

Lecture - 4 -

Algorithms:

A step-by-step procedure for solving a problem in a finite amount of time.
Algorithms can be represented using Flow Charts.

CHARACTERISTICS OF AN ALGORITHM:

1. Algorithms always have a definite starting point and an end point. These points are generally marked with the words like Start, Begin, End, Stop etc.
2. They consist of finite number of steps.
3. They always relate to a specific problem or you can say that they are written for a given problem.
4. They serve as *foundation stone for programming*
5. They are written in easy language.

Example 1: Write algorithm to calculate the area of circle.

Algorithm:

1. Start
2. Read value of radius(R)
3. Compute area of circle $(A)=3.14*R*R$ (Algebraic equation)
4. Print A
5. End

Example 2: Write algorithm to convert the length in feet to centimeter.

Algorithm:

1. Start
2. Read value of length in feet (ft)
3. Compute length in centimeter $(cm)=ft*30$ (Algebraic equation)
4. Print length - in - cm
5. End

Example 3:

Write algorithm to computes the sum,average and product of three numbers.

Algorithm:

1. Start
2. Read value of X
3. Read value of Y
4. Read value of Z
5. Compute Sum (S) as $X+Y+Z$
6. Compute Average(A) sa $S/3$
7. Compute Product (P) as $X*Y*Z$
8. Print (Display) the Sum,Average and Product
9. End

Example.4.

Write an algorithm to find the largest of given three numbers.

Algorithm:

1. Start
2. Read three numbers A, B and C
3. Let Big=0
4. IF $A>B$ Then Big=A Else Big=B
5. IF $C>Big$ Then Big=C
6. Print Big
7. End

Example 5 : Write an algorithm for solving a given quadratic equation, $ax^2+bx+c=0$. Note that roots are determined by formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Algorithm:

1. Start
2. Read value of a, b and c
3. If $a=0$ Stop
4. Calculate values of discriminant $D=b^2-4ac$
5. If $D=0$ then there is one root $p= -b/2a$
6. If $D>0$ then there are two real roots

$$p = \frac{-b+\sqrt{D}}{2a} \quad \text{and} \quad q = \frac{-b-\sqrt{D}}{2a}$$

7. If $D<0$ then there are two complex roots

$$p = \frac{-b+i\sqrt{-D}}{2a} \quad \text{and} \quad q = \frac{-b-i\sqrt{-D}}{2a}$$

8. Print p and q
9. Stop

Lecture - 5 -

FLOWCHARTS


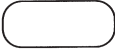
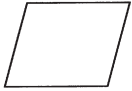

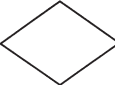



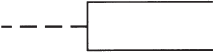
When a step - by - step solution of a given problem is illustrated in the form of graphical chart that chart is called flowchart

FLOWCHARTS SYMBOLS

A flowchart consists of special geometric symbols connected by arrows. Within each symbol is a phrase presenting the activity at that step. The shape of the symbol indicates the type of operation that is to occur. For instance, the parallelogram denotes input or output. The arrows connecting the symbols, called flowlines, show the progression in which the steps take place.

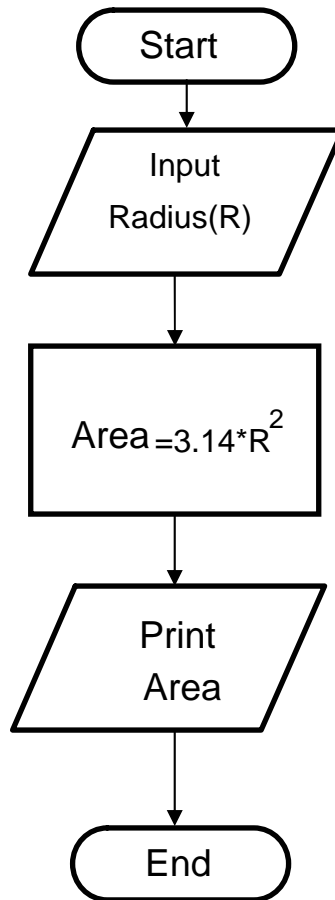
Flowcharts should flow from the top of the page to the bottom. Although the symbols used in flowcharts are standardized, no standards exist for the amount of detail required within each symbol. A table of the flowchart symbols adopted by the American National Standards Institute (ANSI) follows.

The main advantage of using a flowchart to plan a task is that it provides a pictorial representation of the task, which makes the logic easier to follow. We can clearly see every step and how each step is connected to the next. The major disadvantage with flowcharts is that when a program is very large, the flowcharts may continue for many pages, making them difficult to follow and modify.

Symbol	Name	Meaning
	<i>Flowline</i>	Used to connect symbols and indicate the flow of logic.
	<i>Terminal</i>	Used to represent the beginning (Start) or the end (End) of a task.
	<i>Input/Output</i>	Used for input and output operations, such as reading and printing. The data to be read or printed are described inside.
	<i>Processing</i>	Used for arithmetic and data-manipulation operations. The instructions are listed inside the symbol.
	<i>Decision</i>	Used for any logic or comparison operations. Unlike the input/output and processing symbols, which have one entry and one exit flowline, the decision symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is "yes" or "no."
	<i>Connector</i>	Used to join different flowlines.
	<i>Offpage Connector</i>	Used to indicate that the flowchart continues to a second page.
	<i>Predefined Process</i>	Used to represent a group of statements that perform one processing task.
	<i>Annotation</i>	Used to provide additional information about another flowchart symbol.

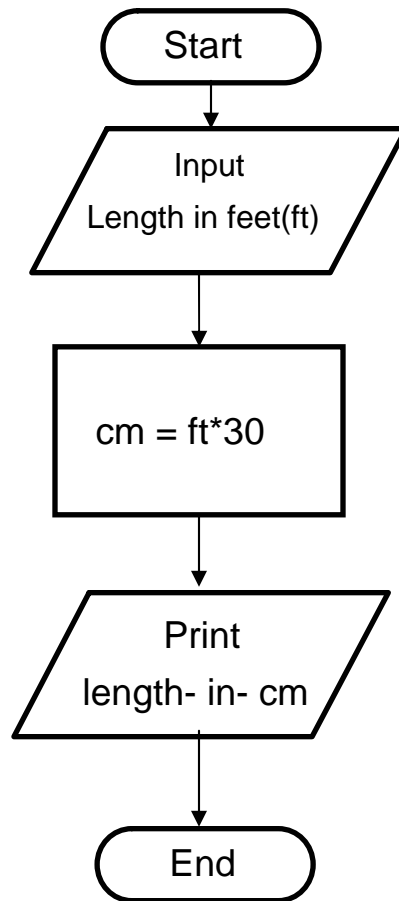
Example (1)

Draw the flowchart to calculate the area of circle ?



