxes
EndTime = GetTickCount() / 1000
ElapsedTime = EndTime - StartTime
lblEnd.Caption = Format(EndTime, "#########0.000")
lblElapsed.Caption = Format(ElapsedTime, "#########0.000")
End Sub
```

5. Run the application. Note we now have timing with millisecond (as opposed to one second) accuracy.

Quick Example 2: Using GetTickCount to Implement a Delay

Many times, you want some delay in a program. We can use GetTickCount to form a user routine to implement such a delay. We'll write a quick example that delays two seconds between beeps.

1. Start a new project. Put a command button on the form. Copy and paste the proper Declare statement.

2. Use this for the **Command1_Click** event:

```
Private Sub Command1_Click()
Beep
Call Delay(2#)
Beep
End Sub
```

3. Add the routine to implement the delay. The routine I use is:

```
Private Sub Delay(DelaySeconds As Single)
Dim T1 As Long
T1 = GetTickCount()
Do While GetTickCount() - T1 < CLng(DelaySeconds * 1000)
Loop
End Sub
```

To use this routine, note you simply call it with the desired delay (in seconds) as the argument. This example delays two seconds. One drawback to this routine is that the application cannot be interrupted and no other events can be processed while in the Do loop. So, keep delays to small values.

4. Run the example. Click on the command button. Note the delay between beeps.

Drawing Ellipses

* There are several DLL routines that support graphic methods (similar to the Line and Circle methods studied in Class 7). The DLL function **Ellipse** allows us to draw an ellipse bounded by a pre-defined rectangular region.

* The **Declare** statement for the **Ellipse** function is:

    Private Declare Function Ellipse Lib "gdi32" Alias "Ellipse" (ByVal hdc As Long, ByVal X1 As Long, ByVal Y1 As Long, ByVal X2 As Long, ByVal Y2 As Long) As Long

    Note there are five arguments: **hdc** is the hDC handle for the region (Form or Picture Box) being drawn to, (**X1, Y1**) define the upper left hand corner of the rectangular region surrounding the ellipse and (**X2,Y2**) define the lower right hand corner. The region drawn to must have its **ScaleMode** property set to **Pixels** (all DLL drawing routine use pixels for coordinates).

* Any ellipse drawn with this routine is drawn using the currently selected **DrawWidth** and **ForeColor** properties and filled according to **FillColor** and **FillStyle**.

Quick Example 3 - Drawing Ellipses

1. Start a new application. Set the form's **ScaleMode** property to **Pixels**.

2. Use the API Text Viewer to obtain the Declare statement for the Ellipse function and copy it into the **General Declarations** area:

```
Option Explicit
Private Declare Function Ellipse Lib "gdi32" (ByVal hdc As
   Long, ByVal X1 As Long, ByVal Y1 As Long, ByVal X2 As
   Long, ByVal Y2 As Long) As Long
```

3. Attach the following code to the **Form_Resize** event:

```
Private Sub Form_Resize()
Dim RtnValue As Long
Form1.Cls
RtnValue = Ellipse(Form1.hdc, 0.1 * ScaleWidth, 0.1 *
   ScaleHeight, 0.9 * ScaleWidth, 0.9 * ScaleHeight)
End Sub
```

4. Run the application. Resize the form and see how the drawn ellipse takes on new shapes. Change the form's **DrawWidth**, **ForeColor**, **FillColor**, and **FillStyle** properties to obtain different styles of ellipses.

Drawing Lines

- Another DLL graphic function is **Polyline**. It is used to connect a series of connected line segments. This is useful for plotting information or just free hand drawing. Polyline uses the **DrawWidth** and **DrawStyle** properties. This function is similar to the Line method studied in Class 7, however the last point drawn to (**CurrentX** and **CurrentY**) is not retained by this DLL function.

- The **Declare** statement for **Polyline** is:

  Private Declare Function Polyline Lib "gdi32" Alias "Polyline" (ByVal hdc As Long, lpPoint As POINTAPI, ByVal nCount As Long) As Long

  Note it has three arguments: **hdc** is the hDC handle of the region (Form or Picture Box-again, make sure **ScaleMode** is **Pixels**) being drawn to, **lpPoint** is the first point in an array of points defining the endpoints of the line segments - it is of a special user-defined type **POINTAPI** (we will talk about this next), and **nCount** is the number of points defining the line segments.

- As mentioned, Polyline employs a special user-defined variable (a data structure) of type POINTAPI. This definition is made in the general declarations area and looks like:

  Private Type POINTAPI
    X As Long
    Y As Long
  End Type

  Any variable defined to be of type POINTAPI will have two coordinates, an X value and a Y value. As an example, say we define variable A to be of type POINTAPI using:

  Dim A As POINTAPI

  A will have an X value referred to using the dot notation A.X and a Y value referred to as A.Y. Such notation makes using the Polyline function simpler. We will use this variable type to define the array of line segment endpoints.

- So, to draw a sequence of line segments in a picture box, first decide on the (X, Y) coordinates of each segment endpoint. Then, decide on line color and line pattern and set the corresponding properties for the picture box. Then, using Polyline to draw the segments is simple. And, as usual, the process is best illustrated using an example.

Quick Example 4 - Drawing Lines

1. Start a new application. Add a command button. Set the form's **ScaleMode** property to **Pixels**:
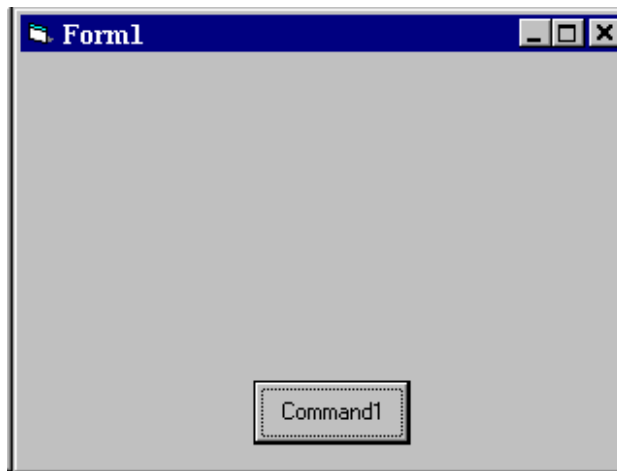
2. Set up the **General Declarations** area to include the user-defined variable (POINTAPI) and the **Declare** statement for **Polyline**. Also define a variable for the line endpoints:

```
Option Explicit
Private Type POINTAPI
 X As Long
 Y As Long
End Type
Private Declare Function Polyline Lib "gdi32" (ByVal hdc As
   Long, lpPoint As POINTAPI, ByVal nCount As Long) As Long

Dim V(20) As POINTAPI
Dim Index As Integer
```

3. Establish the **Form_MouseDown** event (saves the points):

```
Private Sub Form_MouseDown(Button As Integer, Shift As
   Integer, X As Single, Y As Single)
If Index = 0 Then Form1.Cls
Index = Index + 1
V(Index).X = X
V(Index).Y = Y
End Sub
```

4. Establish the **Command1_Click** event (draws the segments):

```
Private Sub Command1_Click()
Dim RtnValue As Integer
Form1.Cls
RtnValue = Polyline(Form1.hdc, V(1), Index)
Index = 0
End Sub
```

5. Run the application. Click on the form at different points, then click the command button to connect the 'clicked' points. Try different colors and line styles.

Drawing Polygons

*   We could try to use the Polyline function to draw closed regions, or polygons. One drawback to this method is that drawing filled regions is not possible. The DLL function **Polygon** allows us to draw any closed region defined by a set of (x, y) coordinate pairs.

*   Let's look at the **Declare** statement for **Polygon** (from the API Text Viewer):

    Private Declare Function Polygon Lib "gdi32" Alias "Polygon" (ByVal hdc As Long, lpPoint As POINTAPI, ByVal nCount As Long) As Long

    Note it has three arguments: **hdc** is the hDC handle of the region (Form or Picture Box) being drawn to, **lpPoint** is the first point in an array of points defining the vertices of the polygon - it is of type **POINTAPI**, and **nCount** is the number of points defining the enclosed region.

*   So, to draw a polygon in a picture box, first decide on the (X, Y) coordinates of each vertex in the polygon. Then, decide on line color, line pattern, fill color and fill pattern and set the corresponding properties for the picture box. Then, using Polygon to draw the shape is simple.

Quick Example 5 - Drawing Polygons

1. Start a new application and establish a form with the following controls: a picture box (**ScaleMode** set to **Pixels**), a control array of five option buttons, and a command button:
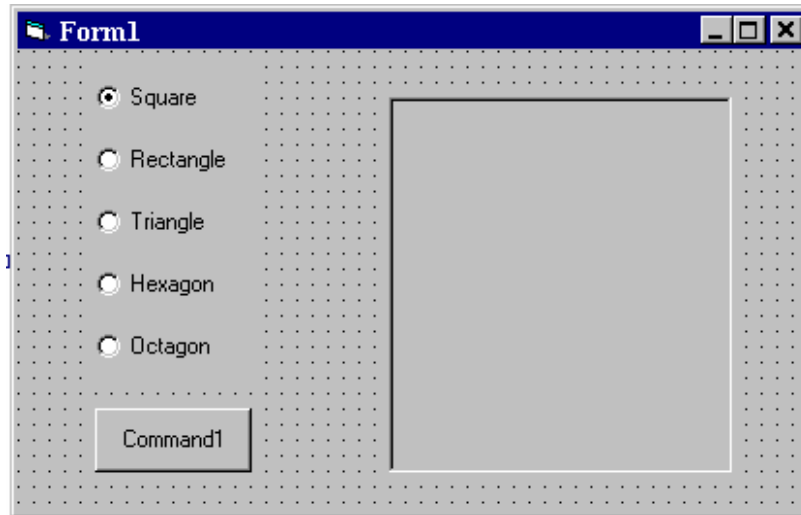
2. Set up the **General Declarations** area to include the user-defined variable (POINTAPI) and the **Declare** statement for **Polygon**:

```
Option Explicit
Private Type POINTAPI
 X As Long
 Y As Long
End Type
Private Declare Function Polygon Lib "gdi32" (ByVal hdc As
   Long, lpPoint As POINTAPI, ByVal nCount As Long) As Long
```

3. Establish the **Command1_Click** event:

```
Private Sub Command1_Click()
Dim I As Integer
For I = 0 To 4
 If Option1(I).Value = True Then
  Exit For
 End If
Next I
Picture1.Cls
Call Draw_Shape(Picture1, I)
End Sub
```

4. Set up a general procedure to draw a particular shape number (**PNum**) in a general control (**PBox**). This procedure can draw one of five shapes (0-Square, 1-Rectangle, 2-Triangle, 3-Hexagon, 4-Octagon). For each shape, it establishes some margin area (**DeltaX** and **DeltaY**) and then defines the vertices of the shape using the **V** array (a **POINTAPI** type variable).

```
Private Sub Draw_Shape(PBox As Control, PNum As Integer)
Dim V(1 To 8) As POINTAPI, Rtn As Long
Dim DeltaX As Integer, DeltaY As Integer
Select Case PNum
Case 0
'Square
  DeltaX = 0.05 * PBox.ScaleWidth
  DeltaY = 0.05 * PBox.ScaleHeight
  V(1).X = DeltaX: V(1).Y = DeltaY
  V(2).X = PBox.ScaleWidth - DeltaX: V(2).Y = V(1).Y
  V(3).X = V(2).X: V(3).Y = PBox.ScaleHeight - DeltaY
  V(4).X = V(1).X: V(4).Y = V(3).Y
  Rtn = Polygon(PBox.hdc, V(1), 4)
Case 1
'Rectangle
  DeltaX = 0.3 * PBox.ScaleWidth
  DeltaY = 0.05 * PBox.ScaleHeight
  V(1).X = DeltaX: V(1).Y = DeltaY
  V(2).X = PBox.ScaleWidth - DeltaX: V(2).Y = V(1).Y
  V(3).X = V(2).X: V(3).Y = PBox.ScaleHeight - DeltaY
  V(4).X = V(1).X: V(4).Y = V(3).Y
  Rtn = Polygon(PBox.hdc, V(1), 4)
Case 2
'Triangle
  DeltaX = 0.05 * PBox.ScaleWidth
  DeltaY = 0.05 * PBox.ScaleHeight
  V(1).X = DeltaX: V(1).Y = PBox.ScaleHeight - DeltaY
  V(2).X = 0.5 * PBox.ScaleWidth: V(2).Y = DeltaY
  V(3).X = PBox.ScaleWidth - DeltaX: V(3).Y = V(1).Y
  Rtn = Polygon(PBox.hdc, V(1), 3)
Case 3
'Hexagon
  DeltaX = 0.05 * PBox.ScaleWidth
  DeltaY = 0.05 * PBox.ScaleHeight
  V(1).X = DeltaX: V(1).Y = 0.5 * PBox.ScaleHeight
  V(2).X = 0.25 * PBox.ScaleWidth: V(2).Y = DeltaY
  V(3).X = 0.75 * PBox.ScaleWidth: V(3).Y = V(2).Y
  V(4).X = PBox.ScaleWidth - DeltaX: V(4).Y = V(1).Y
  V(5).X = V(3).X: V(5).Y = PBox.ScaleHeight - DeltaY
  V(6).X = V(2).X: V(6).Y = V(5).Y
  Rtn = Polygon(PBox.hdc, V(1), 6)
```

```
  Case 4
'Octagon
  DeltaX = 0.05 * PBox.ScaleWidth
  DeltaY = 0.05 * PBox.ScaleHeight
  V(1).X = DeltaX: V(1).Y = 0.3 * PBox.ScaleHeight
  V(2).X = 0.3 * PBox.ScaleWidth: V(2).Y = DeltaY
  V(3).X = 0.7 * PBox.ScaleWidth: V(3).Y = V(2).Y
  V(4).X = PBox.ScaleWidth - DeltaX: V(4).Y = V(1).Y
  V(5).X = V(4).X: V(5).Y = 0.7 * PBox.ScaleHeight
  V(6).X = V(3).X: V(6).Y = PBox.ScaleHeight - DeltaY
  V(7).X = V(2).X: V(7).Y = V(6).Y
  V(8).X = V(1).X: V(8).Y = V(5).Y
  Rtn = Polygon(PBox.hdc, V(1), 8)
End Select
End Sub
```

5. Run the application. Select a shape and click the command button to draw it. Play
    with the picture box properties to obtain different colors and fill patterns.

6. To see the importance of proper variable declarations when using DLL's and the API,
    make the two components (X and Y) in the POINTAPI variable of type Integer rather
    than Long. Rerun the program and see the strange results.

Sounds with DLL Calls - Other Beeps

- As seen in the above example and by perusing the Visual Basic literature, only one sound is available in Visual Basic - **Beep**. Not real exciting. By using available DLL's, we can add all kinds of sounds to our applications.

- A DLL routine like the Visual Basic **Beep** function is **MessageBeep.**   It also beeps the speaker but, with a sound card, you can hear different kinds of beeps. Message Beep has a single argument, that being an long integer that describes the type of beep you want.   MessageBeep returns a **long** integer. The usage syntax is:

    Dim BeepType As Long, RtnValue as Long
    .
    .
    .
    RtnValue = MessageBeep(BeepType)

- **BeepType** has five possible values. Sounds are related to the four possible icons available in the Message Box (these sounds are set from the Windows 95 control panel). The DLL constants available are:

    MB_ICONSTOP - Play sound associated with the critical icon
    MB_ICONEXCLAMATION - Play sound associated with the exclamation icon
    MB_ICONINFORM ATION - Play sound associated with the information icon
    MB_ICONQUESTION - Play sound associated with the question icon
    MB_OK - Play sound associated with no icon

Quick Example 6 - Adding Beeps to Message Box Displays

We can use MessageBeep to add beeps to our display of message boxes.

1. Start a new application. Add a text box and a command button.

2. Copy and paste the Declare statement for the **MessageBeep** function to the **General Declarations** area. Also, copy and paste the following seven constants (we need seven since some of the ones we use are equated to other constants):

```
Private Declare Function MessageBeep Lib "user32" (ByVal
  wType As Long) As Long
Private Const MB_ICONASTERISK = &H40&
Private Const MB_ICONEXCLAMATION = &H30&
Private Const MB_ICONHAND = &H10&
Private Const MB_ICONINFORMATION = MB_ICONASTERISK
Private Const MB_ICONSTOP = MB_ICONHAND
Private Const MB_ICONQUESTION = &H20&
Private Const MB_OK = &H0&
```

3. In the above constant definitions, you will have to change the word Public (which comes from the text viewer) with the word Private.

4. Use this code to the **Command1_Click** event.

```
Private Sub Command1_Click()
Dim BeepType As Long, RtnValue As Long
Select Case Val(Text1.Text)
Case 0
  BeepType = MB_OK
Case 1
  BeepType = MB_ICONINFORMATION
Case 2
  BeepType = MB_ICONEXCLAMATION
Case 3
  BeepType = MB_ICONQUESTION
Case 4
  BeepType = MB_ICONSTOP
End Select
RtnValue = MessageBeep(BeepType)
MsgBox "This is a test", BeepType, "Beep Test"
End Sub
```

5. Run the application. Enter values from 0 to 4 in the text box and click the command button. See if you get different beep sounds.

More Elaborate Sounds

* Beeps are nice, but many times you want to play more elaborate sounds. Most sounds you hear played in Windows applications are saved in **WAV** files (files with WAV extensions). These are the files formed when you record using one of the many sound recorder programs available.

* WAV files are easily played using DLL functions. There is more than one way to play such a file. We'll use the **sndPlaySound** function. This is a **long** function that requires two arguments, a **string** argument with the name of the WAV file and a **long** argument indicating how to play the sound. The usage syntax is:

    Dim WavFile As String, SndType as Long, RtnValue as Long
    .
    .
    .
    RtnValue = sndPlaysound(WavFile, SndType)

* **SndType** has many possible values. We'll just look at two:

    SND_SYNC - Sound is played to completion, then execution continues
    SND_ASYNC - Execution continues as sound is played

Quick Example 7 - Playing WAV Files

1. Start a new application. Add a command button and a common dialog box. Copy and paste the **sndPlaySound** Declare statement from the API Text Viewer program into your application. Also copy the **SND_SYNC** and **SND_ASYNC** constants. When done copying and making necessary scope modifications, you should have:

```
Private Declare Function sndPlaySound Lib "winmm.dll" Alias
   "sndPlaySoundA" (ByVal lpszSoundName As String, ByVal
   uFlags As Long) As Long
Private Const SND_ASYNC = &H1
Private Const SND_SYNC = &H0
```

2. Add this code to the **Command1_Click** procedure:

```
Private Sub Command1_Click()
Dim RtnVal As Integer
'Get name of .wav file to play
CommonDialog1.Filter = "Sound Files|*.wav"
CommonDialog1.ShowOpen
RtnVal = sndPlaySound(CommonDialog1.filename, SND_SYNC)
End Sub
```

3. Run the application. Find a WAV file and listen to the lovely results.

Playing Sounds Quickly

*   Using the sndPlaySound function in the previous example requires first opening a file, then playing the sound. If you want quick sounds, say in games, the loading procedure could slow you down quite a bit. What would be nice would be to have a sound file 'saved' in some format that could be played quickly. We can do that!

*   What we will do is open the sound file (say in the Form_Load procedure) and write the file to a string variable. Then, we just use this string variable in place of the file name in the **sndPlaySound** argument list. We also need to 'Or' the **SndType** argument with the constant **SND_MEMORY** (this tells sndPlaySound we are playing a sound from memory as opposed to a WAV file). This technique is borrowed from "Black Art of Visual Basic Game Programming," by Mark Pruett, published by The Waite Group in 1995. Sounds played using this technique must be short sounds (less than 5 seconds) or mysterious results could happen.

Quick Example 8 - Playing Sounds Quickly

We'll write some code to play a quick 'bonk' sound.

1. Start a new application. Add a command button.

2. Copy and paste the **sndPlaySound** Declare statement and the two needed constants (see Quick Example 4). Declare a variable (BongSound) for the sound file. Add **SND_MEMORY** to the constants declarations. The two added statements are:

```
Dim BongSound As String
Private Const SND_MEMORY = &H4
```

3. Add the following general function, **StoreSound**, that will copy a WAV file into a string variable:

```
Private Function StoreSound(ByVal FileName) As String
'-------------------------------------------------------
' Load a sound file into a string variable.
' Taken from:
'    Mark Pruett
'    Black Art of Visual Basic Game Programming
'    The Waite Group, 1995
'-------------------------------------------------------
Dim Buffer As String
Dim F As Integer
Dim SoundBuffer As String
On Error GoTo NoiseGet_Error
Buffer = Space$(1024)
SoundBuffer = ""
F = FreeFile
Open FileName For Binary As F
Do While Not EOF(F)
  Get #F, , Buffer
  SoundBuffer = SoundBuffer & Buffer
Loop
Close F
StoreSound = Trim(SoundBuffer)
Exit Function
NoiseGet_Error:
SoundBuffer = ""
Exit Function
End Function
```

4. Write the following **Form_Load** procedure:

```
Private Sub Form_Load()
BongSound = StoreSound("bong.wav")
End Sub
```

5. Use this as the **Command1_Click** procedure:

```
Private Sub Command1_Click()
Call sndPlaySound(BongSound, SND_SYNC Or SND_MEMORY)
End Sub
```

6. Make sure the sound (**BONK.WAV**) is in the same directory as your application. Run the application. Each time you click the command button, you should hear a bonk!

Fun With Graphics

*   One of the biggest uses of the API is for graphics, whether it be background scrolling, sprite animation, or many other special effects. A very versatile API function is **BitBlt**, which stands for **Bit Block Transfer**. It is used to copy a section of one bitmap from one place (the source) to another (the destination).

*   Let's look at the Declaration statement for BitBlt (from the API Text Viewer):

    PrivateDeclare Function BitBlt Lib "gdi32" Alias "BitBlt"

    (ByVal hDestDC As Long,
    ByVal x As Long,
    ByVal y As Long,
    ByVal nWidth As Long,
    ByVal nHeight As Long,
    ByVal hSrcDC As Long,
    ByVal xSrc As Long,
    ByVal ySrc As Long,
    ByVal dwRop As Long) As
    Long

    Lots of stuff here, but fairly straightforward. **hDestDC** is the device context handle, or hDC of the destination bitmap. The coordinate pair (**X, Y**) specifies the upper left corner in the destination bitmap to copy the source. The parameters **nWidth** and **nHeight** are, respectively, the width and height of the copied bitmap. **hSrcDC** is the device context handle for the source bitmap and (**Xsrc, Ysrc**) is the upper left corner of the region of the source bitmap being copied. Finally, **dwRop** is a constant that defines how the bitmap is to be copied. We will do a direct copy or set dwRop equal

to the constant **SRCCOPY**. The BitBlt function expects all geometric units to be pixels.

• BitBlt returns an long integer value -- we won't be concerned with its use right now. So, the syntax for using BitBlt is:

>   Dim RtnValue As Long
>
>   .
>   .
>   RtnValue = BitBlt(Dest.hDC, X, Y, Width, Height,
>   Src.hDC, Xsrc, Ysrc, SRCCOPY)

This function call takes the Src bitmap, located at (Xsrc, Ysrc), with width Width and height Height, and copies it directly to the Dest bitmap at (X, Y).

Quick Example 9 - Bouncing Ball With Sound!

We'll build an application with a ball bouncing from the top to the bottom as an illustration of the use of BitBlt.

1. Start a new application. Add two picture boxes, a shape (inside the smaller picture box), a timer control, and a command button.:
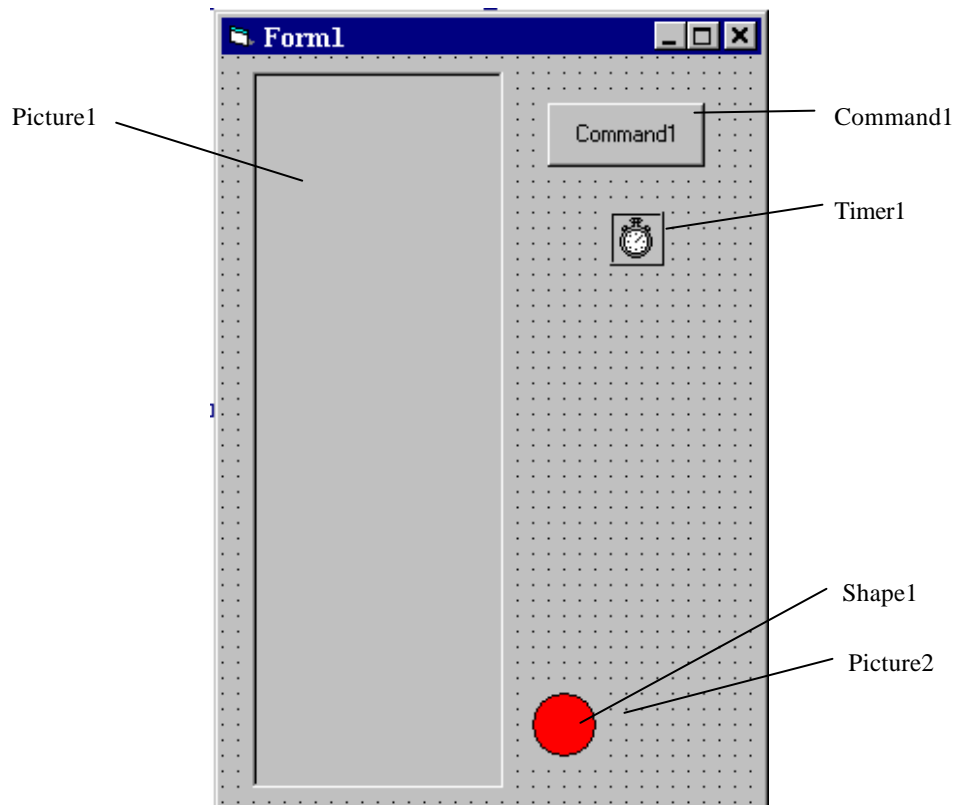
2. For Picture1 (the destination), set the **ScaleMode** property to **Pixel**. For Shape1, set the **FillStyle** property to **Solid**, the **Shape** property to **Circle**, and choose a **FillColor**. For Picture2 (the ball), set the **ScaleMode** property to **Pixel** and the **BorderStyle** property to **None**. For Timer1, set the **Enabled** property to **False** and the **Interval** property to **100**.

3. Copy and paste constants for the **BitBlt** Declare statement and constants. Also copy and paste the necessary sndPlaySound statements and declare some variables. The general declarations area is thus:

```
Option Explicit
Dim BongSound As String
Dim BallY As Long, BallDir As Integer
Private Declare Function sndPlaySound Lib "winmm.dll" Alias
    "sndPlaySoundA" (ByVal lpszSoundName As String, ByVal
    uFlags As Long) As Long
Private Const SND_ASYNC = &H1
Private Const SND_SYNC = &H0
Private Const SND_MEMORY = &H4
Private Declare Function BitBlt Lib "gdi32" (ByVal hDestDC
    As Long, ByVal x As Long, ByVal y As Long, ByVal nWidth
    As Long, ByVal nHeight As Long, ByVal hSrcDC As Long,
    ByVal xSrc As Long, ByVal ySrc As Long, ByVal dwRop As
    Long) As Long
Private Const SRCCOPY = &HCC0020
```

4. Add a **Form_Load** procedure:

```
Private Sub Form_Load()
BallY = 0
BallDir = 1
BongSound = StoreSound("bong.wav")
End Sub
```

5. Write a **Command1_Click** event procedure to toggle the timer:

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

6. The **Timer1_Timer** event controls the bouncing ball position:

```
Private Sub Timer1_Timer()
Static BallY As Long
Dim RtnValue As Long
Picture1.Cls
BallY = BallY + BallDir * Picture1.ScaleHeight / 50
If BallY < 0 Then
 BallY = 0
 BallDir = 1
 Call sndPlaySound(BongSound, SND_ASYNC Or SND_MEMORY)
ElseIf BallY + Picture2.ScaleHeight > Picture1.ScaleHeight
   Then
 BallY = Picture1.ScaleHeight - Picture2.ScaleHeight
 BallDir = -1
 Call sndPlaySound(BongSound, SND_ASYNC Or SND_MEMORY)
End If
RtnValue = BitBlt(Picture1.hDC, CLng(0.5 *
   (Picture1.ScaleWidth - Picture2.ScaleWidth)), _
BallY, CLng(Picture2.ScaleWidth),
   CLng(Picture2.ScaleHeight), Picture2.hDC, CLng(0),
   CLng(0), SRCCOPY)
End Sub
```

7. We also need to make sure we include the **StoreSound** procedure from the last example so we can hear the bong when the ball bounces.

8. Once everything is together, run it and follow the bouncing ball!

Flicker Free Animation

• You may notice in the bouncing ball example that there is a bit of flicker as it bounces. Much smoother animation can be achieved with just a couple of changes.

• The idea behind so-called flicker free animation is to always work with two picture boxes for the animation (each with the same properties, but one is visible and one is not). The non-visible picture box is our working area where everything is positioned where it needs to be at each time point in the animation sequence. Once everything is properly positioned, we then copy (using BitBlt) the entire non-visible picture box into the visible picture box. The results are quite nice.

Quick Example 10 - Flicker Free Animation

We modify the previous example to make it flicker free.

1. Change the **Index** property of Picture1 to **0** (zero). This makes it a control array which we can make a copy of. Once this copy is made. Picture1(0) will be our visible area and Picture1(1) will be our non-visible, working area.

2. Add these statements to the **Form_Load** procedure to create Picture1(1):

```
Load Picture1(1)
Picture1(1).AutoRedraw = True
```

3. Make the italicized changes to the **Timer1_Timer** event. The ball is now drawn to Picture1(1). Once drawn, the last statement in the procedure copies Picture1(1) to Picture1(0).

```
Private Sub Timer1_Timer()
Static BallY As Long
Dim RtnValue As Long
Picture1(1).Cls
BallY = BallY + BallDir * Picture1(1).ScaleHeight / 50
If BallY < 0 Then
  BallY = 0
  BallDir = 1
  Call sndPlaySound(BongSound, SND_ASYNC Or SND_MEMORY)
ElseIf BallY + Picture2.ScaleHeight >
  Picture1(1).ScaleHeight Then
  BallY = Picture1(1).ScaleHeight - Picture2.ScaleHeight
  BallDir = -1
  Call sndPlaySound(BongSound, SND_ASYNC Or SND_MEMORY)
End If
RtnValue = BitBlt(Picture1(1).hDC, CLng(0.5 *
  (Picture1(1).ScaleWidth - Picture2.ScaleWidth)), _
          BallY, CLng(Picture2.ScaleWidth),
  CLng(Picture2.ScaleHeight), Picture2.hDC, CLng(0),
  CLng(0), SRCCOPY)
RtnValue = BitBlt(Picture1(0).hDC, CLng(0), CLng(0),
  CLng(Picture1(1).ScaleWidth),
  CLng(Picture1(1).ScaleHeight), Picture1(1).hDC, CLng(0),
  CLng(0), SRCCOPY)
End Sub
```

4. Run the application and you should notice the smoother ball motion.

Quick Example 11 - Horizontally Scrolling Background

Most action arcade games employ scrolling backgrounds. What they really use is one long background picture that wraps around itself. We can use the BitBlt API function to generate such a background. Here's the idea. Say we have one long bitmap of some background (here, an underseascape created in a paint program and saved as a bitmap file):
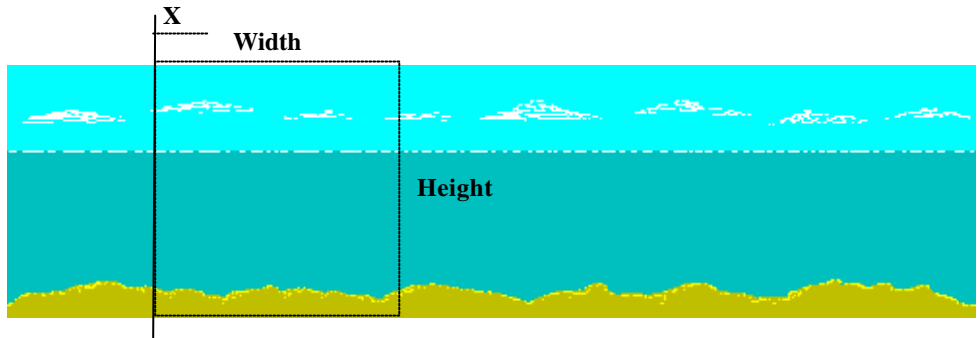
At each program cycle, we copy a bitmap of the size shown to a destination location. As X increases, the background appears to scroll. Note as X reaches the end of this source bitmap, we need to copy a little of both ends to the destination bitmap.

1. Start a new application. Add a horizontal scroll bar, two picture boxes, and a timer control. Your form should resemble:
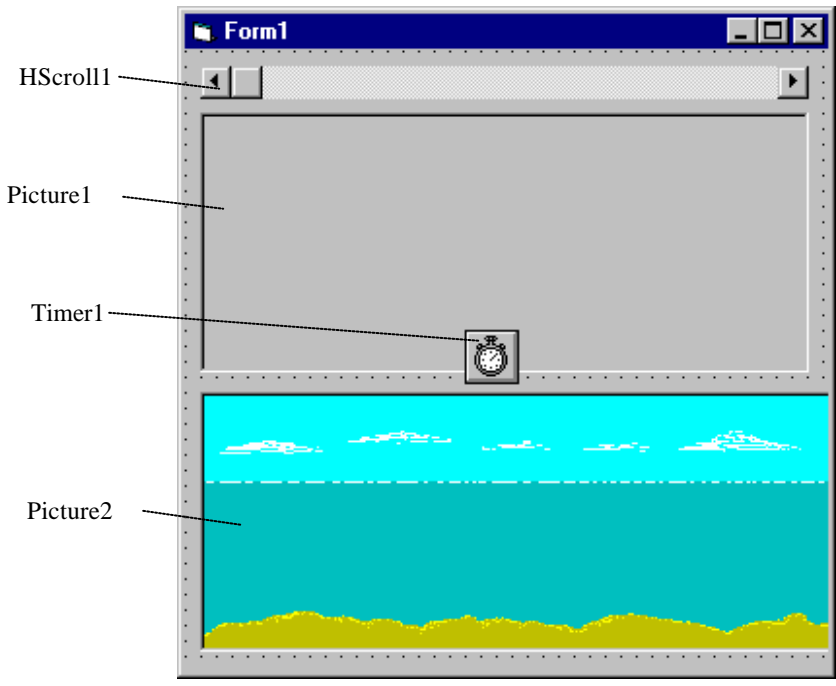
2. For Picture1 (the destination), set the **ScaleMode** property to **Pixel**. For Picture2, set **ScaleMode** to **Pixel**, **AutoSize** and **AutoRedraw** to **True**, and **Picture** to **Undrsea1**.**bmp** (provided on class disk). Set Picture1 **Height** property to the same as Picture2. Set Timer1 **Interval** property to **50**. Set the Hscroll1 **Max** property to **20** and **LargeChange** property to **2**. After setting properties, resize the form so Picture2 does not appear.

3. Copy and paste the **BitBlt** Declare statement from the API text viewer. Also, copy the **SRCCOPY** constant:

4. Attach the following code to the **Timer1_Timer** event:

```
Private Sub Timer1_Timer()
Static x As Long
Dim AWidth As Long
Dim RC As Long
'Find next location on Picture2
x = x + HScroll1.Value
If x > Picture2.ScaleWidth Then x = 0
'When x is near right edge, we need to copy
'two segments of Picture2 into Picture1
If x > (Picture2.ScaleWidth - Picture1.ScaleWidth) Then
  AWidth = Picture2.ScaleWidth - x
RC = BitBlt(Picture1.hDC, CLng(0), CLng(0), AWidth,
  CLng(Picture2.ScaleHeight), Picture2.hDC, x, CLng(0),
  SRCCOPY)
RC = BitBlt(Picture1.hDC, AWidth, CLng(0),
  CLng(Picture1.ScaleWidth - AWidth),
  CLng(Picture2.ScaleHeight), Picture2.hDC, CLng(0),
  CLng(0), SRCCOPY)
Else
RC = BitBlt(Picture1.hDC, CLng(0), CLng(0),
  CLng(Picture1.ScaleWidth), CLng(Picture2.ScaleHeight),
  Picture2.hDC, x, CLng(0), SRCCOPY)
End If
End Sub
```

5. Run the application. The scroll bar is used to control the speed of the scrolling (the amount X increases each time a timer event occurs).

A Bit of Multimedia

- The computer of the 90's is the **multimedia** computer (graphics, sounds, video). Windows provides a set of rich multimedia functions we can use in our Visual Basic applications. Of course, to have access to this power, we use the API. We'll briefly look at using the API to play video files with the **AVI** (audio - visual interlaced) extension.

- In order to play AVI files, your computer needs to have software such as Video for Windows (from Microsoft) or QuickTime for Windows (from Apple) loaded on your machine. When a video is played from Visual Basic, a new window is opened with the title of the video file shown. When the video is complete, the window is automatically closed.

- The DLL function **mciExecute** is used to play video files (note it will also play WAV files). The syntax for using this function is:

  Dim RtnValue as Long
    .
    .
  RtnValue = mciExecute (*Command*)

  where *Command* is a string argument consisting of the keyword '**Play'** concatenated with the complete pathname to the desired file.

Quick Example 12 - Multime dia Sound and Video

1. Start a new application. Add a command button and a common dialog box. Copy and paste the **mciExecute** Declare statement from the API Text Viewer program into your application. It should read:

```
Private Declare Function mciExecute Lib "winmm.dll" (ByVal
    lpstrCommand As String) As Long
```

2. Add this code to the **Command1_Click** procedure:

```
Private Sub Command1_Click()
Dim RtnVal As Long
'Get name of .avi file to play
CommonDialog1.Filter = "Video Files|*.avi"
CommonDialog1.ShowOpen
RtnVal = mciExecute("play " + CommonDialog1.filename)
End Sub
```

3. Run the application. Find a AVI file and see and hear the lovely results.

Exercise 9

The Original Video Game - Pong!

In the early 1970's, Nolan Bushnell began the video game revolution with Atari's Pong game -- a very simple Ping-Pong kind of game. Try to replicate this game using Visual Basic. In the game, a ball bounces from one end of a court to another, bouncing off side walls. Players try to deflect the ball at each end using a controllable paddle. Use sounds where appropriate (look at my solution for some useful DLL's for sound).

My solution freely borrows code and techniques from several reference sources. The primary source is a book on game programming, by Mark Pruett, entitled "Black Art of Visual Basic Game Programming," published by The Waite Group in 1995. In my simple game, the left paddle is controlled with the A and Z keys on the keyboard, while the right paddle is controlled with the K and M keys.

My Solution:

Form:
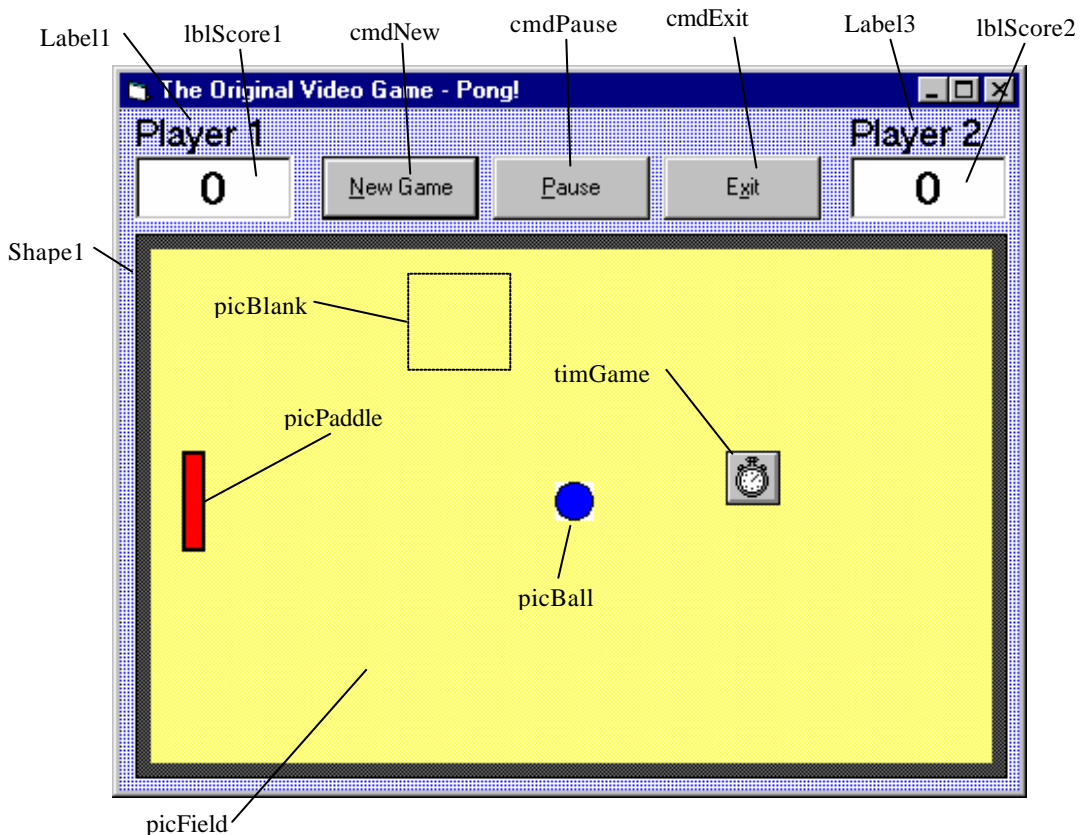
Properties:

Form **frmPong**:
  BackColor = &H00FFC0C0& (Light blue)
  Caption = The Original Video Game - Pong!

Timer **timGame**:
  Enabled = False
  Interval = 25 (may need different values for different machines)

PictureBox **picPaddle**:
  Appearance = Flat
  AutoRedraw = True
  AutoSize = True
  Picture = paddle.bmp
  ScaleMode = Pixel
  Visible = False

CommandButton **cmdPause**:
  Caption = &Pause
  Enabled = 0 'False

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

CommandButton **cmdNew**:
  Caption = &New Game
  Default = True

PictureBox **picField**:
  BackColor = &H0080FFFF& (Light yellow)
  BorderStyle = None
  FontName = MS Sans Serif
  FontSize = 24
  ForeColor = &H000000FF& (Red)
  ScaleMode = Pixel

PictureBox **picBlank**:
  Appearance = Flat
  AutoRedraw = True
  BackColor = &H0080FFFF& (Light yellow)
. BorderStyle = None
  FillStyle = Solid
  Visible = False

PictureBox **picBall**:
      Appearance = Flat
      AutoRedraw = True
      AutoSize = True
      BorderStyle = None
      Picture = ball.bmp
      ScaleMode = Pixel
      Visible = False

Shape **Shape1**:
      BackColor = &H00404040& (Black)
      BackStyle = Opaque

Label **lblScore2**:
      Alignment = Center
      BackColor = &H00FFFFFF& (White)
      BorderStyle = Fixed Single
      Caption = 0
      FontName = MS Sans Serif
      FontBold = True
      FontSize = 18

Label **Label3**:
      BackColor = &H00FFC0C0& (Light blue)
      Caption = Player 2
      FontName = MS Sans Serif
      FontSize = 13.5

Label **lblScore1**:
      Alignment = Center
      BackColor = &H00FFFFFF& (White)
      BorderStyle = Fixed Single
      Caption = 0
      FontName = MS Sans Serif
      FontBold = True
      FontSize = 18

Label **Label1**:
      BackColor = &H00FFC0C0& (Light blue)
      Caption = Player 1
      FontName = MS Sans Serif
      FontSize = 13.5

Code:

General Declarations:

```
Option Explicit
'Sound file strings
Dim wavPaddleHit As String
Dim wavWall As String
Dim wavMissed As String
'A user-defined variable to position bitmaps
Private Type tBitMap
  Left As Long
  Top As Long
  Right As Long
  Bottom As Long
  Width As Long
  Height As Long
End Type
'Ball information
Dim bmpBall As tBitMap
Dim XStart As Long, YStart As Long
Dim XSpeed As Long, YSpeed As Long
Dim SpeedUnit As Long
Dim XDir As Long, YDir As Long
'Paddle information
Dim bmpPaddle1 As tBitMap, bmpPaddle2 As tBitMap
Dim YStartPaddle1 As Long, YStartPaddle2 As Long
Dim XPaddle1 As Long, XPaddle2 As Long
Dim PaddleIncrement As Long

Dim Score1 As Integer, Score2 As Integer
Dim Paused As Boolean
'Number of points to win
Const WIN = 10
'Number of bounces before speed increases
Const BOUNCE = 10
Dim NumBounce As Integer
'API Functions and constants
Private Declare Function BitBlt Lib "gdi32" (ByVal hDestDC
As Long, ByVal x As Long, ByVal y As Long, ByVal nWidth As
Long, ByVal nHeight As Long, ByVal hSrcDC As Long, ByVal
xSrc As Long, ByVal ySrc As Long, ByVal dwRop As Long) As
Long
Const SRCCOPY = &HCC0020 ' (DWORD) dest = source
Private Declare Function sndPlaySound Lib "winmm.dll" Alias
"sndPlaySoundA" (ByVal lpszSoundName As String, ByVal
uFlags As Long) As Long
```

```
Private Declare Function sndStopSound Lib "winmm.dll" Alias
"sndPlaySoundA" (ByVal lpszNull As String, ByVal uFlags As
Long) As Long
Const SND_ASYNC = &H1
Const SND_SYNC = &H0
Const SND_MEMORY = &H4
Const SND_LOOP = &H8
Const SND_NOSTOP = &H10
' Windows API rectangle function
Private Declare Function IntersectRect Lib "user32"
(lpDestRect As tBitMap, lpSrc1Rect As tBitMap, lpSrc2Rect
As tBitMap) As Long
```

NoiseGet General Function:

```
Function NoiseGet(ByVal FileName) As String
'------------------------------------------------------------
--
' Load a sound file into a string variable.
' Taken from:
'    Mark Pruett
'    Black Art of Visual Basic Game Programming
'    The Waite Group, 1995
'------------------------------------------------------------
--
Dim buffer As String
Dim f As Integer
Dim SoundBuffer As String
    On Error GoTo NoiseGet_Error
    buffer = Space$(1024)
    SoundBuffer = ""
    f = FreeFile
    Open FileName For Binary As f
    Do While Not EOF(f)
        Get #f, , buffer        ' Load in 1K chunks
        SoundBuffer = SoundBuffer & buffer
    Loop
    Close f
    NoiseGet = Trim$(SoundBuffer)
Exit Function
NoiseGet_Error:
    SoundBuffer = ""
    Exit Function
End Function
```

NoisePlay General Procedure:

```
Sub NoisePlay(SoundBuffer As String, ByVal PlayMode As
Integer)
'------------------------------------------------------------
--
' Plays a sound previously loaded into memory with function
' NoiseGet().
' Taken from:
'    Mark Pruett
'    Black Art of Visual Basic Game Programming
'    The Waite Group, 1995
'------------------------------------------------------------
--
Dim retcode As Integer
    If SoundBuffer = "" Then Exit Sub
' Stop any sound that may currently be playing.
    retcode = sndStopSound(0, SND_ASYNC)
' PlayMode should be SND_SYNC or SND_ASYNC
    retcode = sndPlaySound(ByVal SoundBuffer, PlayMode Or
SND_MEMORY)
End Sub
```

Bitmap_Move General Procedure:

```
Private Sub Bitmap_Move(ABitMap As tBitMap, ByVal NewLeft
As Integer, ByVal NewTop As Integer, SourcePicture As
PictureBox)
' Move bitmap from one location to the next
' Modified from:
'    Mark Pruett
'    Black Art of Visual Basic Game Programming
'    The Waite Group, 1995
Dim RtnValue As Integer
'First erase at old location
RtnValue = BitBlt(picField.hDC, ABitMap.Left, ABitMap.Top,
ABitMap.Width, ABitMap.Height, picBlank.hDC, 0, 0, SRCCOPY)
'Then, establish and redraw at new location
ABitMap.Left = NewLeft
ABitMap.Top = NewTop
RtnValue = BitBlt(picField.hDC, ABitMap.Left, ABitMap.Top,
ABitMap.Width, ABitMap.Height, SourcePicture.hDC, 0, 0,
SRCCOPY)
End Sub
```

ResetPaddles General Procedure:

```
Private Sub ResetPaddles()
'Reposition paddles
bmpPaddle1.Top = YStartPaddle1
bmpPaddle2.Top = YStartPaddle2
Call Bitmap_Move(bmpPaddle1, bmpPaddle1.Left,
bmpPaddle1.Top, picPaddle)
Call Bitmap_Move(bmpPaddle2, bmpPaddle2.Left,
bmpPaddle2.Top, picPaddle)
End Sub
```

Update_Score General Procedure:

```
Private Sub Update_Score(Player As Integer)
Dim Winner As Integer, RtnValue As Integer
Winner = 0
'Update scores and see if game over
timGame.Enabled = False
Call NoisePlay(wavMissed, SND_SYNC)
Select Case Player
Case 1
  Score2 = Score2 + 1
  lblScore2.Caption = Format(Score2, "#0")
  lblScore2.Refresh
  If Score2 = WIN Then Winner = 2
Case 2
  Score1 = Score1 + 1
  lblScore1.Caption = Format(Score1, "#0")
  lblScore1.Refresh
  If Score1 = WIN Then Winner = 1
End Select
If Winner = 0 Then
  Call ResetBall
  timGame.Enabled = True
Else
  cmdNew.Enabled = False
  cmdPause.Enabled = False
  cmdExit.Enabled = False
  RtnValue = sndPlaySound(App.Path + "\cheering.wav",
SND_SYNC)
  picField.CurrentX = 0.5 * (picField.ScaleWidth -
picField.TextWidth("Game Over"))
  picField.CurrentY = 0.5 * picField.ScaleHeight -
picField.TextHeight("Game Over")
  picField.Print "Game Over"
  cmdNew.Enabled = True
```

```
  cmdExit.Enabled = True
End If
End Sub
```

ResetBall General Procedure:

```
Sub ResetBall()
'Set random directions
XDir = 2 * Int(2 * Rnd) - 1
YDir = 2 * Int(2 * Rnd) - 1
bmpBall.Left = XStart
bmpBall.Top = YStart
End Sub
```

cmdExit_Click Event:

```
Private Sub cmdExit_Click()
'End game
End
End Sub
```

cmdNew Click Event:

```
Private Sub cmdNew_Click()
'New game code
'Reset scores
lblScore1.Caption = "0"
lblScore2.Caption = "0"
Score1 = 0
Score2 = 0
'Reset ball
SpeedUnit = 1
XSpeed = 5 * SpeedUnit
YSpeed = XSpeed
Call ResetBall
'Reset paddles
picField.Cls
PaddleIncrement = 5
NumBounce = 0
Call ResetPaddles
cmdPause.Enabled = True
timGame.Enabled = True
picField.SetFocus
End Sub
```

Collided General Function:

```
Private Function Collided(A As tBitMap, B As tBitMap) As
Integer
'---------------------------------------------------
' Check if the two rectangles (bitmaps) intersect,
' using the IntersectRect API call.
' Taken from:
'    Mark Pruett
'    Black Art of Visual Basic Game Programming
'    The Waite Group, 1995
'---------------------------------------------------

' Although we won't use it, we need a result
' rectangle to pass to the API routine.
Dim ResultRect As tBitMap

    ' Calculate the right and bottoms of rectangles needed
by the API call.
    A.Right = A.Left + A.Width - 1
    A.Bottom = A.Top + A.Height - 1

    B.Right = B.Left + B.Width - 1
    B.Bottom = B.Top + B.Height - 1

    ' IntersectRect will only return 0 (false) if the
    ' two rectangles do NOT intersect.
    Collided = IntersectRect(ResultRect, A, B)
End Function
```

cmdPause Click Event:

```
Private Sub cmdPause_Click()
If Not (Paused) Then
  timGame.Enabled = False
  cmdNew.Enabled = False
  Paused = True
  cmdPause.Caption = "&UnPause"
Else
  timGame.Enabled = True
  cmdNew.Enabled = True
  Paused = False
  cmdPause.Caption = "&Pause"
End If
picField.SetFocus
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
Randomize Timer
'Place from at middle of screen
frmPong.Left = 0.5 * (Screen.Width - frmPong.Width)
frmPong.Top = 0.5 * (Screen.Height - frmPong.Height)
'Load sound files into strings from fast access
wavPaddleHit = NoiseGet(App.Path + "\paddle.wav")
wavMissed = NoiseGet(App.Path + "\missed.wav")
wavWall = NoiseGet(App.Path + "\wallhit.wav")
'Initialize ball and paddle locations
XStart = 0.5 * (picField.ScaleWidth - picBall.ScaleWidth)
YStart = 0.5 * (picField.ScaleHeight - picBall.ScaleHeight)
XPaddle1 = 5
XPaddle2 = picField.ScaleWidth - picPaddle.ScaleWidth - 5
YStartPaddle1 = 0.5 * (picField.ScaleHeight -
picPaddle.ScaleHeight)
YStartPaddle2 = YStartPaddle1
'Get ball dimensions
bmpBall.Left = XStart
bmpBall.Top = YStart
bmpBall.Width = picBall.ScaleWidth
bmpBall.Height = picBall.ScaleHeight
'Get paddle dimensions
bmpPaddle1.Left = XPaddle1
bmpPaddle1.Top = YStartPaddle1
bmpPaddle1.Width = picPaddle.ScaleWidth
bmpPaddle1.Height = picPaddle.ScaleHeight
bmpPaddle2.Left = XPaddle2
bmpPaddle2.Top = YStartPaddle2
bmpPaddle2.Width = picPaddle.ScaleWidth
bmpPaddle2.Height = picPaddle.ScaleHeight
'Get ready to play
Paused = False
frmPong.Show
Call ResetPaddles
End Sub
```

picField KeyDown Event:

```
Private Sub picField_KeyDown(KeyCode As Integer, Shift As
Integer)
Select Case KeyCode
'Player 1 Motion
Case vbKeyA
  If (bmpPaddle1.Top - PaddleIncrement) > 0 Then
    Call Bitmap_Move(bmpPaddle1, bmpPaddle1.Left,
bmpPaddle1.Top - PaddleIncrement, picPaddle)
  End If
Case vbKeyZ
  If (bmpPaddle1.Top + bmpPaddle1.Height + PaddleIncrement)
< picField.ScaleHeight Then
    Call Bitmap_Move(bmpPaddle1, bmpPaddle1.Left,
bmpPaddle1.Top + PaddleIncrement, picPaddle)
  End If
'Player 2 Motion
Case vbKeyK
  If (bmpPaddle2.Top - PaddleIncrement) > 0 Then
    Call Bitmap_Move(bmpPaddle2, bmpPaddle2.Left,
bmpPaddle2.Top - PaddleIncrement, picPaddle)
  End If
Case vbKeyM
  If (bmpPaddle2.Top + bmpPaddle2.Height + PaddleIncrement)
< picField.ScaleHeight Then
    Call Bitmap_Move(bmpPaddle2, bmpPaddle2.Left,
bmpPaddle2.Top + PaddleIncrement, picPaddle)
  End If
End Select
End Sub
```

timGame Timer Event:

```
Private Sub timGame_Timer()
'Main routine
Dim XInc As Integer, YInc As Integer
Dim Collision1 As Integer, Collision2 As Integer, Collision
As Integer
Static Previous As Integer
'If paused, do nothing
If Paused Then Exit Sub
'Determine ball motion increments
XInc = XDir * XSpeed
YInc = YDir * YSpeed
'Ball hits top wall
```

```
If (bmpBall.Top + YInc) < 0 Then
  YDir = -YDir
  YInc = YDir * YSpeed
  Call NoisePlay(wavWall, SND_ASYNC)
End If
'Ball hits bottom wall
If (bmpBall.Top + bmpBall.Height + YInc) >
picField.ScaleHeight Then
  YDir = -YDir
  YInc = YDir * YSpeed
  Call NoisePlay(wavWall, SND_ASYNC)
End If
'Ball goes past left wall - Player 2 scores
If (bmpBall.Left) > picField.ScaleWidth Then
  Call Update_Score(2)
End If
'Ball goes past right wall - Player 1 scores
If (bmpBall.Left + bmpBall.Width) < 0 Then
  Call Update_Score(1)
End If
'Check if either paddle and ball collided
Collision1 = Collided(bmpBall, bmpPaddle1)
Collision2 = Collided(bmpBall, bmpPaddle2)
'Move ball
Call Bitmap_Move(bmpBall, bmpBall.Left + XInc, bmpBall.Top
+ YInc, picBall)
'If paddle hit, redraw paddle
If Collision1 Then
  Call Bitmap_Move(bmpPaddle1, bmpPaddle1.Left,
bmpPaddle1.Top, picPaddle)
  Collision = Collision1
ElseIf Collision2 Then
  Call Bitmap_Move(bmpPaddle2, bmpPaddle2.Left,
bmpPaddle2.Top, picPaddle)
  Collision = Collision2
End If
'If we hit a paddle, change ball direction
If Collision And (Not Previous) Then
  NumBounce = NumBounce + 1
  If NumBounce = BOUNCE Then
    NumBounce = 0
    XSpeed = XSpeed + SpeedUnit
    YSpeed = YSpeed + SpeedUnit
  End If
  XDir = -XDir
  Call NoisePlay(wavPaddleHit, SND_ASYNC)
End If
```

```
Previous = Collision
End Sub
```

Learn Visual Basic 6.0

10. Other Visual Basic Topics

Review and Preview

- In this last class, we look at a lot of relatively unrelated topics - a Visual Basic playground. We'll cover lots of things, each with enough detail to allow you, as a now-experienced Visual Basic programmer, to learn more about the topics that interest you.

Custom Controls

- A **custom control** is an extension to the standard Visual Basic toolbox. You use custom controls just as you would any other control. In fact, you've used (or at least seen) custom controls before. The **common dialog** box, the **DBList** box, the **DBCombo** box, and the **DBGrid** tool, are all examples of custom controls. Custom controls can be used to add some really cool features to your applications.

- Custom controls are also referred to as **ActiveX** controls. ActiveX is a technology newly introduced by Microsoft to describe what used to be known as **OLE Automation**. Prior to Visual Basic 5.0, the only way to create your own controls was to use C or C++. Now, with ActiveX technology, you can create your own controls knowing only Visual Basic! Of course, this would be a course by itself (and is).

- To use a custom control, you must load it into the toolbox. To do this, choose **Components** from the Visual Basic **Project** menu. The Components (custom controls) dialog box is displayed.
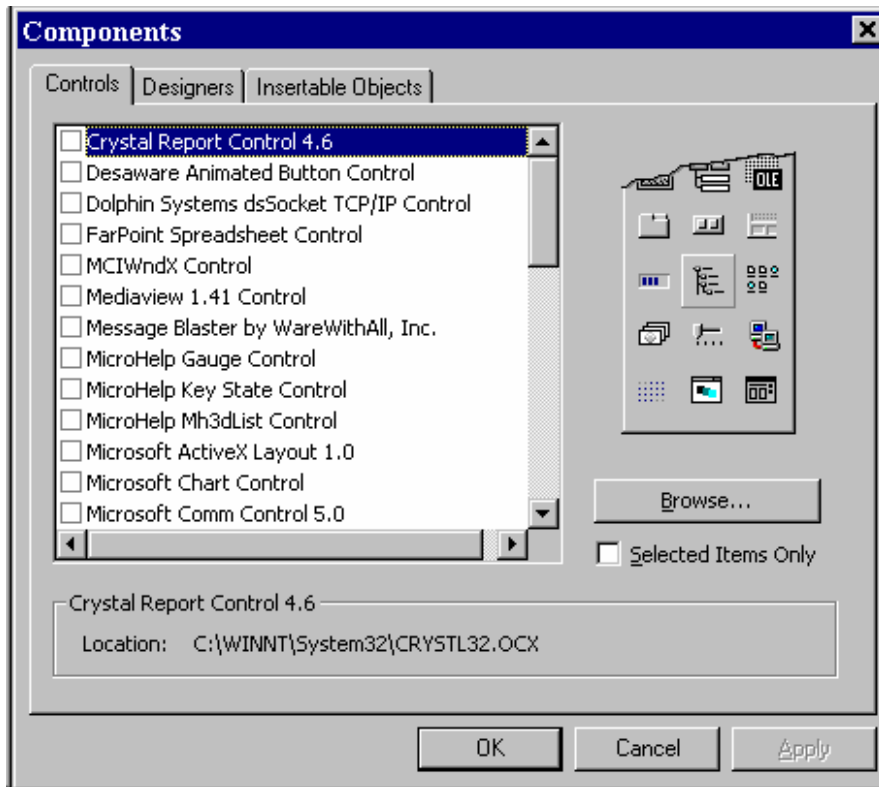
- To add a control, select the check box next to the desired selection. When done, choose **OK** and the selected controls will now appear in the toolbox.

- Each custom control has its own set of **properties**, **events**, and **methods**. The best reference for each control is the *Microsoft Visual Basic Component Tools Guide* manual that comes with Visual Basic 6.0. And, each tool also features on-line help.

- Here, we'll look at several custom controls and brief examples of their usage. And, we'll give some of the more important and unique properties, events, and methods for each. The main purpose here is to expose you to a few of these controls. You are encouraged to delve into the toolbox and look at all the tools and find ones you can use in your applications.

Masked Edit Control