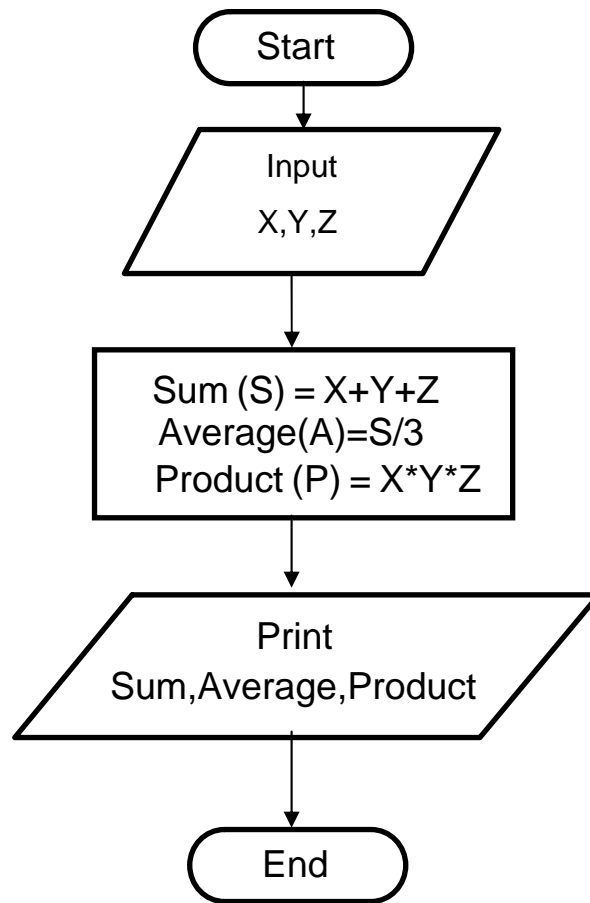
Example (2)

Draw the flowchart to convert the length in feet to centimeter?



Example (3)

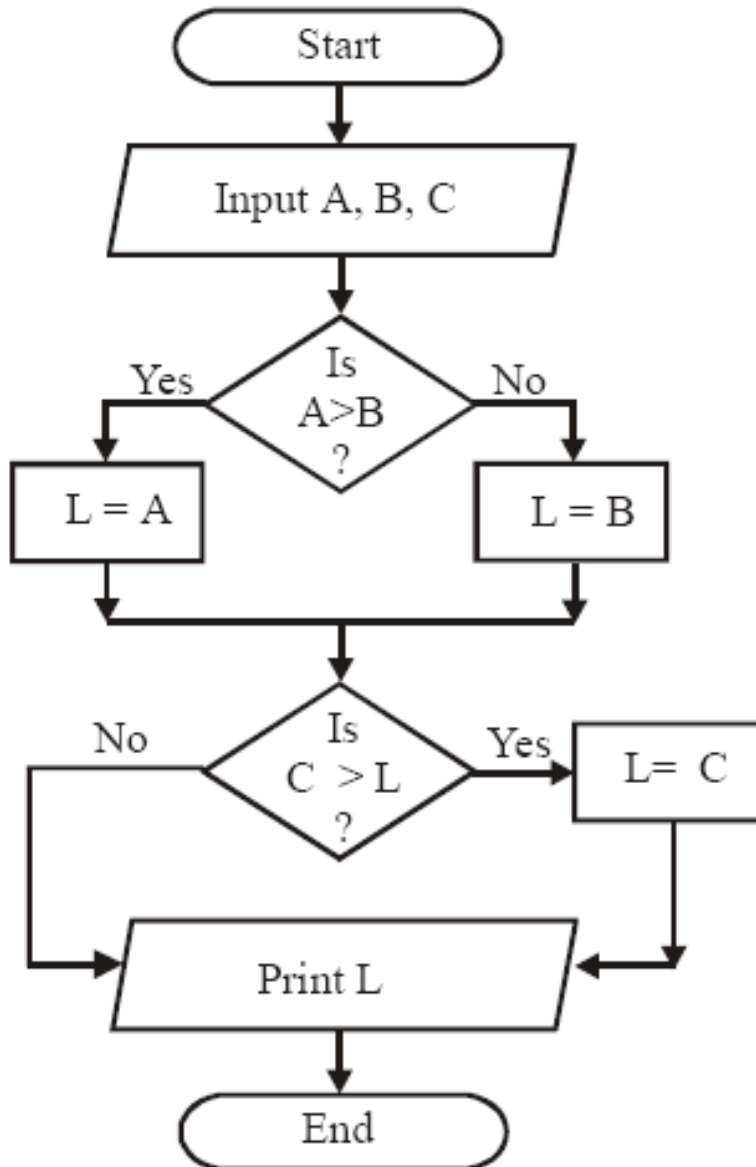
Draw the flowchart to computes the sum, average and product of three numbers.



Lecture - 6 -

Example (4)

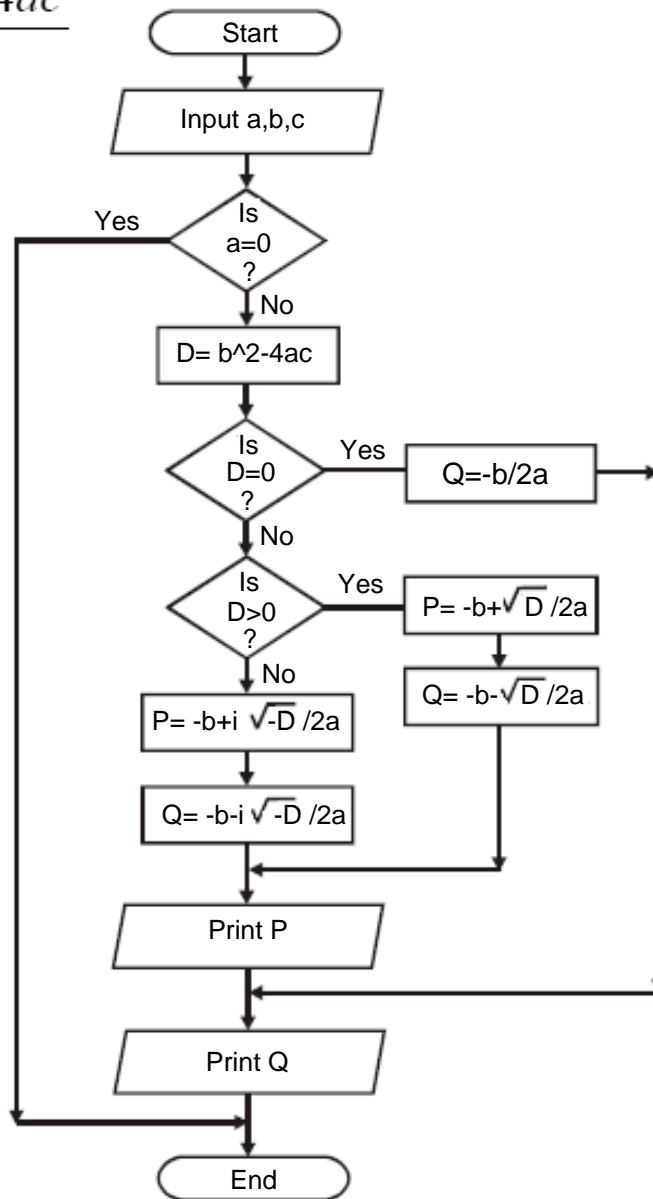
Draw a flowchart to find the larger of the three given numbers.



Example (5)

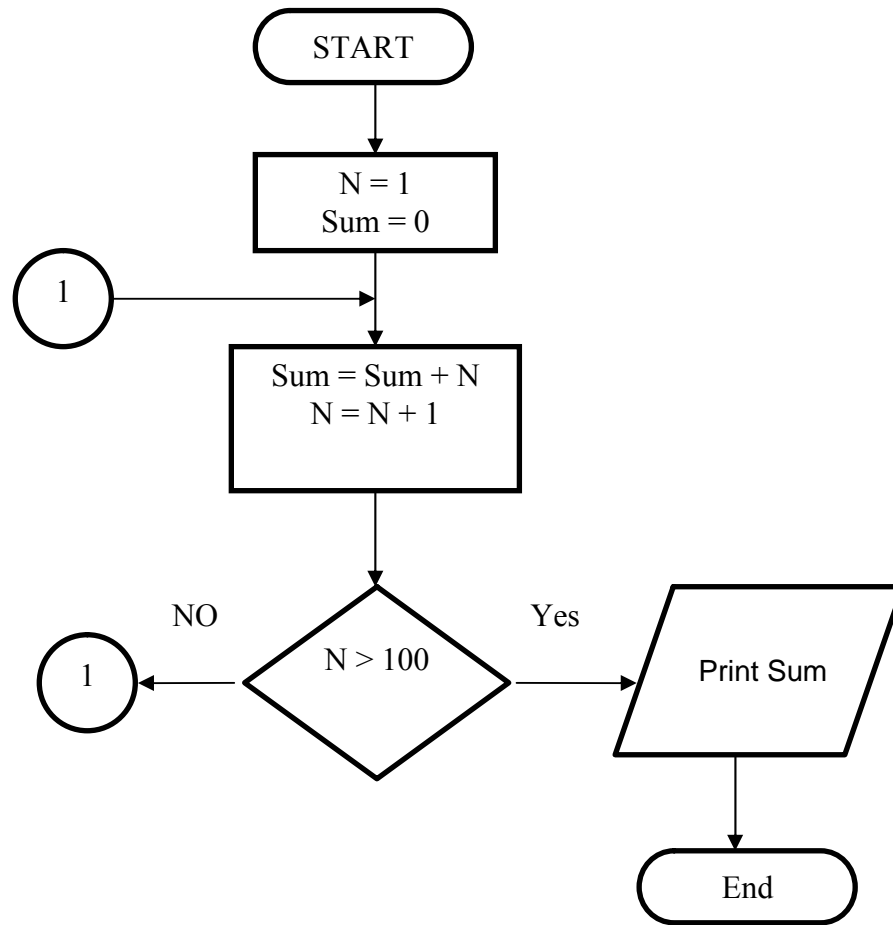
Draw a flowchart for solving a given quadratic equation $ax^2+bx+c=0$.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



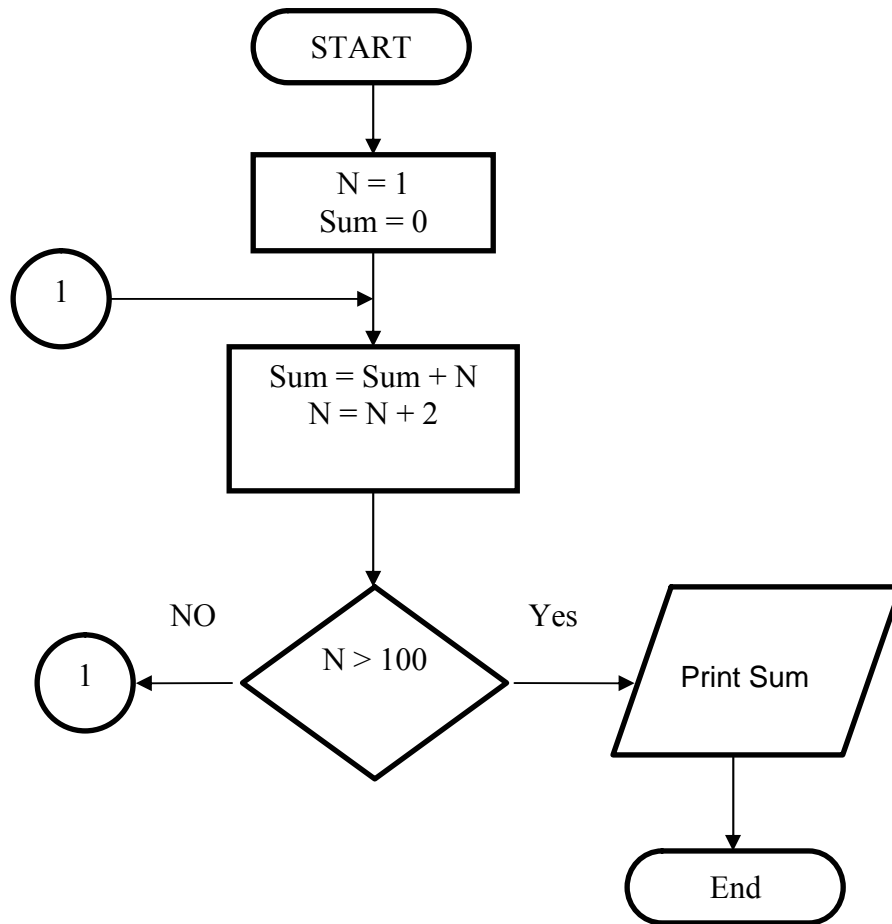
Example (6)

Draw a flowchart for find the sum of numbers from 1.....100 ?