```
##|
```

- The **masked edit control** is used to prompt users for data input using a mask pattern. The mask allows you to specify exactly the desired input format. With a mask, the control acts like a standard text box. This control is loaded by selecting the **Microsoft Masked Edit Control** from the Components dialog box.

- Possible uses for this control include:

   ◊ To prompt for a date, a time, number, or currency value.
   ◊ To prompt for something that follows a pattern, like a phone number or social security number.
   ◊ To format the display and printing of mask input data.

- Masked Edit Properties:

Mask	Determines the type of information that is input into the control. It uses characters to define the type of input (see on-line help for complete descriptions).
Text	Contains data entered into the control (including all prompt characters of the input mask).

- Masked Edit Events:

Change	Event called when the data in the control changes.
Validation Error	Event called when the data being entered by the user does not match the input mask.

• Masked Edit Example:

We'll use the masked edit control to obtain a phone number. Place a masked edit control on a form. Set the masked edit controls **Mask** property equal to:

(###)-###-####

Set the **Font Size** property to **12**. My form now looks like this:
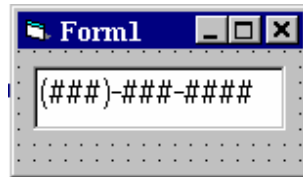
Run the example and notice how simple it is to fill in the phone number. Break the application and examine the Text property of the control in the Immediate Window.

Chart Control

• The **chart control** is an amazing tool. In fact, it's like a complete program in itself. It allows you to design all types of graphs interactively on your form. Then, at run-time, draw graphs, print them, copy them, and change their styles. The control is loaded by selecting **Microsoft Chart Control** from the Components dialog box.

• Possible uses for this control include:

  ◊   To display data in one of many 2D or 3D charts.
  ◊   To load data into a grid from an array.

• Chart Control Properties:

ChartType	Establishes the type of chart to display.
RandomFill	Used to fill chart with random values (good for chcking out chart at design-time). Data is normally loaded from a data grid object associated with the chart control (consult on-line help).

Obviously, there are many more properties used with the chart control. We only look at these two to illustrate what can be done with this powerful control.

*   Chart Control Examples:

Start a new application. Add a chart control to the form. A default bar graph will appear:
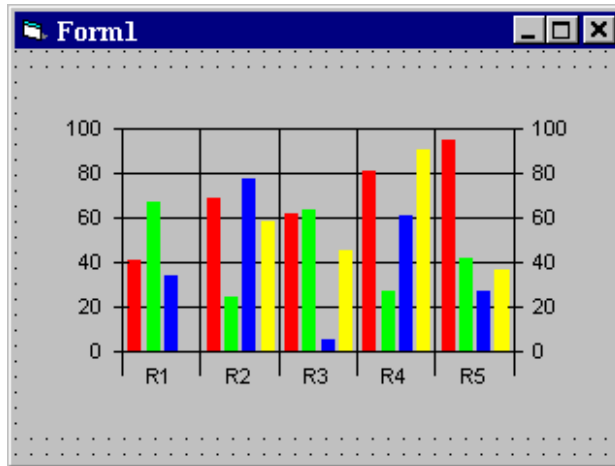
Change the **ChartType** property to a **3** and obtain a line chart:
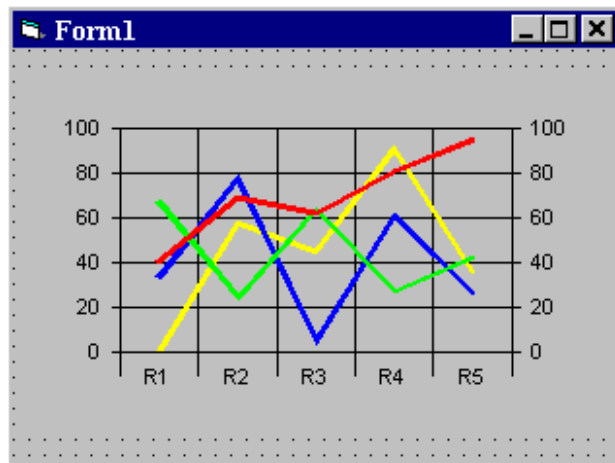
or obtain a fancy 3D chart by using a ChartType of **8**:
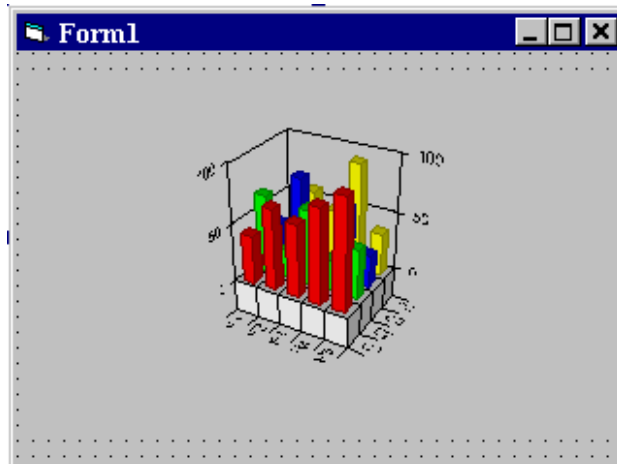
These few quick examples should give you an appreciation for the power and ease of use of the chart control.

Multimedia Control

- The **multimedia control** allows you to manage Media Control Interface (MCI) devices. These devices include: sound boards, MIDI sequencers, CD-ROM drives, audio players, videodisc players, and videotape recorders and players. This control is loaded by selecting the **Microsoft Multimedia Control** from the Components dialog box.

- The primary use for this control is:

    ◊ To manage the recording and playback of MCI devices. This includes the ability to play CD's, record WAV files, and playback WAV files.

- When placed on a form, the multimedia control resembles the buttons you typically see on a VCR:

You should recognize buttons such as Play, Rewind, Pause, etc.

- Programming the Multimedia Control:

  The multimedia control uses a set of high-level, device-independent commands, known as **MCI** (media control interface) commands, to control various multimedia devices. Our example will show you what these commands look like. You are encouraged to further investigate the control (via on-line help) for further functions.

- Multimedia Control Example:

  We'll use the multimedia control to build a simple audio CD player. Put a multimedia control on a form. Place the following code in the **Form_Load** Event:

```
Private Sub Form_Load()
'Set initial properties
Form1.MMControl1.Notify = False
Form1.MMControl1.Wait = True
Form1.MMControl1.Shareable = False
Form1.MMControl1.DeviceType = "CDAudio"
'Open the device
Form1.MMControl1.Command = "Open"
End Sub
```

  This code initializes the device at run time. If an audio CD is loaded into the CD drive, the appropriate buttons on the Multimedia control are enabled:

  This button enabling is an automatic process - no coding is necessary. Try playing a CD with this example and see how the button status changes.

Rich Textbox Control

- The **rich textbox control** allows the user to enter and edit text, providing more advanced formatting features than the conventional textbox control. You can use different fonts for different text sections. You can even control indents, hanging indents, and bulleted paragraphs. This control is loaded by selecting the **Microsoft Rich Textbox Control** from the Components dialog box.

- Possible uses for this control include:

    ◊   Read and view large text files.
    ◊   Implement a full-featured text editor into any applications.

- Rich Textbox Properties, Events, and Methods:

    Most of the properties, events, and methods associated with the conventional textbox are available with the rich text box. A major difference between the two controls is that with the rich textbox, multiple font sizes, styles, and colors are supported. Some unique properties of the rich textbox are:

FileName	Can be used to load the contents of a .txt or .rtf file into the control.
SelFontName	Set the font name for the selected text.
SelFontSize	Set the font size for the selected text.
SelFontColor	Set the font color for the selected text.

    Some unique methods of the rich textbox are:

LoadFile	Open a file and load the contents into the control.
SaveFile	Save the control contents into a file.

- Rich Textbox Example:

    Put a rich textbox control on a form. Put a combo box on the form (we will use this to display the fonts available for use). Use the following code in the **Form_Load** event:

```
Private Sub Form_Load()
Dim I As Integer
For I = 0 To Screen.FontCount - 1
  Combo1.AddItem Screen.Fonts(I)
Next I
End Sub
```

Use the following code in the **Combo1_Click** event:

```
Private Sub Combo1_Click()
RichTextBox1.SelFontName = Combo1.Text
End Sub
```

Run the application. Type some text. Highlight text you want to change the font on. Go to the combo box and select the font. Notice that different areas within the text box can have different fonts:
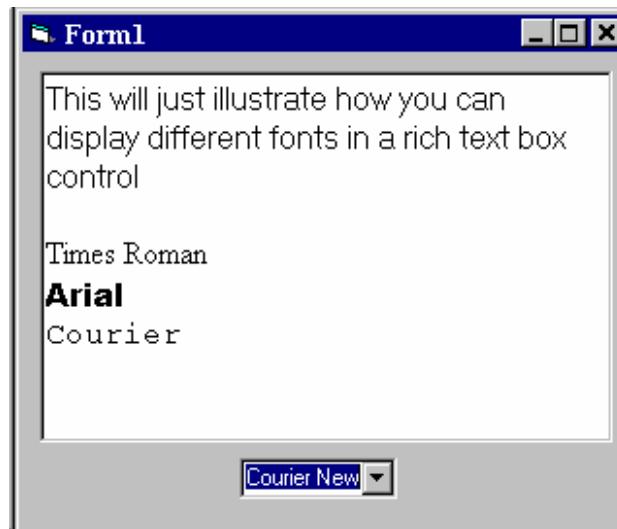
Slider Control

• The **slider control** is similar to a scroll bar yet allows the ability to select a range of values, as well as a single value. This control is part of a group of controls loaded by selecting the **Microsoft Windows Common Controls** from the Components dialog box.

• Possible uses for this control include:

> ◊   To set the value of a point on a graph.
> ◊   To select a range of numbers to be passed into an array.
> ◊   To resize a form, field, or other graphics object.

• Slider Control Properties:

Value	Current slider value.
Min, Max	Establish upper and lower slider limits.
TickFrequency	Determines how many ticks appear on slider.
TickStyle	Determines how and where ticks appear.
SmallChange	Amount slider value changes when user presses left or right arrow keys.
LargeChange	Amount slider value changes when user clicks the slider or presses PgUp or PgDn arrow keys.
SelectRange	Enable selecting a range of values.
SelStart	Starting selected value.
SelLength	Length of select range of values.

• Slider Control Example:

We'll build a slider that lets us select a range of number somewhere between the extreme values of 0 to 100. Put two label boxes and a slider on a form:
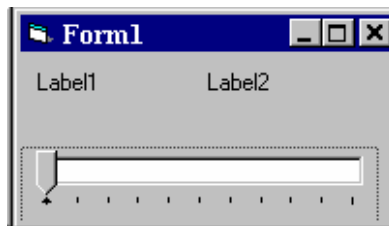
Set the slider control **SmallChange** to **1**, **LargeChange** to **10**, **Min** to **0**, **Max** to **100**, **TickFrequency** to **10**, and **SelectRange** to **True**. Use the following in the **Slider1_MouseDown** event:

```
Private Sub Slider1_MouseDown(Button As Integer, Shift As
Integer, x As Single, y As Single)
If Shift = 1 Then
  Slider1.SelStart = Slider1.Value
  Label1.Caption = Slider1.Value
  Slider1.SelLength = 0
  Label2.Caption = ""
End If
End Sub
```

and this code in the **Slider1_MouseUp** event:

```
Private Sub Slider1_MouseUp(Button As Integer, Shift As
Integer, x As Single, y As Single)
On Error Resume Next
If Shift = 1 Then
   Slider1.SelLength = Slider1.Value - Slider1.SelStart
   Label2.Caption = Slider1.Value
Else
   Slider1.SelLength = 0
End If
End Sub
```

Run the application. Establish a starting value for the selected range by moving the slider to a desired point. Then, click the slider thumb while holding down the Shift key and move it to the desired upper value.

Tabbed Dialog Control

- The **tabbed dialog control** provides an easy way to present several dialogs or screens of information on a single form using the same interface seen in many commercial Windows applications. This control is loaded by selecting the **Sheridan Tabbed Dialog Control** from the Components dialog box.

- The tabbed dialog control provides a group of tabs, each of which acts as a container (works just like a frame or separate form) for other controls. Only one tab can be active at a time. Using this control is easy. Just build each tab container as separate applications: add controls, set properties, and write code like you do for any application. Navigation from one container to the next is simple: just click on the corresponding tab.

- Tabbed Dialog Control Example:

  Start an application and put a tabbed dialog control on the form:
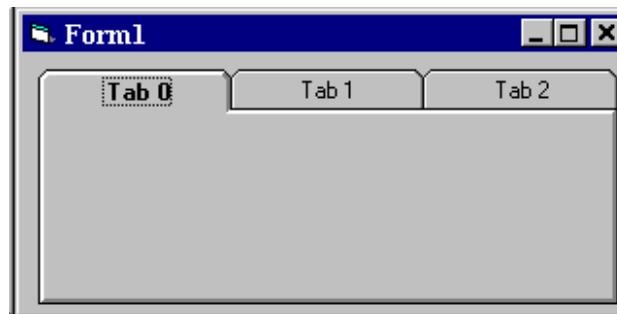
  Design each tab with some controls, then run the application. Note how each tab in the folder has its own working space.

UpDown Control

* The **updown control** is a pair of arrow buttons that the user can click to increment or decrement a value. It works with a **buddy control** which uses the updown control's value property. This control is part of a group of controls loaded by selecting the **Microsoft Windows Common Controls** from the Components dialog box.

* UpDown Control Properties:

Value	Current control value.
Min, Max	Establish upper and lower control limits.
Increment	Amount control value changes each time an arrow is clicked.
Orientation	Determines whether arrows lie horizontally or vertically.

* UpDown Control Events:

Change	Invoked when value property changes.
UpClick	Invoked when up arrow is clicked.
DownClick	Invoked when down arrow is clicked.

* UpDown Control Example:

We'll build an example that lets us establish a number between 1 and 25. Add a updown control and a label box to a form. Set the updown control's Min property to 1 and Max property to 25. The form should resemble:
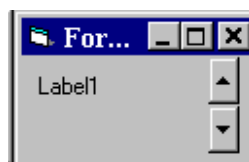
Use this simple code in the **UpDown1_Change** event, then give it a try:

```
Private Sub UpDown1_Change()
Label1.Caption = UpDown1.Value
End Sub
```

Toolbar Control

- Almost all Windows applications these days use toolbars. A toolbar provides quick access to the most frequently used menu commands in an application. The **toolbar control** is a mini-application in itself. It provides everything you need to design and implement a toolbar into your application. This control is part of a group of controls loaded by selecting the **Microsoft Windows Common Controls** from the Components dialog box.

- Possible uses for this control include:

    ◊   Provide a consistent interface between applications with matching toolbars.
    ◊   Place commonly used functions in an easily-accessed space.
    ◊   Provide an intuitive, graphical interface for your application.

- To create a basic toolbar, you need to follow a sequence of steps. You add buttons to a **Button** collection - each button can have optional text and/or an image, supplied by an associated **ImageList** control (another custom control). Buttons can have **tooltips**. In more advanced applications, you can even allow your user to customize the toolbar to their liking!

- After setting up the toolbar, you need to write code for the **ButtonClick** event. The index of the clicked button is passed as an argument to this event. Since toolbar buttons provide quick access to already coded menu options, the code in this event is usually just a call to the respective menu item's **Click** procedure.

- Toolbar Control Example

    We'll look at the simplest use of the toolbar control - building a fixed format toolbar (pictures only) at design time. We'll create a toolbar with five buttons: one to create a **new** file, one to **open** a file, one to **save** a file, one to **print** a file, and one for **help**. Place a toolbar and imagelist control on a form. Right click on the imagelist control to set the pictures to be used. Using the **Images** tab, assign the following five images: Image 1 - NEW.BMP, Image 2 - OPEN.BMP, Image 3 - SAVE.BMP, Image 4 - PRINT.BMP, and Image 5 - HELP.BMP

When done, the image control should look like this:
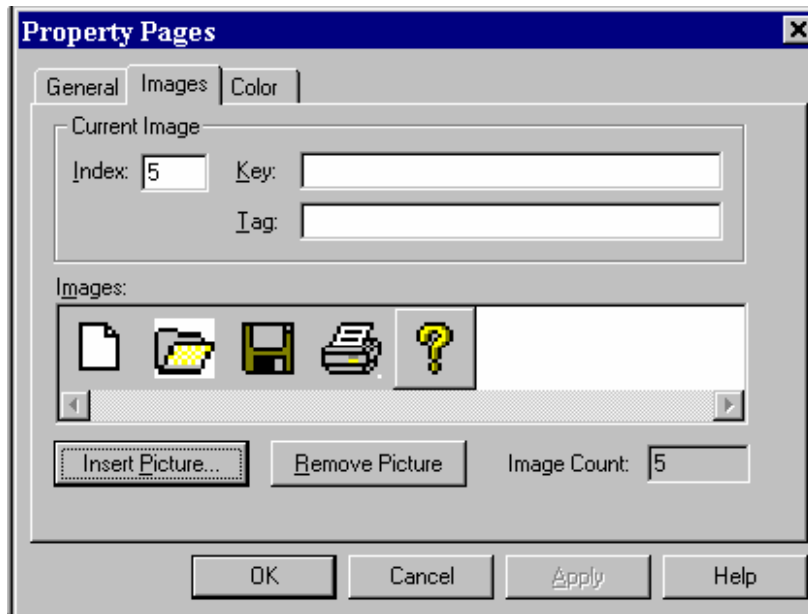
Click **OK** to close this box. Now, right mouse click on the toolbar control.
The **Property Pages** dialog box will appear. Using the General tab, select
the imagelist control just formed. Now, choose the **Buttons** tab to define
each button:
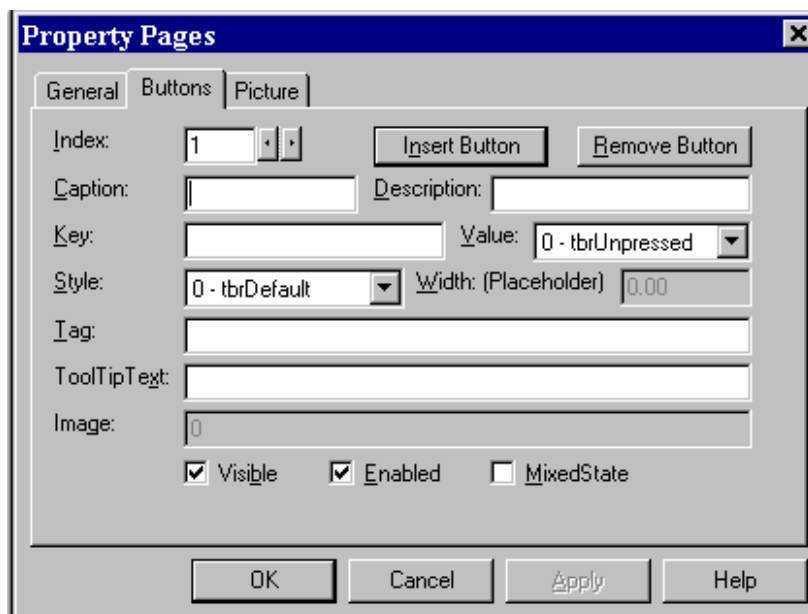
A new button is added to the toolbar by clicking **Insert Button**. At a minimum, for each button, specify the **ToolTipText** property, and the **Image** number. Values I used are:

Index	ToolTipText	Image
1	New File	1
2	Open File	2
3	Save File	3
4	Print File	4
5	-None-	0
6	Help me!	5

Note button 5 is a placeholder (set **Style** property to **tbrPlaceholder**) that puts some space between the first four buttons and the Help button. When done, my form looked like this:
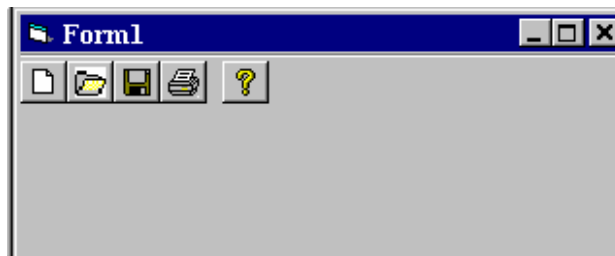
Save and run the application. Note the button's just click - we didn't write any code (as mentioned earlier, the code is usually just a call to an existing menu item's click event). Check out how the tool tips work.

- Quick Note on Tooltips:

Many of the Visual Basic controls support tooltips to inform the user of what a particular control. Simply set individual control's **ToolTipText** property to a non-blank text string to enable this capability.

Using the Windows Clipboard

- The **Clipboard** object has no properties or events, but it has several methods that allow you to transfer data to and from the Windows clipboard. Some methods transfer text, some transfer graphics.

- A method that works with both text and graphics is the **Clear** method:

    Clipboard.Clear   Clear the clipboard contents.

- To move text information to and from the clipboard, use the **SetText** and **GetText** methods:

    Clipboard.SetText   Places text in clipboard.
    Clipboard.GetText   Returns text stored in clipboard.

    These methods are most often used to implement cutting, copying, and pasting operations.

- To move graphics to and from the clipboard, use the **SetData** and **GetData** methods:

    Clipboard.SetData   Places a picture in clipboard.
    Clipboard.GetData   Returns a picture stored in clipboard.

- When using the clipboard methods, you need to know what type of data you are transferring (text or graphics). The **GetFormat** method allows that:

    Clipboard.GetFormat(*datatype*)   Returns **True** if the clipboard contents are of the type specified by *datatype*.

    Possible datatypes are:

Type	Value	Symbolic Constant
DDE conversation info	HBF00	vbCFLink
Rich text format	HBF01	vbCFRTF
Text	1	vbCFText
Bitmap	2	vbCFBitmap
Metafile	3	vbCFMetafile
Device-independent bitmap	8	vbCFDIB
Color palette	9	vbCFPalette

Printing with Visual Basic

- Any serious Visual Basic application will need to use the **printer** to provide the user with a hard copy of any work done or results (text or graphics) obtained. Printing is one of the more complex programming tasks within Visual Basic.

- Visual Basic uses two primary approaches to printing text and graphics:

  ⇒ You can produce the output you want on a form and then print the entire form using the **PrintForm** method.
  ⇒ You can send text and graphics to the **Printer** object and then print them using the **NewPage** and **EndDoc** methods.

  We'll look at how to use each approach, examining advantages and disadvantages of both. All of these techniques use the system default printer. You can also select a printer in Visual Basic, but we won't look at that here.

- The **PrintForm** method sends a pixel-by-pixel image of the specified form to the printer. To print, you first display the form as desired and via code invoke the command: **PrintForm**. This command will print the entire form, using its selected dimensions, even if part of the form is not visible on the screen. If a form contains graphics, they will be printed only if the form's **AutoRedraw** property is **True**.

- The PrintForm method is by far the easiest way to print from an application. But, graphics results may be disappointing because they are reproduced in the resolution of the screen, not the printer. And small forms are still small when printed.

- PrintForm Example:

  Start a new application. Put an image box on the form. Size it and set the **Stretch** property to **True**. Set the **Picture** property to some picture (metafiles are best, you choose). Add a label box. Put some formatted text in the box. My form looks like this:

Add this code to the **Form_Click** event:

```
Private Sub Form_Click()
PrintForm
End Sub
```

Run the application. Click on the form (not the image or label) and things should print. Not too hard, huh?

- Using the **Printer** object to print in Visual Basic is more complicated, but usually provides superior results. But, to get these better results requires a bit (and, at times, more than a bit) of coding.

- The Printer object is a drawing space that supports many methods, like **Print**, **PSet**, **CurrentX**, **CurrentY**, **Line**, **PaintPicture** (used to print contents of Picture boxes), and **Circle**, to create text and graphics. You use these methods just like you would on a form. When you finish placing information on the Printer object, use the **EndDoc** method to send the output to the printer. The **NewPage** method allows printing multi-page documents.

- The Printer object also has several properties that control print quality, page size, number of copies, scaling, page numbers, and more. Consult Visual Basic on-line help for further information.

- The usual approach to using the Printer object is to consider each printed page to be a form with its own coordinate system. Use this coordinate system and the above listed methods to place text and graphics on the page. When complete, use the **EndDoc** method (or **NewPage** method if there are more pages). At that point, the page will print. The main difficulty in using the Printer object is planning where everything goes. I usually use the **Scale** method to define an 8.5" by 11" sheet of standard paper in 0.01" increments:

    Printer.Scale (0, 0) - (850, 1100)

I then place everything on the page relative to these coordinates. The example illustrates the use of a few of these techniques. Consult other Visual Basic documentation for advanced printing techniques.

- Printer Object Example:

  In this example, we'll first define a standard sheet of paper. Then, we'll use the **Line** method to draw a box, the **Circle** method to draw a circle, and the **Print** method to 'draw' some text. Start a new application. We don't need any controls on the form - all the printing is done in the **Form_Click** procedure.

```
Private Sub Form_Click()
Printer.Scale (0, 0)-(850, 1100)
Printer.Line (100, 100)-(400, 300), , B
Printer.Circle (425, 550), 300
Printer.CurrentX = 100
Printer.CurrentY= 800
Printer.Print "This is some text."
Printer.EndDoc
End Sub
```

A few words on each line in this code. First, we establish the printing area to be 850 units wide by 1100 units long. This allows us to place items on a standard page within 0.01 inches. Next, we draw a box, starting 1 inch from the left and 1 inch from the top, that is 3 inches wide and 2 inches high. Then, a circle, centered at mid-page, with radius of 3 inches is drawn. Finally, a line of text is printed near the bottom of the page. The **EndDoc** method does the printing for us. The printed page is shown to the right.
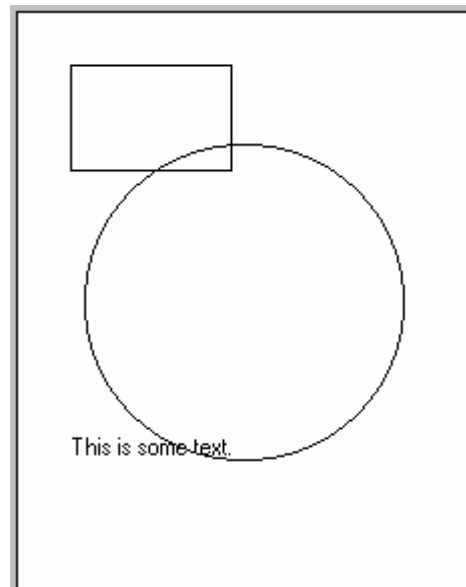
Run the application. Click the form to start the printing. Relate the code to the finished drawing.

- The best way to learn how to print in Visual Basic is to do lots of it. You'll develop your own approaches and techniques as you gain familiarity. Use **FormPrint** for simple jobs. For detailed, custom printing, you'll need to use the **Printer** object.

Multiple Form Visual Basic Applications

- All applications developed in this class use a single form. In reality, most Visual Basic applications use **multiple forms**. The **About** window associated with most applications is a common example of using a second form in an application. We need to learn how to manage multiple forms in our projects.

- To add a form to an application, click the **New** Form button on the toolbar or select **Form** under the **Insert** menu. Each form is designed using exactly the same procedure we always use: draw the controls, assign properties, and write code. Display of the different forms is handled by code you write. You need to decide when and how you want particular forms to be displayed. The user always interacts with the 'active' form.

- The first decision you need to make is to determine which form will be your **startup form**. This is the form that appears when your application first begins. The startup form is designated using the **Project Properties** window, activated using the Visual Basic **Project** menu:

Startup Form

Project1 - Project Properties

General | Make | Compile | Component

Project Type:
Standard EXE

Startup Object:
Form1

Project Name:
Project1

Help File Name:

Project Help Context ID:
0

Project Description:

☑ Upgrade ActiveX Controls
☐ Require License Key

☐ Unattended Execution
○ Thread per Object
● Thread Pool 1 threads

OK   Cancel   Help
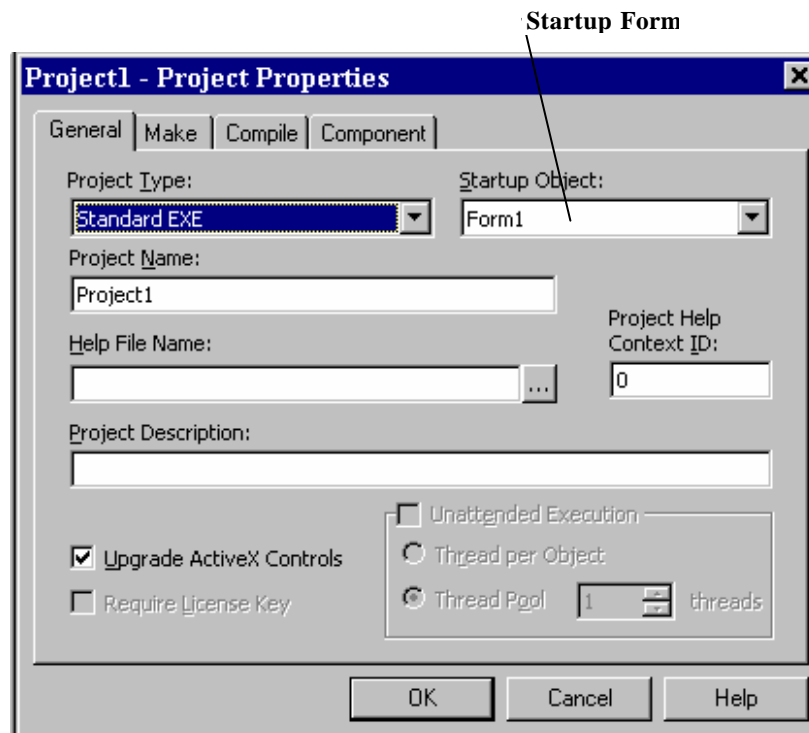
• As mentioned, the startup form automatically loads when your application is run. When you want another form to appear, you write code to load and display it. Similarly, when you want a form to disappear, you write code to unload or hide it. This form management is performed using various **keywords**:

Keyword	Task
Load	Loads a form into memory, but does not display it.
Show vbModeless	Loads (if not already loaded) and displays a modeless form (default **Show** form style).
Show vbModal	Loads (if not already loaded) and displays a modal form.
Hide	Sets the form's **Visible** property to **False**. Form remains in memory.
Unload	Hides a form and removes it from memory.

A **modeless** form can be left to go to other forms. A **modal** form must be closed before going to other forms. The startup form is modeless.

Examples

   Load Form1 ' Loads Form1 into memory, but does not display it
   Form1.Show ' Loads (if needed) and shows Form1 as modeless
   Form1.Show vbModal ' Loads (if needed) and shows Form1 as modal.
   Form1.Hide ' Sets Form1's Visible property to False
   Hide ' Hides the current form
   Unload Form1 ' Unloads Form1 from memory and hides it.

• Hiding a form allows it to be recalled quickly, if needed. Hiding a form retains any data attached to it, including property values, print output, and dynamically created controls. You can still refer to properties of a hidden form. Unload a form if it is not needed any longer, or if memory space is limited.

• If you want to speed up display of forms and memory is not a problem, it is a good idea to **Load** all forms when your application first starts. That way, they are in memory and available for fast recall.

- Multiple Form Example:

Start a new application. Put two command buttons on the form (**Form1**). Set one's **Caption** to **Display Form2** and the other's **Caption** to **Display Form3**. The form will look like this:
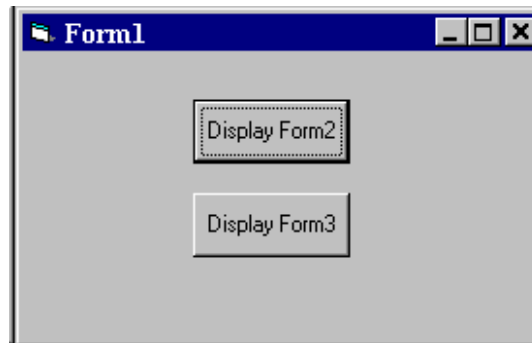
Attach this code to the two command buttons **Click** events.

```
Private Sub Command1_Click()
Form2.Show vbModeless
End Sub

Private Sub Command2_Click()
Form3.Show vbModal
End Sub
```

Add a second form to the application (**Form2**). This form will be **modeless**. Place a command button on the form. Set its **Caption** to **Hide Form**.
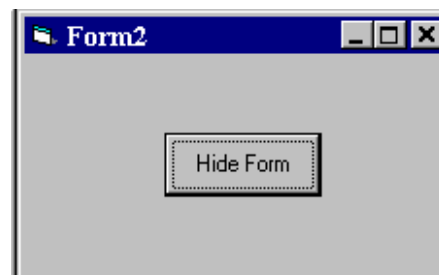
Attach this code to the button's **Click** event.

```
Private Sub Command1_Click()
Form2.Hide
Form1.Show
End Sub
```

Add a third form to the application (**Form3**). This form will be **modal**. Place a command button on the form. Set its **Caption** to **Hide Form**.
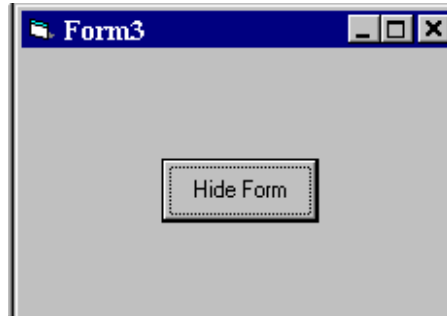
Attach this code to the button's **Click** event.

```
Private Sub Command1_Click()
Form3.Hide
Form1.Show
End Sub
```

Make sure Form1 is the startup form (check the **Project Properties** window under the **Project** menu). Run the application. Note the difference between modal (Form3) and modeless (Form2) forms.

Visual Basic Multiple Document Interface (MDI)

- In the previous section, we looked at using multiple forms in a Visual Basic application. Visual Basic actually provides a system for maintaining multiple-form applications, known as the **Multiple Document Interface** (**MDI**). MDI allows you to maintain multiple forms within a single container form. Examples of MDI applications are Word, Excel, and the Windows Explorer program.

- An MDI application allows the user to display many forms at the same time. The container window is called the **parent** form, while the individual forms within the parent are the **child** forms. Both parent and child forms are modeless, meaning you can leave one window to move to another. An application can have only one parent form. Creating an MDI application is a two-step process. You first create the MDI form (choose **Add MDI Form** from **Project** menu) and define its menu structure. Next, you design each of the application's child forms (set **MDIChild** property to **True**).

- Design-Time Features of MDI Child Forms:

  At **design time**, child forms are not restricted to the area inside the parent form. You can add controls, set properties, write code, and design the features of child forms anywhere on the desktop.

  You can determine whether a form is a child by examining its **MDIChild** property, or by examining the project window. The project window uses special icons to distinguish **standard** forms, MDI **child** forms, and MDI **parent** forms:
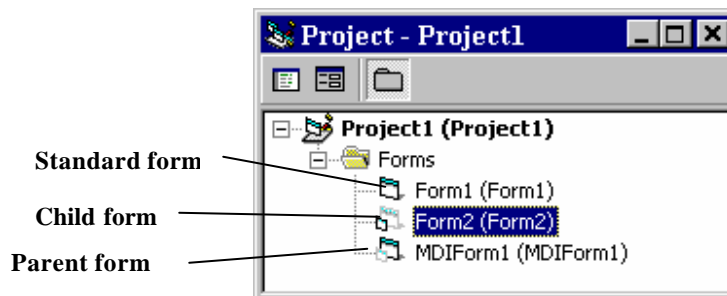
- Run-Time Features of MDI Child Forms:

  At **run-time**, the parent and child forms take on special characteristics and abilities. Some of these are:

  1. At run-time all child forms are displayed within the parent form's internal area. The user can move and size child forms like any other form, but they must stay in this internal area.

2. When a child is minimized, its icon appears on the MDI parent form instead of the user's desktop. When the parent form is minimized, the entire application is represented by a single icon. When restored, all forms are redisplayed as they were.

3. When a child form is maximized, its caption is combined with the parent form's caption and displayed in the parent title bar.

4. By setting the **AutoShowChildren** property, you can display child forms automatically when forms are loaded (**True**), or load child forms as hidden (**False**).

5. The active child form's menus (if any) are displayed on the parent form's menu bar, not the child form.

6. New child forms can be created at run-time using a special form of the **Dim** statement and the **Show** statement (the example illustrates this process).

7. The parent form's **ActiveForm** property indicates which child form is currently active. The **ActiveControl** property indicates which control on the active child form has focus.

8. The **Arrange** command can be used to determine how the child forms and their icons (if closed) are displayed. The syntax is:

    Arrange *style*

    where *style* can take on these values:

Style	Symbolic Constant	Effect
0	vbCascade	Cascade all nonminimized MDI child forms.
1	vbTileHorizontal	Horizontally tile all nonminimized MDI child forms.
2	vbTileVertical	Vertically tile all nonminimized MDI child forms.
3	vbArrangeIcons	Arrange icons for minimized MDI child forms.

- Multiple-Document Application (MDI) Example:

  We'll create an MDI application which uses a simple, text box-based, editor as the child application. There are a lot of steps, even for a simple example. Start a new application. Create a parent form by selecting **MDI Form** from the **Insert** menu. At this point, the project will contain an MDI parent form (**MDIForm1**) and a standard form (**Form1**) which we will use as a child form. Make MDIForm1 the startup form. We work with the parent form first:

  1. Set the following properties:

Caption	MDI Example
Name	frmParent
WindowState	2-Maximized

  2. Set up the following menu structure:

Caption	Name	Indented	
&File	mnuFile	No	
&New	mnuFileNew	Yes	
&Arrange	mnuArrange	No	
&Cascade	mnuArrangeItem	Yes	Index = 0
&Horizontal Tile	mnuArrangeItem	Yes	Index = 1
&Vertical Tile	mnuArrangeItem	Yes	Index = 2
&Arrange Icons	mnuArrangeItem	Yes	Index = 3

  3. Attach this code to the **mnuFileNew_Click** procedure. This code creates new child forms (named **frmChild** - developed next).

```
Private Sub mnuFileNew_Click()
Dim NewDoc As New frmChild
NewDoc.Show
End Sub
```

  4. Attach this code to the **mnuArrangeItem_Click** procedure. This establishes how child forms are displayed.

```
Private Sub mnuArrangeItem_Click(Index As Integer)
Arrange Index
End Sub
```

Now, we'll work with Form1 which will hold the child application:

5. Draw a text box on the form. Set the following properties for the form and the text box:

> **Form1**:
> | | |
> |---|---|
> | Caption | Child Form |
> | MDIChild | True |
> | Name | frmChild |
> | Visible | False |
>
> **Text1**:
> | | |
> |---|---|
> | Left | 0 |
> | MultiLine | True |
> | ScrollBars | 2-Vertical |
> | Text | [Blank] |
> | Top | 0 |

My form resembles this:
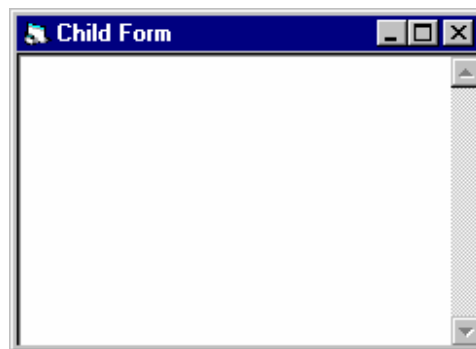
6. Attach this code to the **Form_Resize** procedure. This insures that whenever a child window is resized, the text box fills up the entire window.