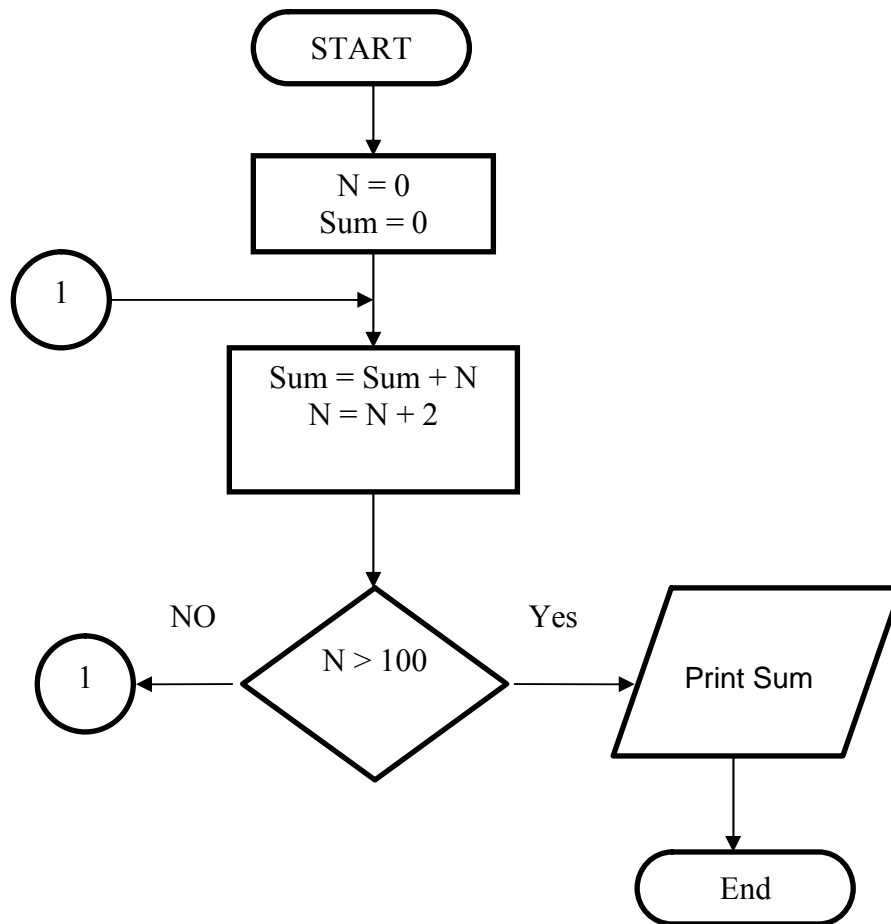
Example (7)

Draw a flowchart for find the sum of odd numbers from 1.....100 ?



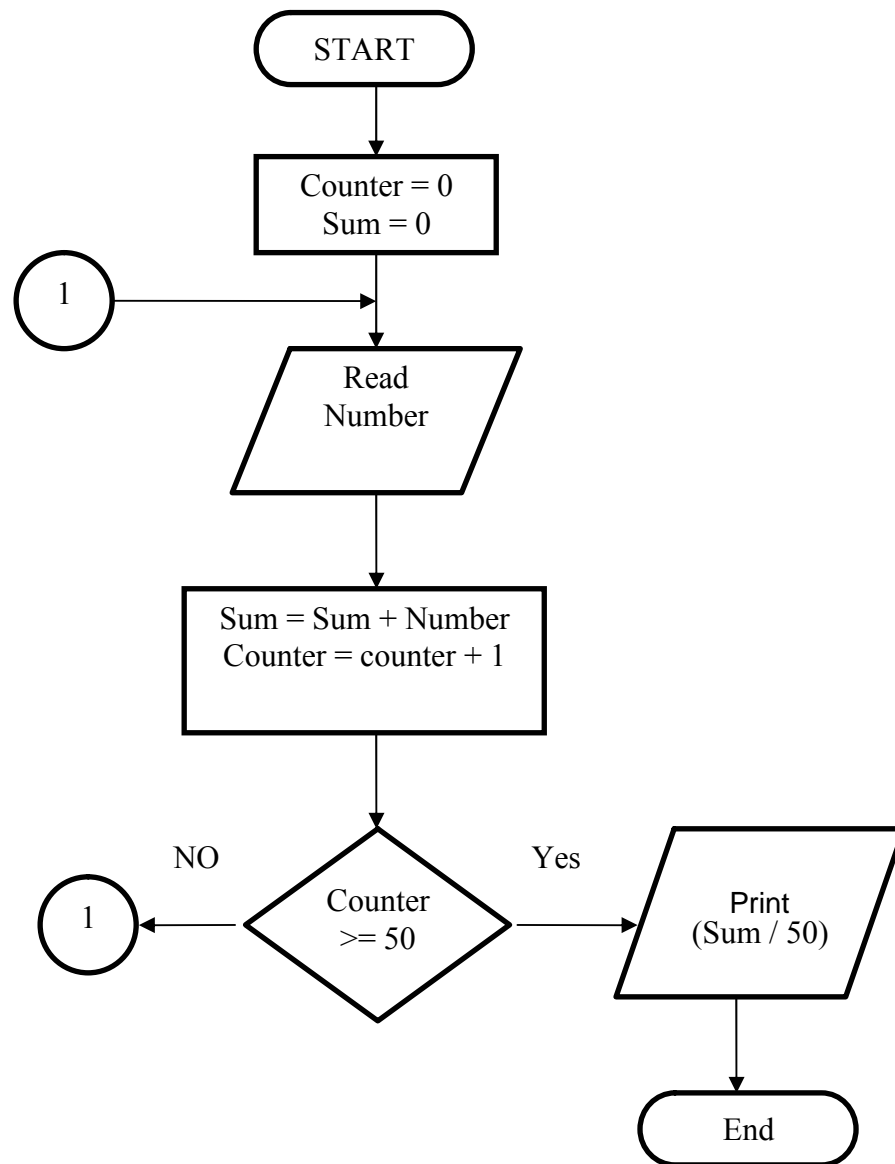
Example (8)

Draw the flowchart to find the sum of even numbers from 1.....100 ?



Example (9)

Draw the flowchart to read 50 number then find the average ?



Lecture 2

Basic Internet



International Network (Internet)

Internet Basics

Getting to the Internet: Browsers



A browser is a software application used to locate and display web pages. The two most popular browsers are Microsoft Internet Explorer and Firefox. Both of these are *graphical browsers*, which means that they can display graphics as well as text. In addition, most modern browsers can present multimedia information, including sound and video, though they require plug-ins for some formats.

The Internet: URLs

Every document on the Web has an address which is called the URL (Uniform Resource Locator). Each URL has several parts: the protocol, the host name and the domain name

The URL to locate GOOGLE is <http://www.google.com>

http:// the first part identifies the document as a Web page. Other parts of the Web have different identifiers such as FTP (File Transfer Protocol) and IP (Internet Protocol)

www document on the World Wide Web. Not all Web sites have this as part of their address.

.google host name

.com domain name that identifies category of the page

. Other typical domain names include:

- **gov** - Government agencies
- **edu** - Educational institutions
- **org** - Organizations
- **mil** - Military
- **com** - commercial business
- **net** - Network Organizations;

Country Codes

The original top-domain categories were adequate for their original purpose but soon became inadequate when the Internet became international. The top-level domains were expanded to include two letter country codes. The following are examples:

.iq Iraq
.ca Canada
.de Germany
.uk United Kingdom

A URL may have additional components that identify special features of the page such as htm l (hypertext markup language)

Opening Microsoft Internet Explorer



→ Programs → Internet Explorer

Or double-click on the shortcut icon on the desktop or pinned to the taskbar.



Internet Explorer Toolbar



File menu: contains selections such as page setup, print preview, print and properties.

Edit menu: contains selections such as copy, paste, and select all.

View menu: contains selections such as changing the toolbars available, changing size of text on screen, refreshing the current page.

Tools menu: contains popup blocker, phishing filter, internet options, etc.

Help menu: contains selections for seeking help with the program (Notice that for many actions there is a keyboard shortcut using the Control key and a letter key. For example, open file Ctrl-O and close file Ctrl-W)

Icon Shortcuts

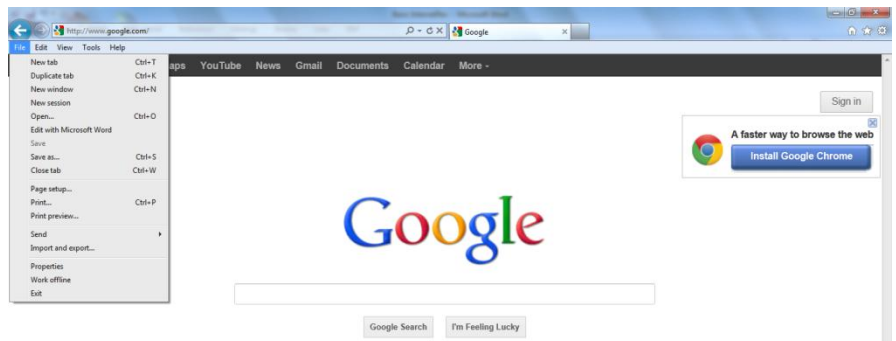


Home: returns you to the page that you see when you first open your browser.

★ **Favorites (or Bookmarks) menu:** creates a type of “shortcut” to a favorite website.

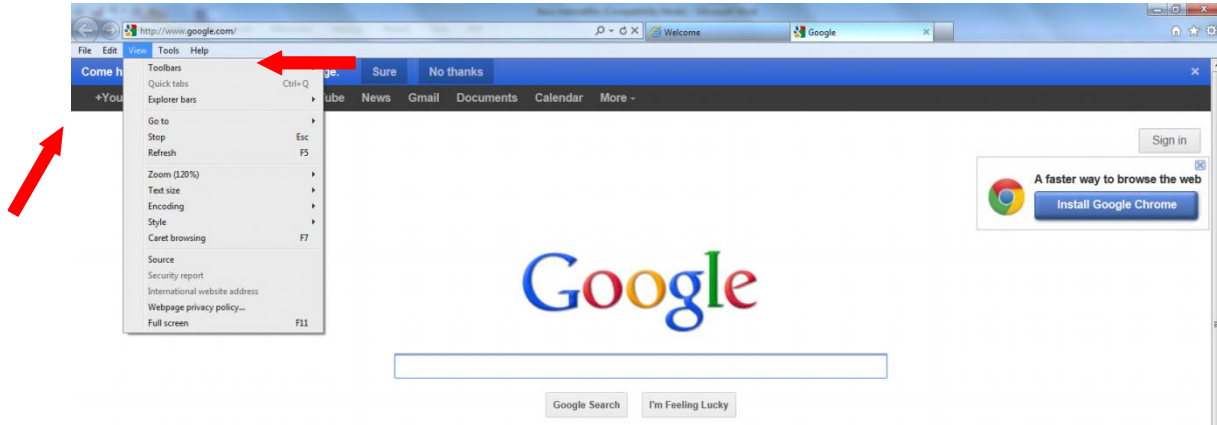


Tool: includes internet options, print, safety, and About Internet Explorer

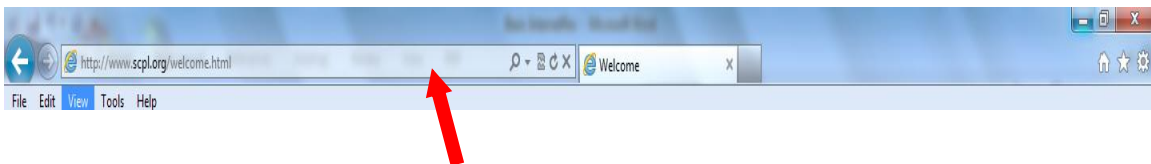


Getting to Know the Toolbars

From the View menu: you can select or deselect different toolbars to appear on your browser. Various search engine toolbars can also be downloaded.




The address bar contains the website's URL. The symbols at the end of the address bar allow you to Search, Select address field, Refresh, or Stop loading a page.



Visiting a Web Page


After you have opened your browser (Internet Explorer, Firefox, etc.), it will load (bring up) the Home Page which may be set as Google, another search engine, or any webpage of your choice. Click in the address bar to activate the I-bar which tells the computer you are about to enter a URL (web address).

Navigating a Page from the Web


Pages will generally contain “links,” which will take you to a different portion of the site, a file to view, or a different site all together. Links will generally be noted by blue text or underlined, or will change color if the cursor is placed over it. The arrow cursor will also change into a . You are, in effect, saying to the computer, “Take me here.”

If a page is larger than the screen, scroll bars appear – usually on the right and/or bottom.



If you want to return to a previously viewed site, click the back OR forward button OR the down arrow at the right end of the address bar. Another option is to click View  History on the main toolbar to view at least ten previously viewed sites. The Star icon on the right side of your screen will also bring up your “view favorites, feeds, and history.”