```
Private Sub Form_Resize()
Text1.Height = ScaleHeight
Text1.Width = ScaleWidth
End Sub
```

Run the application. Create new forms by selecting **New** from the **File** menu. Try resizing forms, maximizing forms (notice how the parent form title bar changes), minimizing forms, closing forms. Try all the **Arrange** menu options.

Creating a Help File

- During this course, we've made extensive use of the Visual Basic on-line help system. In fact, one of the major advances in software in the past few years has been improvements in such interactive help. Adding a **help file** to your Visual Basic application will give it real polish, as well as making it easier to use.

- Your help file will contain text and graphics information needed to be able to run your application. The help file will be displayed by the built-in Windows help utility that you use with every Windows application, hence all functions available with that utility are available with your help system. For example, each file can contain one or more topics that your user can select by clicking a **hot spot**, using a **keyword search**, or **browsing** through text. And, it's easy for your user to print any or all help topics.

- Creating a complete help file is a major task and sometimes takes as much time as creating the application itself! Because of this, we will only skim over the steps involved, generate a simple example, and provide guidance for further reference.

- There are five major steps involved in building your own help file:

    1. Create your application and develop an outline of help system topics.
    2. Create the **Help Text File** (or Topic File) in RTF format.
    3. Create the **Help Project File** (HPJ).
    4. Compile the Help File using the **Help Compiler** and Project File.
    5. **Attach** the Help File to your Visual Basic application.

  Step 1 is application-dependent. We'll look briefly at the last four steps here. More complete details, including formatting and file structure requirements, are available in many Visual Basic references..

- Creating a Help Text File:

  To create a **Help Text File**, you need to use a word processor capable of saving documents in rich-text format (**RTF**). **Word** and **WordPerfect** do admirable jobs. You must also be familiar with text formatting procedures such as underlining, double-underlining, typing hidden text, and using footnotes. This formatting is used to delineate different parts of the help file. You should make sure all formatting options are visible when creating the Help Text File.

The Help Text File is basically a cryptically encoded list of **hypertext** jumps (jump phrases) and context strings. These are items that allow navigation through the topics in your help file. Some general rules of Help Text Files:

* Topics are separated by hard page breaks.
* Each topic must have a unique context string.
* Each topic can have a title.
* A topic can have many keywords attached to it to enable quick access utilizing a search facility.
* Topics can have build-tag indicators and can be assigned a browse sequence.
* Jumps can be to another secondary window or to another file.

Once completed, your Help Text File must be saved as an RTF file.

• Help Text File Example:

We'll create a very simple help text file with three topics. I used Word 6.0 in this example. Create a document with the following structure and footnotes:
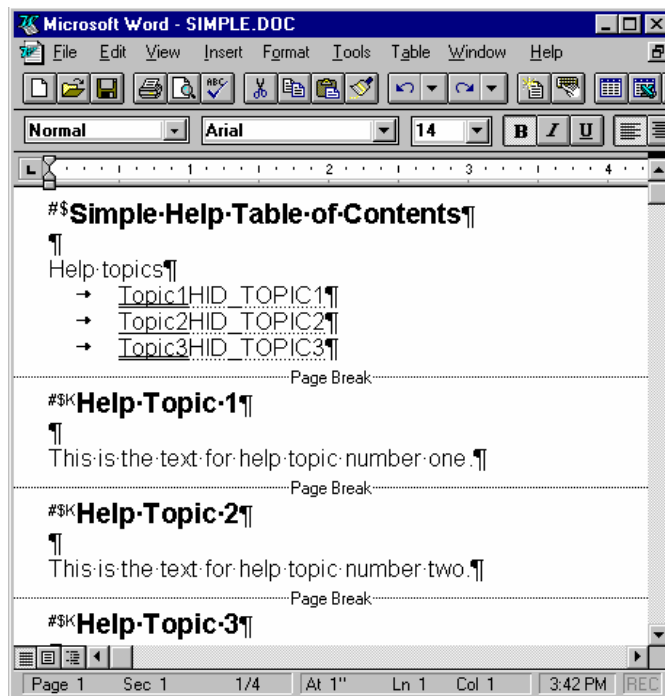
Some things to note: **Topic1** and **Topic3** (hypertext jumps) are double-underlined to indicate clickable jumps to topics. **Topic2** is single-underlined to indicate a jump to a pop-up topic. The words **HID_TOPIC1**, **HID_TOPIC2**, and **HID_TOPIC3** (context strings) are formatted as hidden text. Note page breaks separate each section. Do **not** put a page break at the end of the file.

Also, note the use of footnotes. The **#** footnote is used to specify a Help context ID, the **$** provides Topic Titles for searching, and **K** yields search keywords. The footnotes for this example are:
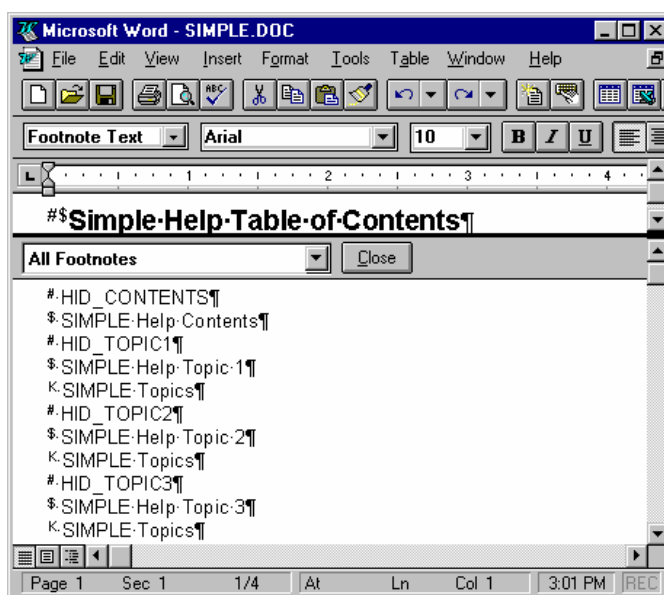
When done, save this file as **SIMPLE.RTF** (Rich Text Format).

- Creating the Help Project File:

The **Help Project File** contains the information required by the Help Compiler to create the Help file. The file is created using any text editor and must be saved as unformatted text (**ASCII**). The file extension is **HPJ**.

The Help Project File can contain up to nine sections, each of which supplies information about the source file to compile. Sections names are placed within square brackets **[ ]**. Semicolons are used to indicate a comment. Sections can be in any order. The sections are:

[OPTIONS]	Specifies options for build (optional).
[FILES]	Specifies Help Text Files (RTF) (required).
[BUILDTAGS]	Specifies any build tags (optional).
[CONFIG]	Author defined menus, macros, etc. (optional)

[BITMAPS]	Specifies any bitmaps needed for build.
[ALIAS]	Can be used to specify context strings to topics (optional).
[MAP]	Associates context strings with numbers. Used with context-sensitive help (optional).
[WINDOWS]	Defines primary and secondary windows (required only if secondary windows used).
[BAGGAGE]	Lists files to be included in **HLP** file.

• Help Project File Example:

For our simple example, the Help Project File is equally simple:

[OPTIONS]
CONTENTS=HID_CONTENTS
TITLE=SIMPLE Application Help
[FILES]
SIMPLE.RTF

This file specifies the context ID of the Table of Contents screen and the name of the RTF file that contains the help text. Save this file as **SIMPLE.HPJ** (in Text, or ASCII format).

• Compiling the Help File:

This is the easiest step. The **help compiler** is located in the **c:\Program Files\DevStudio\vb\hc** directory and is the program **hc.exe**. Your file is compiled within the DOS window. Once in that window, move to the directory containing your HPJ file and type:

c:\Program Files\DevStudio \vb\hc\hc *filename*.HPJ

where *filename* is your Help Project File. This process generates a binary help resource file and may take a long time to complete. Any errors are probably due to problems in the RTF file(s). The created file has the same name as your Help Project File with an **HLP** extension.

• Help File Example:

To compile the exa mple, at a DOS prompt, type:

c:\Program Files\DevStudio\vb\hc\hc SIMPLE.HPJ

The help file SIMPLE.HLP will be created (if no errors occur) and saved in the same directory as your HPJ file.

• Attaching the Help File:

The final step is to **attach** the compiled **help file** to your application. As a first step, open the **Project Properties** window under the **Project** menu. Under **Help File**, select the name of your **HLP** file by clicking the ellipsis (**...**). This ties the help file to the application, enabling the user to press **F1** for help.

You can also add a Help item somewhere in your menu structure that invokes help via its **Click** event. If you do this, you must write code to invoke the help file. The code involves a call to the Windows API function, **WinHelp**. But, after last class, we're not daunted by such functions, are we? First, we need the function declaration (from the API Text Viewer):

    Declare Function WinHelp Lib "user32" Alias "WinHelpA" (ByVal hwnd
    As Long, ByVal lpHelpFile As String, ByVal wCommand As Long,
    ByVal dwData As Long) As Long

We also need a constant (also from the API Text Viewer):

    Const HELP_INDEX = &H3   ' Display index

This constant will declare the Help files index page upon invocation of WinHelp. There are other constants that can be used with WinHelp - this is just a simple example. The **Declare** statement and constant definitions usually go in the general declarations area of a code module and made **Public**. If you only have one form in your application, then put these statements in the general declarations area of your form (and declare them **Private**). Once everything is in-place, to invoke the Help file from code, use the function call:

    Dim R As Long
        .
        .
    R = WinHelp(*startupform*.hWnd, *filename*.HLP, HELP_INDEX, CLng(0))

where *startupform* is the name of your application main form and *filename* is the help file name, including path information.

- Help File Example:

  We can now try our example help file in a Visual Basic application. We'll only use the **F1** option here. Start a new application. Bring up the **Project Properties** window via the **Project** menu. Select the correct Help File by clicking the ellipsis and finding your newly created file. Click **OK**. Now, run your application (I know there's nothing in the application, but that's all right). Once, it's running press **F1**. This Help screen should appear:
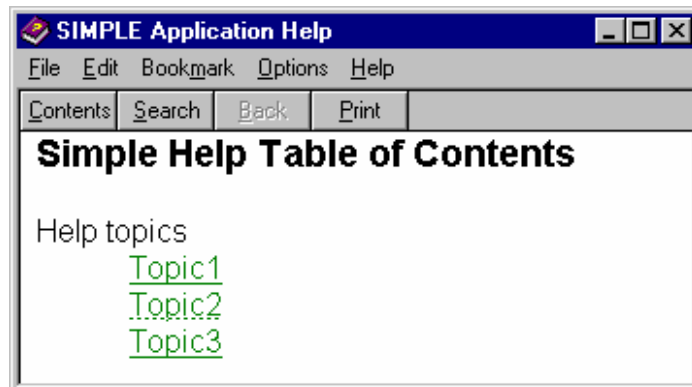
  Move the mouse cursor to **Topic1** and notice the cursor changes to a hand. Click there and the corresponding Topic 1 screen appears:
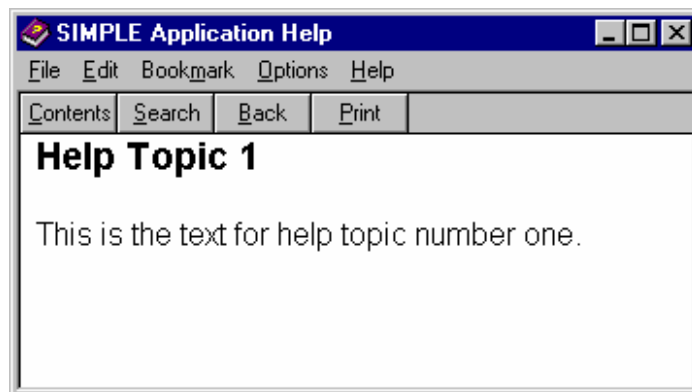
  The **HID_TOPIC1** text in the Table of Contents screen links to the corresponding context ID (the # footnote) in the topic page. This link is a **jump**. The link to Topic 2 is a pop-up jump, try it and you'll see.

Go back to the Table of Contents screen and click the **Search** button. A dialo g box displaying the help file's list of keywords appears. In our example, the three topics all have the same keyword (the **K** footnotes), **SIMPLE Topics**. When you double-click on this keyword, you see all the associated topic titles (the **$** footnotes):
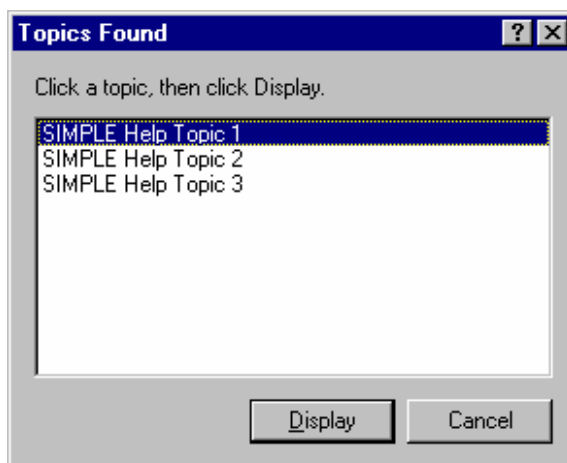
You can now select your topic of choice.

- More Help File Topics:

  After all this work, you will still only have a simple help file, nothing that rivals those seen in most applications. To improve your help system, you need to add some more **stuff**. Information on these advanced help topics is found in many Visual Basic references.

  A big feature of help systems is **context-sensitive help**. With this, you place the cursor on or in something your interested in knowing about and press **F1**. A Help topic, if one exists, shows up. The application is smart enough to know what you want help with. **Graphics** always spiff up a help system. Help systems use a special type of graphics called **hypergraphics**. Lastly, **Help macros** add functionality to your help system. The re are over 50 macro routines built into the DLL WinHelp application.

- If, after seeing the rather daunting tasks involved in creating a help system, you don't want to tackle the job, take heart. There are several third party software packages that assist in help system authoring and development. Look at computer magazine advertisements (especially the *Visual Basic Programmer's Journal*) for potential leads.

Class Summary

- That's all I know about Visual Basic. You should now have a good breadth of knowledge concerning the Visual Basic environment and language. This breadth should serve as a springboard into learning more as you develop your own applications. Feel free to contact me, if you think I can answer any questions you might have.

- Where do you go from here? With Visual Basic 6.0, you can extend your knowledge to write Web-based applications, develop massive database front-ends using Visual Basic's powerful database tools and techniques, and even develop your own ActiveX (custom) controls. Other classes cover such topics.

- And, the last example:
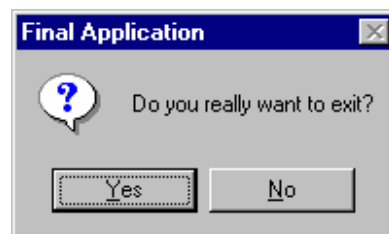
Exercise 10

The Ultimate Application

Design an application in Visual Basic that everyone on the planet wants to buy. Draw objects, assign properties, attach code. Thoroughly debug and test your application. Create a distribution disk. Find a distributor or distribute it yourself through your newly created company. Become fabulously wealthy. Remember those who made it all possible by rewarding them with jobs and stock options.

My Solution:

Still working on it ...

This page intentionally not left blank.

Learn Visual Basic 6.0

Appendix I. Visual Basic Symbolic Constants

Contents

Alignment Constants

Align Property

Constant	Value	Description
vbAlignNone	0	Size and location set at design time or in code.
vbAlignTop	1	Picture box at top of form.
vbAlignBottom	2	Picture box at bottom of form.
vbAlignLeft	3	Picture box at left of form.
vbAlignRight	4	Picture box at right of form.

Alignment Property

Constant	Value	Description
vbLeftJustify	0	Left align.
vbRightJustify	1	Right align.
vbCenter	2	Center.

Border Property Constants

BorderStyle Property (Form)

Constant	Value	Description
vbBSNone	0	No border.
vbFixedSingle	1	Fixed single.
vbSizable	2	Sizable (forms only)
vbFixedDouble	3	Fixed double (forms only)

BorderStyle Property (Shape and Line)

Constant	Value	Description
vbTransparent	0	Transparent.
vbBSSolid	1	Solid.
vbBSDash	2	Dash.
vbBSDot	3	Dot.
vbBSDashDot	4	Dash-dot.
vbBSDashDotDot	5	Dash-dot-dot.
vbBSInsideSolid	6	Inside solid.

Clipboard Object Constants

Constant	Value	Description
vbCFLink	0xBF00	DDE conversation information.
vbCFRTF	0xBF01	Rich Text Format (.RTF file)
vbCFText	1	Text (.TXT file)
vbCFBitmap	2	Bitmap (.BMP file)
vbCFMetafile	3	Metafile (.WMF file)
vbCFDIB	8	Device-independent bitmap.
vbCFPalette	9	Color palette.

Color Constants

Colors

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

System Colors

Constant	Value	Description
vbScrollBars	0x80000000	Scroll bar color.
vbDesktop	0x80000001	Desktop color.
vbActiveTitleBar	0x80000002	Color of the title bar for the active window.
vbInactiveTitleBar	0x80000003	Color of the title bar for the inactive window.
vbMenuBar	0x80000004	Menu background color.
vbWindowBackground	0x80000005	Window background color.
vbWindowFrame	0x80000006	Window frame color.
vbMenuText	0x80000007	Color of text on menus.
vbWindowText	0x80000008	Color of text in windows.
vbTitleBarText	0x80000009	Color of text in caption, size box, and scroll arrow.
vbActiveBorder	0x8000000A	Border color of active window.
vbInactiveBorder	0x8000000B	Border color of inactive window.
vbApplicationWorkspace	0x8000000C	Background color of multiple-document interface (MDI)

System Colors (continued)

Constant	Value	Description
vbHighlight	0x8000000D	Background color of items selected in a control.
vbHighlightText	0x8000000E	Text color of items selected in a control.
vbButtonFace	0x8000000F	Color of shading on the face of command buttons.
vbButtonShadow	0x80000010	Color of shading on the edge of command buttons.
vbGrayText	0x80000011	Grayed (disabled)
vbButtonText	0x80000012	Text color on push buttons.
vbInactiveCaptionText	0x80000013	Color of text in an inactive caption.
vb3DHighlight	0x80000014	Highlight color for 3D display elements.
vb3DDKShadow	0x80000015	Darkest shadow color for 3D display elements.
vb3DLight	0x80000016	Second lightest of the 3D colors after vb3DHighlight.
vbInfoText	0x80000017	Color of text in ToolTips.
vbInfoBackground	0x80000018	Background color of ToolTips.

Control Constants

ComboBox Control

Constant	Value	Description
vbComboDropdown	0	Dropdown Combo.
vbComboSimple	1	Simple Combo.
vbComboDropdownList	2	Dropdown List.

ListBox Control

Constant	Value	Description
vbMultiSelectNone	0	None.
vbMultiSelectSimple	1	Simple.
vbMultiSelectExtended	2	Extended.

ScrollBar Control

Constant	Value	Description
vbSBNone	0	None.
vbHorizontal	1	Horizontal.
vbVertical	2	Vertical.
vbBoth	3	Both.

Shape Control

Constant	Value	Description
vbShapeRectangle	0	Rectangle.
vbShapeSquare	1	Square.
vbShapeOval	2	Oval.
vbShapeCircle	3	Circle.
vbShapeRoundedRectangle	4	Rounded rectangle.
vbShapeRoundedSquare	5	Rounded square.

Data Control Constants

Error Event Constants

Constant	Value	Description
vbDataErrContinue	0	Continue.
vbDataErrDisplay	1	(Default)

EditMode Property Constants

Constant	Value	Description
vbDataEditNone	0	No editing operation in progress.
vbDataEditMode	1	Edit method invoked; current record in copy buffer.
vbDataEditAdd	2	AddNew method invoked; current record hasn't been saved.

Options Property Constants

Constant	Value	Description
vbDataDenyWrite	1	Other users can't change records in recordset.
vbDataDenyRead	2	Other users can't read records in recordset.
vbDataReadOnly	4	No user can change records in recordset.
vbDataAppendOnly	8	New records can be added to the recordset, but existing records can't be read.
vbDataInconsistent	16	Updates can apply to all fields of the recordset.
vbDataConsistent	32	Updates apply only to those fields that will not affect other records in the recordset.
vbDataSQLPassThrough	64	Sends an SQL statement to an ODBC database.

Validate Event Action Constants

Constant	Value	Description
vbDataActionCancel	0	Cancel the operation when the Sub exits.
vbDataActionMoveFirst	1	MoveFirst method.
vbDataActionMovePrevious	2	MovePrevious method.
vbDataActionMoveNext	3	MoveNext method.
vbDataActionMoveLast	4	MoveLast method.
vbDataActionAddNew	5	AddNew method.
vbDataActionUpdate	6	Update operation (not UpdateRecord)
vbDataActionDelete	7	Delete method.
vbDataActionFind	8	Find method.
vbDataActionBookmark	9	The Bookmark property is set.
vbDataActionClose	10	Close method.
vbDataActionUnload	11	The form is being unloaded.

Beginning-of-File Constants

Constant	Value	Description
vbMoveFirst	0	Move to first record.
vbBOF	1	Move to beginning of file.