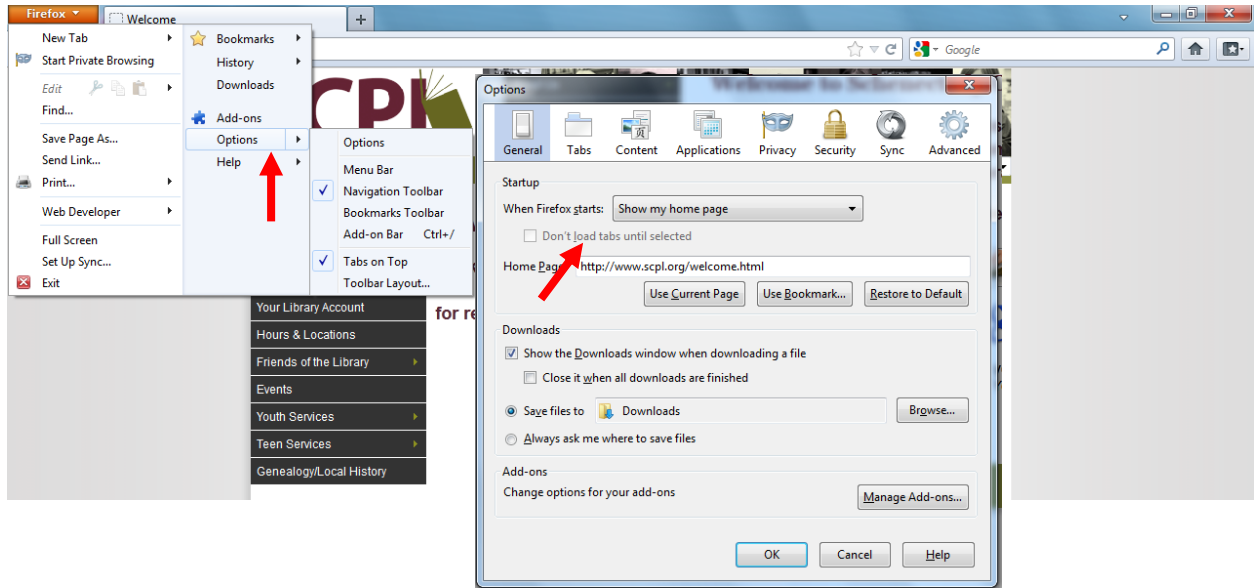
Favorites

In your browsing of the web, you will undoubtedly come across pages that you will wish to save for later viewing. In web browsing, you do not save web pages; you add the site’s URL to your Favorites area. To mark a page as a Favorite, browse to the page you wish to bookmark. Click the Favorites  icon and click Add to Favorites. You can create folders and save similar URL sites in the appropriate folder.

Home Page: Be creative! There is more to life than your Search Engine.

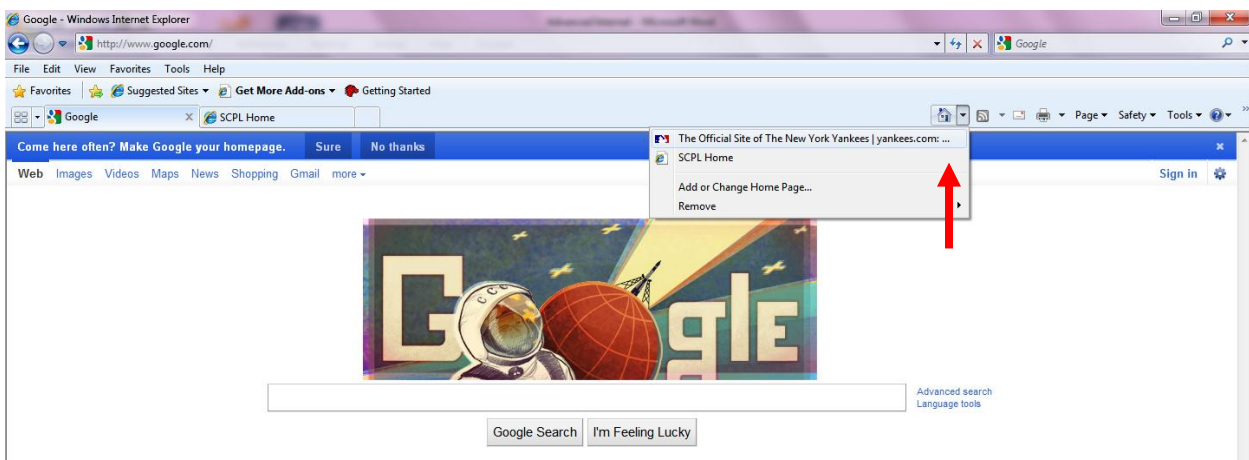
Do you have a favorite sports team page, cooking show page, or perhaps news network? Make that page your homepage with just a few simple clicks of your mouse!

Open your browser and **click** on the Tools > Options feature.



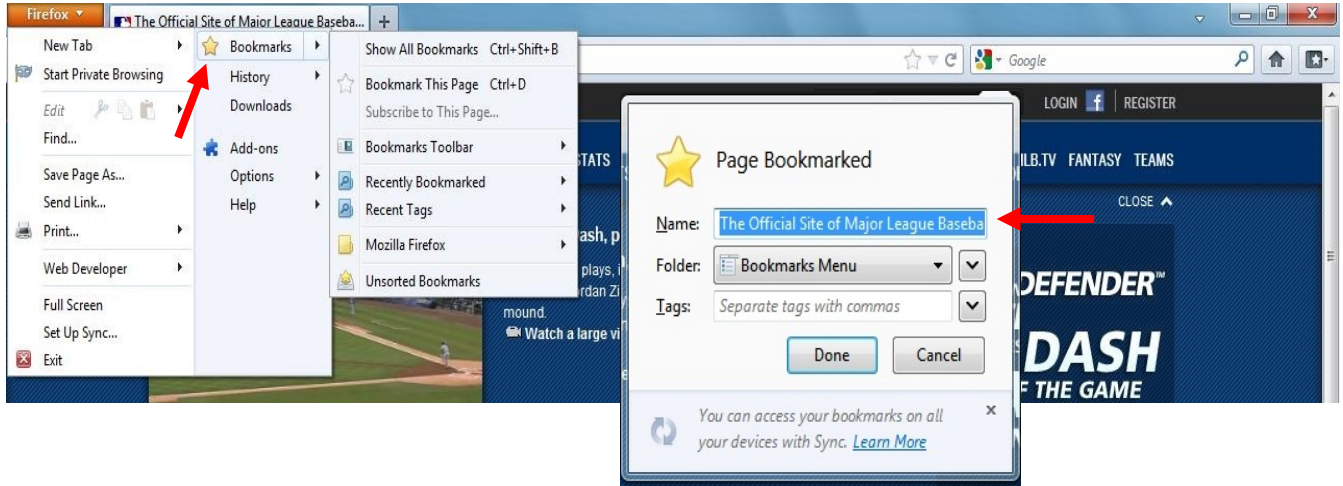
In Options > General, enter web address that you would like for your home page. Click OK.

In the event you have the homepage option through the Home icon, you can complete the same tasks using that dropdown menu.



Favorites Make Life Easier!