End-of-File Constants

Constant	Value	Description
vbMoveLast	0	Move to last record.
vbEOF	1	Move to end of file.
vbAddNew	2	Add new record to end of file.

Recordset-Type Constants

Constant	Value	Description
vbRSTypeTable	0	Table-type recordset.
vbRSTypeDynaset	1	Dynaset-type recordset.
vbRSTypeSnapShot	2	Snapshot-type recordset.

Date Constants

firstdayofweek **Argument Values**

Constant	Value	Description
vbUseSystem	0	Use NLS API setting.
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

firstweekofyear **Argument Values**

Constant	Value	Description
vbUseSystem	0	Use application setting if one exists; otherwise use NLS API setting.
vbFirstJan1	1	Start with week in which January 1 occurs (default)
vbFirstFourDays	2	Start with the first week that has at least four days in the new year.
vbFirstFullWeek	3	Start with the first full week of the year.

Return Values

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

DBGrid Control Constants

Alignment Constants

Constant	Value	Description
dbgLeft	0	Left.
dbgRight	1	Right.
dbgCenter	2	Center.
dbgGeneral	3	General.

BorderStyle Constants

Constant	Value	Description
dbgNone	0	None.
dbgFixedSingle	1	FixedSingle.

DataMode Constants

Constant	Value	Description
dbgBound	0	Bound.
dbgUnbound	1	Unbound.

DividerStyle Constants

Constant	Value	Description
dbgNoDividers	0	NoDividers.
dbgBlackLine	1	BlackLine.
dbgDarkGrayLine	2	DarkGrayLine.
dbgRaised	3	Raised.
dbgInset	4	Inset.
dbgUseForeColor	5	UseForeColor.

RowDividerStyle Constants

Constant	Value	Description
dbgNoDividers	0	NoDividers.
dbgBlackLine	1	BlackLine.
dbgDarkGrayLine	2	DarkGrayLine.
dbgRaised	3	Raised.
dbgInset	4	Inset.
dbgUseForeColor	5	UseForeColor.

Scroll Bar Constants

Constant	Value	Description
dbgNone	0	None.
dbgHorizontal	1	Horizontal.
dbgVertical	2	Vertical.
dbgBoth	3	Both.
dbgAutomatic	4	Automatic.

DDE Constants

linkerr (LinkError Event)

Constant	Value	Description
vbWrongFormat	1	Another application requested data in wrong format.
vbDDESourceClosed	6	Destination application attempted to continue after source closed.
vbTooManyLinks	7	All source links are in use.
vbDataTransferFailed	8	Failure to update data in destination.

LinkMode Property (Forms and Controls)

Constant	Value	Description
vbLinkNone	0	None.
vbLinkSource	1	Source (forms only)
vbLinkAutomatic	1	Automatic (controls only)
vbLinkManual	2	Manual (controls only)
vbLinkNotify	3	Notify (controls only)

Dir, GetAttr, and SetAttr Constants

Constant	Value	Description
vbNormal	0	Normal (default for Dir and SetAttr)
vbReadOnly	1	Read-only.
vbHidden	2	Hidden.
vbSystem	4	System file.
vbVolume	8	Volume label.
vbDirectory	16	Directory.
vbArchive	32	File has changed since last backup.

Drag-and-Drop Constants

DragOver Event

Constant	Value	Description
vbEnter	0	Source control dragged into target.
vbLeave	1	Source control dragged out of target.
vbOver	2	Source control dragged from one position in target to another.

Drag Method (Controls)

Constant	Value	Description
vbCancel	0	Cancel drag operation.
vbBeginDrag	1	Begin dragging control.
vbEndDrag	2	Drop control.

DragMode Property

Constant	Value	Description
vbManual	0	Manual.
vbAutomatic	1	Automatic.

Drawing Constants

DrawMode Property

Constant	Value	Description
vbBlackness	1	Black.
vbNotMergePen	2	Not Merge pen.
vbMaskNotPen	3	Mask Not pen.
vbNotCopyPen	4	Not Copy pen.
vbMaskPenNot	5	Mask pen Not.
vbInvert	6	Invert.
vbXorPen	7	Xor pen.
vbNotMaskPen	8	Not Mask pen.
vbMaskPen	9	Mask pen.
vbNotXorPen	10	Not Xor pen.
vbNop	11	No operation; output remains unchanged.
vbMergeNotPen	12	Merge Not pen.
vbCopyPen	13	Copy pen.
vbMergePenNot	14	Merge pen Not.
vbMergePen	15	Merge pen.
vbWhiteness	16	White.

DrawStyle Property

Constant	Value	Description
vbSolid	0	Solid.
vbDash	1	Dash.
vbDot	2	Dot.
vbDashDot	3	Dash-dot.
vbDashDotDot	4	Dash-dot-dot.
vbInvisible	5	Invisible.
vbInsideSolid	6	Inside solid.

Form Constants

Show Parameters

Constant	Value	Description
vbModal	1	Modal form.
vbModeless	0	Modeless form.

Arrange Method for MDI Forms

Constant	Value	Description
vbCascade	0	Cascade all nonminimized MDI child forms.
vbTileHorizontal	1	Horizontally tile all nonminimized MDI child forms.
vbTileVertical	2	Vertically tile all nonminimized MDI child forms.
vbArrangeIcons	3	Arrange icons for minimized MDI child forms.

WindowState Property

Constant	Value	Description
vbNormal	0	Normal.
vbMinimized	1	Minimized.
vbMaximized	2	Maximized.

Graphics Constants

FillStyle Property

Constant	Value	Description
vbFSSolid	0	Solid.
vbFSTransparent	1	Transparent.
vbHorizontalLine	2	Horizontal line.
vbVerticalLine	3	Vertical line.
vbUpwardDiagonal	4	Upward diagonal.
vbDownwardDiagonal	5	Downward diagonal.
vbCross	6	Cross.
vbDiagonalCross	7	Diagonal cross.

ScaleMode Property

Constant	Value	Description
vbUser	0	User.
vbTwips	1	Twips.
vbPoints	2	Points.
vbPixels	3	Pixels.
vbCharacters	4	Characters.
vbInches	5	Inches.
vbMillimeters	6	Millimeters.
vbCentimeters	7	Centimeters.

Grid Control Constants

ColAlignment, FixedAlignment Properties

Constant	Value	Description
grdAlignCenter	2	Center data in column.
grdAlignLeft	0	Left-align data in column.
grdAlignRight	1	Right-align data in column.

FillStyle Property

Constant	Value	Description
grdSingle	0	Changing Text property setting affects only active cell.
grdRepeat	1	Changing Text property setting affects all selected cells.

Help Constants

Constant	Value	Description
cdlHelpContext	0x1	Displays Help for a particular topic.
cdlHelpQuit	0x2	Notifies the Help application that the specified Help file is no longer in use.
cdlHelpIndex	0x3	Displays the index of the specified Help file.
cdlHelpContents	0x3	Displays the contents topic in the current Help file.
cdlHelpHelpOnHelp	0x4	Displays Help for using the Help application itself.
cdlHelpSetIndex	0x5	Sets the current index for multi-index Help.
cdlHelpSetContents	0x5	Designates a specific topic as the contents topic.
cdlHelpContextPopup	0x8	Displays a topic identified by a context number.
cdlHelpForceFile	0x9	Creates a Help file that displays text in only one font.
cdlHelpKey	0x101	Displays Help for a particular keyword.
cdlHelpCommandHelp	0x102	Displays Help for a particular command.
cdlHelpPartialKey	0x105	Calls the search engine in Windows Help.

Key Code Constants

Key Codes

Constant	Value	Description
vbKeyLButton	0x1	Left mouse butto n.
vbKeyRButton	0x2	Right mouse button.
vbKeyCancel	0x3	CANCEL key.
vbKeyMButton	0x4	Middle mouse button.
vbKeyBack	0x8	BACKSPACE key.
vbKeyTab	0x9	TAB key.
vbKeyClear	0xC	CLEAR key.
vbKeyReturn	0xD	ENTER key.
vbKeyShift	0x10	SHIFT key.
vbKeyControl	0x11	CTRL key.
vbKeyMenu	0x12	MENU key.

Key Codes (continued)

Constant	Value	Description
vbKeyPause	0x13	PAUSE key.
vbKeyCapital	0x14	CAPS LOCK key.
vbKeyEscape	0x1B	ESC key.
vbKeySpace	0x20	SPACEBAR key.
vbKeyPageUp	0x21	PAGE UP key.
vbKeyPageDown	0x22	PAGE DOWN key.
vbKeyEnd	0x23	END key.
vbKeyHome	0x24	HOME key.
vbKeyLeft	0x25	LEFT ARROW key.
vbKeyUp	0x26	UP ARROW key.
vbKeyRight	0x27	RIGHT ARROW key.
vbKeyDown	0x28	DOWN ARROW key.
vbKeySelect	0x29	SELECT key.
vbKeyPrint	0x2A	PRINT SCREEN key.
vbKeyExecute	0x2B	EXECUTE key.
vbKeySnapshot	0x2C	SNAPSHOT key.
vbKeyInsert	0x2D	INS key.
vbKeyDelete	0x2E	DEL key.
vbKeyHelp	0x2F	HELP key.
vbKeyNumlock	0x90	NUM LOCK key.

KeyA Through KeyZ Are the Same as Their ASCII Equivalents: 'A' Through 'Z'

Constant	Value	Description
vbKeyA	65	A key.
vbKeyB	66	B key.
vbKeyC	67	C key.
vbKeyD	68	D key.
vbKeyE	69	E key.
vbKeyF	70	F key.
vbKeyG	71	G key.
vbKeyH	72	H key.
vbKeyI	73	I key.
vbKeyJ	74	J key.
vbKeyK	75	K key.
vbKeyL	76	L key.
vbKeyM	77	M key.
vbKeyN	78	N key.
vbKeyO	79	O key.
vbKeyP	80	P key.
vbKeyQ	81	Q key.
vbKeyR	82	R key.
vbKeyS	83	S key.
vbKeyT	84	T key.

KeyA Through KeyZ (continued)

Constant	Value	Description
vbKeyU	85	U key.
vbKeyV	86	V key.
vbKeyW	87	W key.
vbKeyX	88	X key.
vbKeyY	89	Y key.
vbKeyZ	90	Z key.

Key0 Through Key9 Are the Same as Their ASCII Equivalents: '0' Through '9'

Constant	Value	Description
vbKey0	48	0 key.
vbKey1	49	1 key.
vbKey2	50	2 key.
vbKey3	51	3 key.
vbKey4	52	4 key.
vbKey5	53	5 key.
vbKey6	54	6 key.
vbKey7	55	7 key.
vbKey8	56	8 key.
vbKey9	57	9 key.

Keys on the Numeric Keypad

Constant	Value	Description
vbKeyNumpad0	0x60	0 key.
vbKeyNumpad1	0x61	1 key.
vbKeyNumpad2	0x62	2 key.
vbKeyNumpad3	0x63	3 key.
vbKeyNumpad4	0x64	4 key.
vbKeyNumpad5	0x65	5 key.
vbKeyNumpad6	0x66	6 key.
vbKeyNumpad7	0x67	7 key.
vbKeyNumpad8	0x68	8 key.
vbKeyNumpad9	0x69	9 key.
vbKeyMultiply	0x6A	MULTIPLICATION SIGN (*)
vbKeyAdd	0x6B	PLUS SIGN (+)
vbKeySeparator	0x6C	ENTER key.
vbKeySubtract	0x6D	MINUS SIGN (-)
vbKeyDecimal	0x6E	DECIMAL POINT (.)
vbKeyDivide	0x6F	DIVISION SIGN (/)

Function Keys

Constant	Value	Description
vbKeyF1	0x70	F1 key.
vbKeyF2	0x71	F2 key.
vbKeyF3	0x72	F3 key.
vbKeyF4	0x73	F4 key.
vbKeyF5	0x74	F5 key.
vbKeyF6	0x75	F6 key.
vbKeyF7	0x76	F7 key.
vbKeyF8	0x77	F8 key.
vbKeyF9	0x78	F9 key.
vbKeyF10	0x79	F10 key.
vbKeyF11	0x7A	F11 key.
vbKeyF12	0x7B	F12 key.
vbKeyF13	0x7C	F13 key.
vbKeyF14	0x7D	F14 key.
vbKeyF15	0x7E	F15 key.
vbKeyF16	0x7F	F16 key.

Menu Accelerator Constants

Constant	Value	Description
vbMenuAccelCtrlA	1	User-defined shortcut keystrokes.
vbMenuAccelCtrlB	2	User-defined shortcut keystrokes.
vbMenuAccelCtrlC	3	User-defined shortcut keystrokes.
vbMenuAccelCtrlD	4	User-defined shortcut keystrokes.
vbMenuAccelCtrlE	5	User-defined shortcut keystrokes.
vbMenuAccelCtrlF	6	User-defined shortcut keystrokes.
vbMenuAccelCtrlG	7	User-defined shortcut keystrokes.
vbMenuAccelCtrlH	8	User-defined shortcut keystrokes.
vbMenuAccelCtrlI	9	User-defined shortcut keystrokes.
vbMenuAccelCtrlJ	10	User-defined shortcut keystrokes.
vbMenuAccelCtrlK	11	User-defined shortcut keystrokes.

Menu Accelerator Constants (continued)

Constant	Value	Description
vbMenuAccelCtrlL	12	User-defined shortcut keystrokes.
vbMenuAccelCtrlM	13	User-defined shortcut keystrokes.
vbMenuAccelCtrlN	14	User-defined shortcut keystrokes.
vbMenuAccelCtrlO	15	User-defined shortcut keystrokes.
vbMenuAccelCtrlP	16	User-defined shortcut keystrokes.
vbMenuAccelCtrlQ	17	User-defined shortcut keystrokes.
vbMenuAccelCtrlR	18	User-defined shortcut keystrokes.
vbMenuAccelCtrlS	19	User-defined shortcut keystrokes.
vbMenuAccelCtrlT	20	User-defined shortcut keystrokes.
vbMenuAccelCtrlU	21	User-defined shortcut keystrokes.
vbMenuAccelCtrlV	22	User-defined shortcut keystrokes.
vbMenuAccelCtrlW	23	User-defined shortcut keystrokes.
vbMenuAccelCtrlX	24	User-defined shortcut keystrokes.
vbMenuAccelCtrlY	25	User-defined shortcut keystrokes.
vbMenuAccelCtrlZ	26	User-defined shortcut keystrokes.
vbMenuAccelF1	27	User-defined shortcut keystrokes.
vbMenuAccelF2	28	User-defined shortcut keystrokes.
vbMenuAccelF3	29	User-defined shortcut keystrokes.
vbMenuAccelF4	30	User-defined shortcut keystrokes.
vbMenuAccelF5	31	User-defined shortcut keystrokes.
vbMenuAccelF6	32	User-defined shortcut keystrokes.
vbMenuAccelF7	33	User-defined shortcut keystrokes.

Menu Accelerator Constants (continued)

Constant	Value	Description
vbMenuAccelF8	34	User-defined shortcut keystrokes.
vbMenuAccelF9	35	User-defined shortcut keystrokes.
vbMenuAccelF11	36	User-defined shortcut keystrokes.
vbMenuAccelF12	37	User-defined shortcut keystrokes.
vbMenuAccelCtrlF1	38	User-defined shortcut keystrokes.
vbMenuAccelCtrlF2	39	User-defined shortcut keystrokes.
vbMenuAccelCtrlF3	40	User-defined shortcut keystrokes.
vbMenuAccelCtrlF4	41	User-defined shortcut keystrokes.
vbMenuAccelCtrlF5	42	User-defined shortcut keystrokes.
vbMenuAccelCtrlF6	43	User-defined shortcut keystrokes.
vbMenuAccelCtrlF7	44	User-defined shortcut keystrokes.
vbMenuAccelCtrlF8	45	User-defined shortcut keystrokes.
vbMenuAccelCtrlF9	46	User-defined shortcut keystrokes.
vbMenuAccelCtrlF11	47	User-defined shortcut keystrokes.
vbMenuAccelCtrlF12	48	User-defined shortcut keystrokes.
vbMenuAccelShiftF1	49	User-defined shortcut keystrokes.
vbMenuAccelShiftF2	50	User-defined shortcut keystrokes.
vbMenuAccelShiftF3	51	User-defined shortcut keystrokes.
vbMenuAccelShiftF4	52	User-defined shortcut keystrokes.
vbMenuAccelShiftF5	53	User-defined shortcut keystrokes.
vbMenuAccelShiftF6	54	User-defined shortcut keystrokes.
vbMenuAccelShiftF7	55	User-defined shortcut keystrokes.

Menu Accelerator Constants (continued)

Constant	Value	Description
vbMenuAccelShiftF8	56	User-defined shortcut keystrokes.
vbMenuAccelShiftF9	57	User-defined shortcut keystrokes.
vbMenuAccelShiftF11	58	User-defined shortcut keystrokes.
vbMenuAccelShiftF12	59	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF1	60	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF2	61	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF3	62	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF4	63	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF5	64	ser-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF6	65	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF7	66	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF8	67	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF9	68	ser-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF11	69	User-defined shortcut keystrokes.
vbMenuAccelShiftCtrlF12	70	User-defined shortcut keystrokes.
vbMenuAccelCtrlIns	71	User-defined shortcut keystrokes.
vbMenuAccelShiftIns	72	User-defined shortcut keystrokes.
vbMenuAccelDel	73	User-defined shortcut keystrokes.
vbMenuAccelShiftDel	74	User-defined shortcut keystrokes.
vbMenuAccelAltBksp	75	User-defined shortcut keystrokes.

Menu Control Constants

PopupMenu Method Alignment

Constant	Value	Description
vbPopupMenuLeftAlign	0	Pop-up menu left-aligned.
vbPopupMenuCenterAlign	4	Pop-up menu centered.
vbPopupMenuRightAlign	8	Pop-up menu right-aligned.

PopupMenu Mouse Button Recognition

Constant	Value	Description
vbPopupMenuLeftButton	0	Pop-up menu recognizes left mouse button only.
vbPopupMenuRightButton	2	Pop-up menu recognizes right and left mouse buttons.

Miscellaneous Constants

ZOrder Method

Constant	Value	Description
vbBringToFront	0	Bring to front.
vbSendToBack	1	Send to back.

QueryUnload Method

Constant	Value	Description
vbAppWindows	2	Current Windows session ending.
vbFormMDIForm	4	MDI child form is closing because the MDI form is closing.
vbFormCode	1	Unload method invoked from code.
vbFormControlMenu	0	User has chosen Close command from the Control-menu box on a form.
vbAppTaskManager	3	Windows Task Manager is closing the application.

Shift Parameter Masks

Constant	Value	Description
vbShiftMask	1	SHIFT key bit mask.
vbCtrlMask	2	CTRL key bit mask.
vbAltMask	4	ALT key bit mask.

Button Parameter Masks

Constant	Value	Description
vbLeftButton	1	Left mouse button.
vbRightButton	2	Right mouse button.
vbMiddleButton	4	Middle mouse button.

Application Start Mode

Constant	Value	Description
vbSModeStandalone	0	Stand-alone application.
vbSModeAutomation	1	OLE automation server.

LoadResPicture Method

Constant	Value	Description
vbResBitmap	0	Bitmap resource.
vbResIcon	1	Icon resource.
vbResCursor	2	Cursor resource.

Check Value

Constant	Value	Description
vbUnchecked	0	Unchecked.
vbChecked	1	Checked.
vbGrayed	2	Grayed.

Mouse Pointer Constants

Constant	Value	Description
vbDefault	0	Default.
vbArrow	1	Arrow.
vbCrosshair	2	Cross.
vbIbeam	3	I beam.
vbIconPointer	4	Icon.
vbSizePointer	5	Size.
vbSizeNESW	6	Size NE, SW.
vbSizeNS	7	Size N, S.
vbSizeNWSE	8	Size NW , SE.
vbSizeWE	9	Size W, E.
vbUpArrow	10	Up arrow.
vbHourglass	11	Hourglass.
vbNoDrop	12	No drop.
vbArrowHourglass	13	Arrow and hourglass. (Only available in 32-bit Visual Basic 4.0.)
vbArrowQuestion	14	Arrow and question mark. (Only available in 32-bit Visual Basic 4.0.)
vbSizeAll	15	Size all. (Only available in 32-bit Visual Basic 4.0.)
vbCustom	99	Custom icon specified by the MouseIcon property.

MsgBox Constants

MsgBox Arguments

Constant	Value	Description
vbOKOnly	0	OK button only (default)
vbOKCancel	1	OK and Cancel buttons.
vbAbortRetryIgnore	2	Abort, Retry, and Ignore buttons.
vbYesNoCancel	3	Yes, No, and Cancel buttons.
vbYesNo	4	Yes and No buttons.
vbRetryCancel	5	Retry and Cancel buttons.
vbCritical	16	Critical message.
vbQuestion	32	Warning query.
vbExclamation	48	Warning message.
vbInformation	64	Information message.
vbDefaultButton1	0	First button is default (default)
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbApplicationModal	0	Application modal message box (default)
vbSystemModal	4096	System modal message box.

MsgBox Return Values

Constant	Value	Description
vbOK	1	OK button pressed.
vbCancel	2	Cancel button pressed.
vbAbort	3	Abort button pressed.
vbRetry	4	Retry button pressed.
vbIgnore	5	Ignore button pressed.
vbYes	6	Yes button pressed.
vbNo	7	No button pressed.

OLE Container Control Constants

OLEType Property

Constant	Value	Description
vbOLELinked	0	OLE container control contains a linked object.
vbOLEEmbedded	1	OLE container control contains an embedded object.
vbOLENone	3	OLE container control doesn't contain an object.

OLETypeAllowed Property

Constant	Value	Description
vbOLEEither	2	OLE container control can contain either a linked or an embedded object.

UpdateOptions Property

Constant	Value	Description
vbOLEAutomatic	0	Object is updated each time the linked data changes.
vbOLEFrozen	1	Object is updated whenever the user saves the linked document from within the application in which it was created.
vbOLEManual	2	Object is updated only when the Action property is set to 6 (Update)