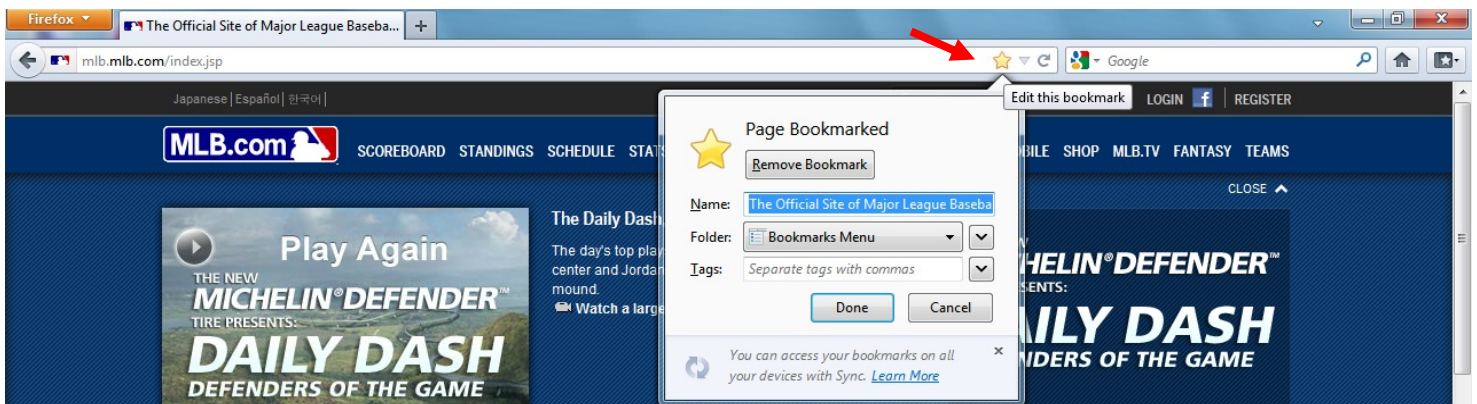
Your browser toolbar (or shortcut feature) has a tab called Favorites (or Bookmarks if you are using Firefox). You can save your favorite links in Bookmarks and organize them into folders that reflect your personal needs. For example, you are a baseball enthusiast. You visit the major league baseball site frequently.



Click *Bookmarks* > *Add to Bookmarks*.

You can either **click** *Add* **OR** **click** *New Folder* where you will continue to add websites that reflect your interest in baseball.

Notice the yellow Star. You can select the *Bookmark Toolbar* as an added convenience feature.



Lecture 3

Searching the Internet

Finding Information

There are basically three ways to locate information on the World Wide Web:

1. Enter the URL
2. Links on a Web page
3. Search engine

Search Engines

Search engines are web sites that allow you to type in a few key words and then present you with a list of possible links that might have the information you want. Although *search engine* is really a general class of programs, the term is often used to specifically describe systems like Google, Alta Vista and Excite that enable users to search for documents on the World Wide Web.

Google <http://www.google.com>

Yahoo <http://www.yahoo.com>

Bing <http://www.bing.com>

AltaVista <http://www.altavista.com/>

Basic Searching Methods and Language:-

Searching the Internet can bring the information from around the world into your home/office or it can be an incredibly frustrating and time consuming disaster. Here are some strategies that will increase the likelihood of finding relevant information.

Analyze your topic – what are you looking for? Searching for very broad subjects, such as history of the United States, will produce a huge number of “hits” and be very confusing. What do we really want to know – a particular period in U.S. history, a list of democratic presidents, inflation rates during the 20th century

Select keywords for your topic - The keywords you choose may or may not bring you the results you want. Be prepared to use similar words for your topic

Keyword Phrase searching - Some search engines require the use of “ “, +, -. Quotation marks are used around words you want searched as a phrase. The + in front of a word (with no space) tells the search engine the word **MUST** appear in the results. The – in front of a word (with no space) tells the search engine that the word **MUST NOT** be included in the results. Since Google automatically returns pages that include all keywords, the plus sign (+) and the operator AND are not needed.

Case sensitivity - Most search engines ignore case. Generally, enter your search terms using lower case. It is quicker and will give you more results. However, if you are looking for a specific person, place, title, capitalize the first letter of each word.

Internet E-mail Account

To Set Up a New Account

1. Open your browser. If Google is not your default home page, enter www.google.com in the address bar. Click on **Gmail**.

2. Click on **Create an account**.

3. In order to create an account, you will have to answer a series of questions, such as **Desired Login Name** and **Choose a password**.

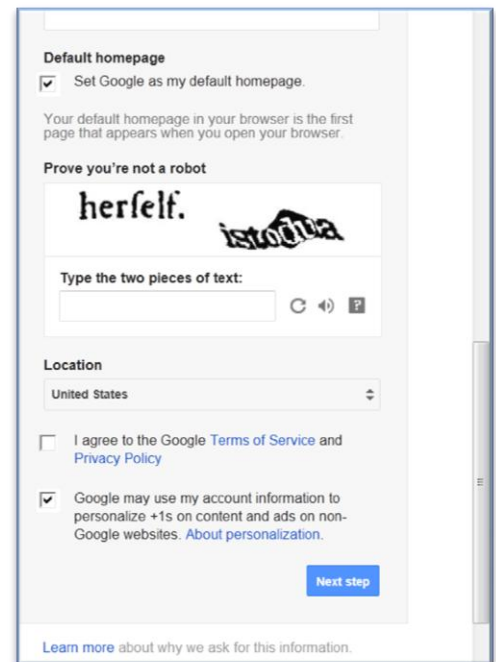
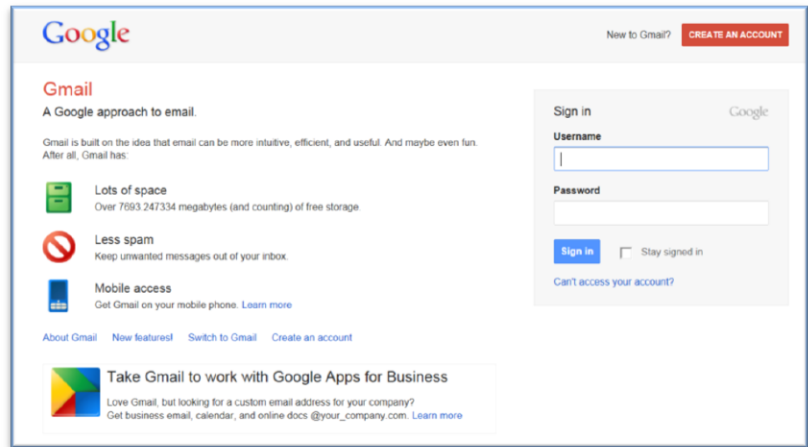
4. You may **Stay logged in** even if you deselect that feature. Be sure to **Sign out** if you share your computer with others.

5. If you do not want Google as your default homepage, be sure to deselect that option.

6. “Prove you’re not a robot” can be troublesome. Use your zoom feature to see the two pieces of text you need to enter. Return zoom to original setting – usually 100% before clicking Next Step .