AutoActivate Property

Constant	Value	Description
vbOLEActivateManual	0	OLE object isn't automatically activated.
vbOLEActivateGetFocus	1	Object is activated when the OLE container control gets the focus.
vbOLEActivateDoubleclick	2	Object is activated when the OLE container control is double-clicked.
vbOLEActivateAuto	3	Object is activated based on the object's default method of activation.

SizeMode Property

Constant	Value	Description
vbOLESizeClip	0	Object's image is clipped by the OLE container control's borders.
vbOLESizeStretch	1	Object's image is sized to fill the OLE container control.
vbOLESizeAutoSize	2	OLE container control is automatically resized to display the entire object.
vbOLESizeZoom	3	Object's image is stretched but in proportion.

DisplayType Property

Constant	Value	Description
vbOLEDisplayContent	0	Object's data is displayed in the OLE container control.
vbOLEDisplayIcon	1	Object's icon is displayed in the OLE container control.

Updated Event Constants

Constant	Value	Description
vbOLEChanged	0	Object's data has changed.
vbOLESaved	1	Object's data has been saved by the application that created the object.
vbOLEClosed	2	Application file containing the linked object's data has been closed.
vbOLERenamed	3	Application file containing the linked object's data has been renamed.

Special Verb Values

Constant	Value	Description
vbOLEPrimary	0	Default action for the object.
vbOLEShow	-1	Activates the object for editing.
vbOLEOpen	-2	Opens the object in a separate application window.
vbOLEHide	-3	For embedded objects, hides the application that created the object.
vbOLEInPlaceUIActivate	-4	All UI's associated with the object are visible and ready for use.
vbOLEInPlaceActivate	-5	Object is ready for the user to click inside it and start working with it.
vbOLEDiscardUndoState	-6	For discarding all record of changes that the object's application can undo.

Verb Flag Bit Masks

Constant	Value	Description
vbOLEFlagEnabled	0x0	Enabled menu item.
vbOLEFlagGrayed	0x1	Grayed menu item.
vbOLEFlagDisabled	0x2	Disabled menu item.
vbOLEFlagChecked	0x8	Checked menu item.
vbOLEFlagSeparator	0x800	Separator bar in menu item list.
vbOLEMiscFlagMemStorage	0x1	Causes control to use memory to store the object while it's loaded.
vbOLEMiscFlagDisableInPlace	0x2	Forces OLE container control to activate objects in a separate window.

VBTranslateColor/OLETranslateColor Constants

Constant	Value	Description
vbInactiveCaptionText	0x80000013	Color of text in an inactive caption.
vb3DHighlight	0x80000014	Highlight color for 3-D display elements.
vb3DFace	0x8000000F	Dark shadow color for 3-D display elements.
vbMsgBox	0x80000017	Background color for message boxes and system dialog boxes.
vbMsgBoxText	0x80000018	Color of text displayed in message boxes and system dialog boxes.
vb3DShadow	0x80000010	Color of automatic window shadows.
vb3DDKShadow	0x80000015	Darkest shadow.
vb3DLight	0x80000016	Second lightest of the 3-D colors (after vb3DHighlight)

Picture Object Constants

Constant	Value	Description
vbPicTypeBitmap	1	Bitmap type of Picture object.
vbPicTypeMetafile	2	Metafile type of Picture object.
vbPicTypeIcon	3	Icon type of Picture object.

Printer Object Constants

Printer Color Mode

Constant	Value	Description
vbPRCMMonochrome	1	Monochrome output.
vbPRCMColor	2	Color output.

Duplex Printing

Constant	Value	Description
vbPRDPSimplex	1	Single-sided printing.
vbPRDPHorizontal	2	Double-sided horizontal printing.
vbPRDPVertical	3	Double-sided vertical printing.

Printer Orientation

Constant	Value	Description
vbPRORPortrait	1	Documents print with the top at the narrow side of the paper.
vbPRORLandscape	2	Documents print with the top at the wide side of the paper.

Print Quality

Constant	Value	Description
vbPRPQDraft	-1	Draft print quality.
vbPRPQLow	-2	Low print quality.
vbPRPQMedium	-3	Medium print quality.
vbPRPQHigh	-4	High print quality.

PaperBin Property

Constant	Value	Description
vbPRBNUpper	1	Use paper from the upper bin.
vbPRBNLower	2	Use paper from the lower bin.
vbPRBNMiddle	3	Use paper from the middle bin.
vbPRBNManual	4	Wait for manual insertion of each sheet of paper.
vbPRBNEnvelope	5	Use envelopes from the envelope feeder.
vbPRBNEnvManual	6	Use envelopes from the envelope feeder, but wait for manual insertion.
vbPRBNAuto	7	(Default)
vbPRBNTractor	8	Use paper fed from the tractor feeder.

PaperBin Property (continued)

Constant	Value	Description
vbPRBNSmallFmt	9	Use paper from the small paper feeder.
vbPRBNLargeFmt	10	Use paper from the large paper bin.
vbPRBNLargeCapacity	11	Use paper from the large capacity feeder.
vbPRBNCassette	14	Use paper from the attached cassette cartridge.

PaperSize Property

Constant	Value	Description
vbPRPSLetter	1	Letter, 8 1/2 x 11 in.
vbPRPSLetterSmall	2	+A611Letter Small, 8 1/2 x 11 in.
vbPRPSTabloid	3	Tabloid, 11 x 17 in.
vbPRPSLedger	4	Ledger, 17 x 11 in.
vbPRPSLegal	5	Legal, 8 1/2 x 14 in.
vbPRPSStatement	6	Statement, 5 1/2 x 8 1/2 in.
vbPRPSExecutive	7	Executive, 7 1/2 x 10 1/2 in.
vbPRPSA3	8	A3, 297 x 420 mm.
vbPRPSA4	9	A4, 210 x 297 mm.
vbPRPSA4Small	10	A4 Small, 210 x 297 mm.
vbPRPSA5	11	A5, 148 x 210 mm.
vbPRPSB4	12	B4, 250 x 354 mm.
vbPRPSB5	13	B5, 182 x 257 mm.
vbPRPSFolio	14	Folio, 8 1/2 x 13 in.
vbPRPSQuarto	15	Quarto, 215 x 275 mm.
vbPRPS10x14	16	10 x 14 in.
vbPRPS11x17	17	11 x 17 in.
vbPRPSNote	18	Note, 8 1/2 x 11 in.
vbPRPSEnv9	19	Envelope #9, 3 7/8 x 8 7/8 in.
vbPRPSEnv10	20	Envelope #10, 4 1/8 x 9 1/2 in.
vbPRPSEnv11	21	Envelope #11, 4 1/2 x 10 3/8 in.
vbPRPSEnv12	22	Envelope #12, 4 1/2 x 11 in.
vbPRPSEnv14	23	Envelope #14, 5 x 11 1/2 in.
vbPRPSCSheet	24	C size sheet.
vbPRPSDSheet	25	D size sheet.
vbPRPSESheet	26	E size sheet.
vbPRPSEnvDL	27	Envelope DL, 110 x 220 mm.
vbPRPSEnvC3	29	Envelope C3, 324 x 458 mm.
vbPRPSEnvC4	30	Envelope C4, 229 x 324 mm.
vbPRPSEnvC5	28	Envelope C5, 162 x 229 mm.
vbPRPSEnvC6	31	Envelope C6, 114 x 162 mm.

| vbPRPSEnvC65 | 32 | Envelope C65, 114 x 229 mm. |

PaperSize Property (continued)

Constant	Value	Description
vbPRPSEnvB4	33	Envelope B4, 250 x 353 mm.
vbPRPSEnvB5	34	Envelope B5, 176 x 250 mm.
vbPRPSEnvB6	35	Envelope B6, 176 x 125 mm.
vbPRPSEnvItaly	36	Envelope, 110 x 230 mm.
vbPRPSEnvMonarch	37	Envelope Monarch, 3 7/8 x 7 1/2 in.
vbPRPSEnvPersonal	38	Envelope, 3 5/8 x 6 1/2 in.
vbPRPSFanfoldUS	39	U.S. Standard Fanfold, 14 7/8 x 11 in.
vbPRPSFanfoldStdGerman	40	German Standard Fanfold, 8 1/2 x 12 in.
vbPRPSFanfoldLglGerman	41	German Legal Fanfold, 8 1/2 x 13 in.
vbPRPSUser	256	User-defined.

RasterOp Constants

Constant	Value	Description
vbDstInvert	0x00550009	Inverts the destination bitmap.
vbMergeCopy	0x00C000CA	Combines the pattern and the source bitmap.
vbMergePaint	0x00BB0226	Combines the inverted source bitmap with the destination bitmap by using Or.
vbNotSrcCopy	0x00330008	Copies the inverted source bitmap to the destination.
vbNotSrcErase	0x001100A6	Inverts the result of combining the destination and source bitmaps by using Or.
vbPatCopy	0x00F00021L	Copies the pattern to the destination bitmap.
vbPatInvert	0x005A0049L	Combines the destination bitmap with the pattern by using Xor.
vbPatPaint	0x00FB0A09L	Combines the inverted source bitmap with the pattern by using Or. Combines the result of this operation with the destination bitmap by using Or.
vbSrcAnd	0x008800C6	Combines pixels of the destination and source bitmaps by using And.

RasterOp Constants (continued)

Constant	Value	Description
vbSrcCopy	0x00CC0020	Copies the source bitmap to the destination bitmap.
vbSrcErase	0x00440328	Inverts the destination bitmap and combines the result with the source bitmap by using And.
vbSrcInvert	0x00660046	Combines pixels of the destination and source bitmaps by using Xor.
vbSrcPaint	0x00EE0086	Combines pixels of the destination and source bitmaps by using Or.

Shell Constants

Constant	Value	Description
vbHide	0	Window is hidden and focus is passed to the hidden window.
vbNormalFocus	1	Window has focus and is restored to its original size and position.
vbMinimizedFocus	2	Window is displayed as an icon with focus.
vbMaximizedFocus	3	Window is maximized with focus.
vbNormalNoFocus	4	Window is restored to its most recent size and position. The currently active window remains active.
vbMinimizedNoFocus	6	Window is displayed as an icon. The currently active window remains active.

StrConv Constants

Constant	Value	Description
vbUpperCase	1	Uppercases the string.
vbLowerCase	2	Lowercases the string.
vbProperCase	3	Uppercases first letter of every word in string.
vbWide*	4*	Converts narrow (single-byte)(double-byte)
vbNarrow*	8*	Converts wide (double-byte)(single-byte)
vbKatakana**	16**	Converts Hiragana characters in string to Katakana characters.
vbHiragana**	32**	Converts Katakana characters in string to Hiragana characters.
vbUnicode***	64***	Converts the string to Unicode using the default code page of the system.
vbFromUnicode***	128***	Converts the string from Unicode to the default code page of the system.

_____

*Applies to Far East locales
**Applies to Japan only.
***Specifying this bit on 16-bit systems causes a run-time error
.

Variant Type Constants

Constant	Value	Description
vbVEmpty	0	Empty (uninitialized)
vbVNull	1	Null (no valid data)
vbVInteger	2	Integer data type.
vbVLong	3	Long integer data type.
vbVSingle	4	Single-precision floating-point data type.
vbVDouble	5	Double-precision floating-point data type.
vbVCurrency	6	Currency (scaled integer)
vbVDate	7	Date data type.
vbVString	8	String data type.

VarType Constants

Constant	Value	Description
vbEmpty	0	Uninitialized (default)
vbNull	1	Contains no valid data.
vbInteger	2	Integer.
vbLong	3	Long integer.
vbSingle	4	Single-precision floating-point number.
vbDouble	5	Double-precision floating-point number.
vbCurrency	6	Currency.
vbDate	7	Date.
vbString	8	String.
vbObject	9	OLE Automation object.
vbError	10	Error.
vbBoolean	11	Boolean.
vbVariant	12	Variant (used only for arrays of Variants)
vbDataObject	13	Non-OLE Automation object.
vbByte	17	Byte
vbArray	8192	Array.

Programming Microsoft Windows with Visual Basic

Appendix II. Common Dialog Box Constants

CommonDialog Control Constants

File Open/Save Dialog Box Flags

Constant	Value	Description
cdlOFNReadOnly	0x1	Checks Read-Only check box for Open and Save As dialog boxes.
cdlOFNOverwritePrompt	0x2	Causes the Save As dialog box to generate a message box if the selected file already exists.
cdlOFNHideReadOnly	0x4	Hides the Read-Only check box.
cdlOFNNoChangeDir	0x8	Sets the current directory to what it was when the dialog box was invoked.

cdlOFNHelpButton	0x10	Causes the dialog box to display the Help button.
cdlOFNNoValidate	0x100	Allows invalid characters in the returned filename.
cdlOFNAllowMultiselect	0x200	Allows the File Name list box to have multiple selections.
cdlOFNExtensionDifferent	0x400	The extension of the returned filename is different from the extension set by the DefaultExt property.
cdlOFNPathMustExist	0x800	User can enter only valid path names.
cdlOFNFileMustExist	0x1000	User can enter only names of existing files.
cdlOFNCreatePrompt	0x2000	Sets the dialog box to ask if the user wants to create a file that doesn't currently exist.
cdlOFNShareAware	0x4000	Sharing violation errors will be ignored.
cdlOFNNoReadOnlyReturn	0x8000	The returned file doesn't have the Read-Only attribute set and won't be in a write-protected directory.

File Open/Save Dialog Box Flags (continued)

Constant	Value	Description
cdlOFNExplorer	0x0008000	Use the Explorer-like Open A File dialog box template. (Windows 95 only.)
cdlOFNNoDereferenceLinks	0x00100000	Do not dereference shortcuts (shell links) default, choosing a shortcut causes it to be dereferenced by the shell. (Windows 95 only.)
cdlOFNLongNames	0x00200000	Use Long filenames. (Windows 95 only.)

Color Dialog Box Flags

Constant	Value	Description
cdlCCRGBInit	0x1	Sets initial color value for the dialog box.
cdlCCFullOpen	0x2	Entire dialog box is displayed, including the Define Custom Colors section.
cdlCCPreventFullOpen	0x4	Disables the Define Custom Colors section of the dialog box.
cdlCCHelpButton	0x8	Dialog box displays a Help button.

Fonts Dialog Box Flags

Constant	Value	Description
cdlCFScreenFonts	0x1	Dialog box lists only screen fonts supported by the system.
cdlCFPrinterFonts	0x2	Dialog box lists only fonts supported by the printer.
cdlCFBoth	0x3	Dialog box lists available screen and printer fonts.
cdlCFHelpButton	0x4	Dialog box displays a Help button.
cdlCFEffects	0x100	Dialog box enables strikeout, underline, and color effects.
cdlCFApply	0x200	Dialog box enables the Apply button.
cdlCFANSIOnly	0x400	Dialog box allows only a selection of fonts that use the Windows character set.
cdlCFNoVectorFonts	0x800	Dialog box should not allow vector-font selections.

Fonts Dialog Box Flags (continued)

Constant	Value	Description
cdlCFNoSimulations	0x1000	Dialog box should not allow graphic device interface (GDI)
cdlCFLimitSize	0x2000	Dialog box should select only font sizes within the range specified by the Min and Max properties.
cdlCFFixedPitchOnly	0x4000	Dialog box should select only fixed-pitch fonts.
cdlCFWYSIWYG	0x8000	Dialog box should allow only the selection of fonts available to both the screen and printer.
cdlCFForceFontExist	0x10000	An error dialog box is displayed if a user selects a font or style that doesn't exist.
cdlCFScalableOnly	0x20000	Dialog box should allow only the selection of scalable fonts.
cdlCFTTOnly	0x40000	Dialog box should allow only the selection of TrueType fonts.
cdlCFNoFaceSel	0x80000	No font name selected.
cdlCFNoStyleSel	0x100000	No font style selected.
cdlCFNoSizeSel	0x200000	No font size selected.

Printer Dialog Box Flags

Constant	Value	Description
cdlPDAllPages	0x0	Returns or sets state of All Pages option button.
cdlPDCollate	0x10	Returns or sets state of Collate check box.
cdlPDDisablePrintToFile	0x80000	Disables the Print To File check box.
cdlPDHidePrintToFile	0x100000	The Print To File check box isn't displayed.
cdlPDNoPageNums	0x8	Returns or sets the state of the Pages option button.
cdlPDNoSelection	0x4	Disables the Selection option button.
cdlPDNoWarning	0x80	Prevents a warning message when there is no default printer.
cdlPDPageNums	0x2	Returns or sets the state of the Pages option button.
cdlPDPrintSetup	0x40	Displays the Print Setup dialog box rather than the Print dialog box.

Printer Dialog Box Flags (continued)

Constant	Value	Description
cdlPDPrintToFile	0x20	Returns or sets the state of the Print To File check box.
cdlPDReturnDC	0x100	Returns a device context for the printer selection value returned in the hDC property of the dialog box.
cdlPDReturnDefault	0x400	Returns default printer name.
cdlPDReturnIC	0x200	Returns an information context for the printer selection value returned in the hDC property of the dialog box.
cdlPDSelection	0x1	Returns or sets the state of the Selection option button.
cdlPDHelpButton	0x800	Dialog box displays the Help button.
cdlPDUseDevModeCopies	0x40000	Sets support for multiple copies action; depends upon whether or not printer supports multiple copies.

CommonDialog Error Constants

Constant	Value	Description
cdlAlloc	&H7FF0&	Couldn't allocate memory for FileName or Filter property.
cdlCancel	&H7FF3&	Cancel was selected.
cdlDialogFailure	&H8000&	The function failed to load the dialog box.
cdlFindResFailure	&H7FF9&	The function failed to load a specified resource.
cdlHelp	&H7FEF&	Call to Windows Help failed.
cdlInitialization	&H7FFD&	The function failed during initialization.
cdlLoadResFailure	&H7FF8&	The function failed to load a specified string.
cdlLockResFailure	&H7FF7&	The function failed to lock a specified resource.
cdlMemAllocFailure	&H7FF6&	The function was unable to allocate memory for internal data structures.
cdlMemLockFailure	&H7FF5&	The function was unable to lock the memory associated with a handle.
cdlNoFonts	&H5FFE&	No fonts exist.
cdlBufferTooSmall	&H4FFC&	The buffer at which the member lpstrFile points is too small.
cdlInvalidFileName	&H4FFD&	Filename is invalid.
cdlSubclassFailure	&H4FFE&	An attempt to subclass a list box failed due to insufficient memory.
cdlCreateICFailure	&H6FF5&	The PrintDlg function failed when it attempted to create an information context.
cdlDndmMismatch	&H6FF6&	Data in the DevMode and DevNames data structures describe two different printers.
cdlGetDevModeFail	&H6FFA&	The printer device driver failed to initialize a DevMode data structure.
cdlInitFailure	&H6FF9&	The PrintDlg function failed during initialization.
cdlLoadDrvFailure	&H6FFB&	The PrintDlg function failed to load the specified printer's device driver.

CommonDialog Error Constants (continued)

Constant	Value	Description
cdlNoDefaultPrn	&H6FF7&	A default printer doesn't exist.
cdlNoDevices	&H6FF8&	No printer device drivers were found.
cdlParseFailure	&H6FFD&	The CommonDialog function failed to parse the strings in the [devices] section of WIN.INI.
cdlPrinterCodes	&H6FFF&	The PDReturnDefault flag was set, but either the hDevMode or hDevNames field was nonzero.
cdlPrinterNotFound	&H6FF4&	The [devices] section of WIN.INI doesn't contain an entry for the requested printer.
cdlRetDefFailure	&H6FFC&	The PDReturnDefault flag was set, but either the hDevMode or hDevNames field was nonzero.
cdlSetupFailure	&H6FFE&	Failed to load required resources.