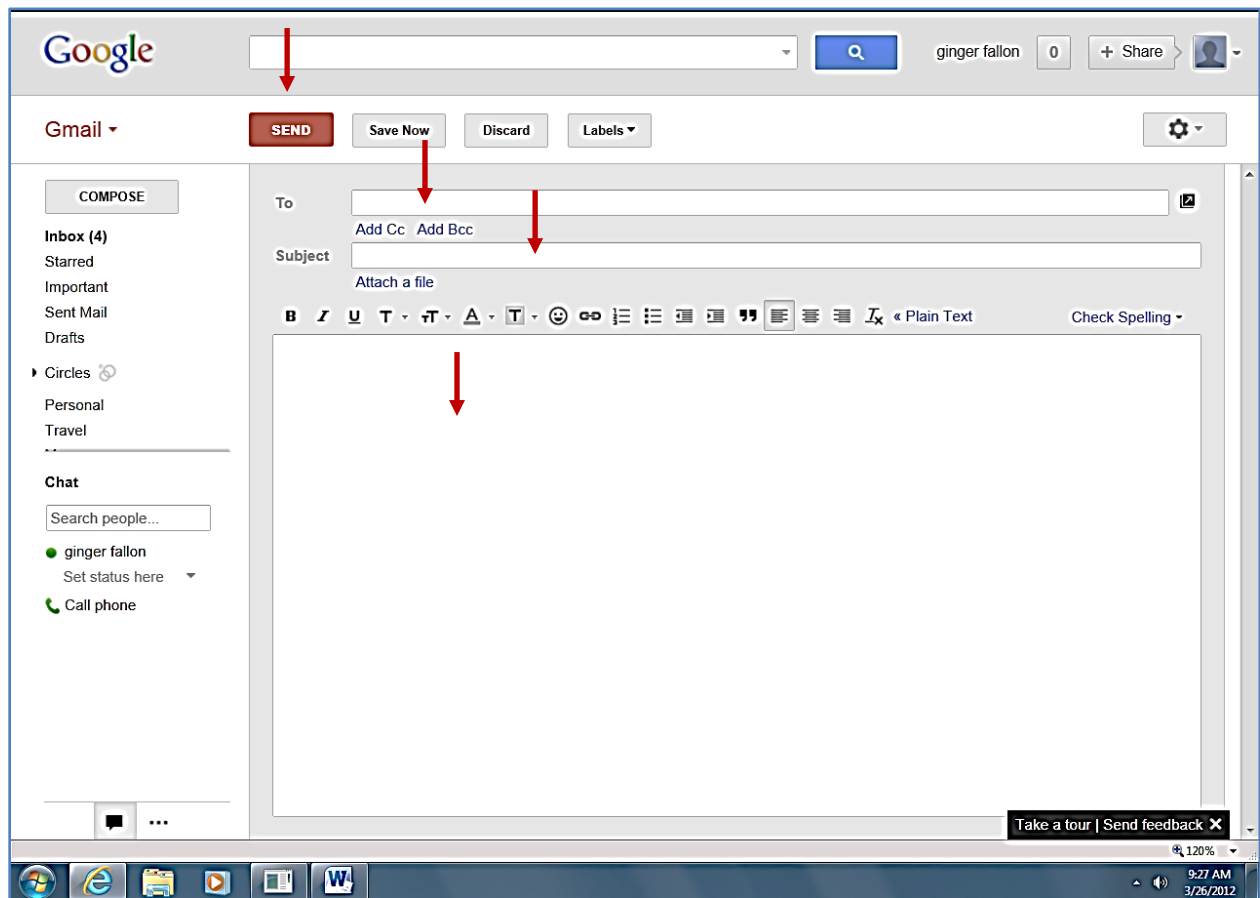
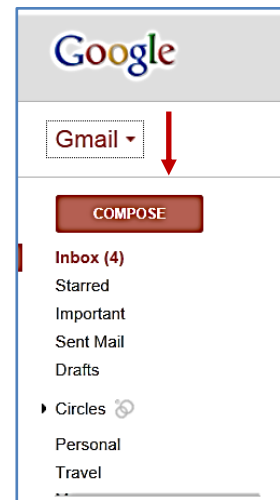
7. After you have completed answering all the questions and reviewed Google policies, terms and conditions, click **Next Step** where you have the opportunity to create a profile (not required) then **Continue to Gmail**. (At some point you may be asked to enter a recovery e-mail address. If you do not have a secondary e-mail address, click **Save and Continue**.)

*Click **Help** for information on how to change password and other questions about your account.



To Send an E-mail

1. Click on *Compose Mail*.
2. Enter e-mail address of recipient and subject of e-mail. E-mail etiquette recommends a subject. Many people will delete e-mails without opening them if there is no subject listed.
3. Click in message box and enter your message. You can change your font by using the format font toolbar. Click *Check Spelling*.
4. If you want to add either a Cc address or blind CC address, click on the *Add Cc* and/or *Add Bcc links*.
5. Click *Send*.

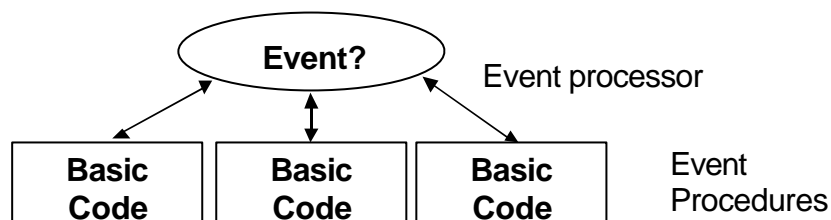


PROGRAMMING
WITH
VISUAL BASIC 6

Chapter 1

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

1.1: Getting Started

This section helps you get Visual Basic loaded and running, and shows you how to control the development environment elements.

1.1.1: Starting Visual Basic

The first step in using Visual Basic is launching it and opening existing files or creating new ones.

► **to start Visual Basic 6.0**

1. In Windows, click **Start**, point to Programs, and point to the Microsoft Visual Basic 6.0 folder.

The icons in the folder appear in a list.

2. Click the **Microsoft Visual Basic 6.0** program icon.

The **New Project** dialog box appears. This dialog box prompts you for the type of programming project you want to create.

See the picture of **New Project** window below.

3. To accept the default new project, click **OK**.

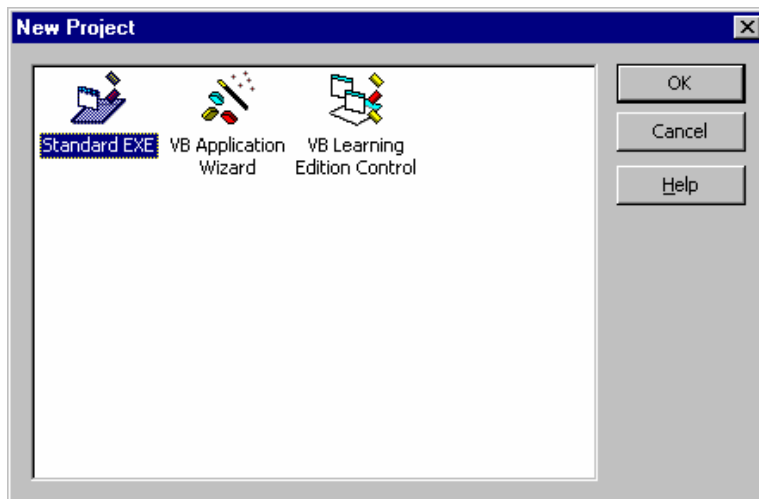


Figure 1 : New Project Window

In the Visual Basic development environment, a new project (a standard, 32-bit Visual Basic application) and the related windows and tools open. See the picture below.

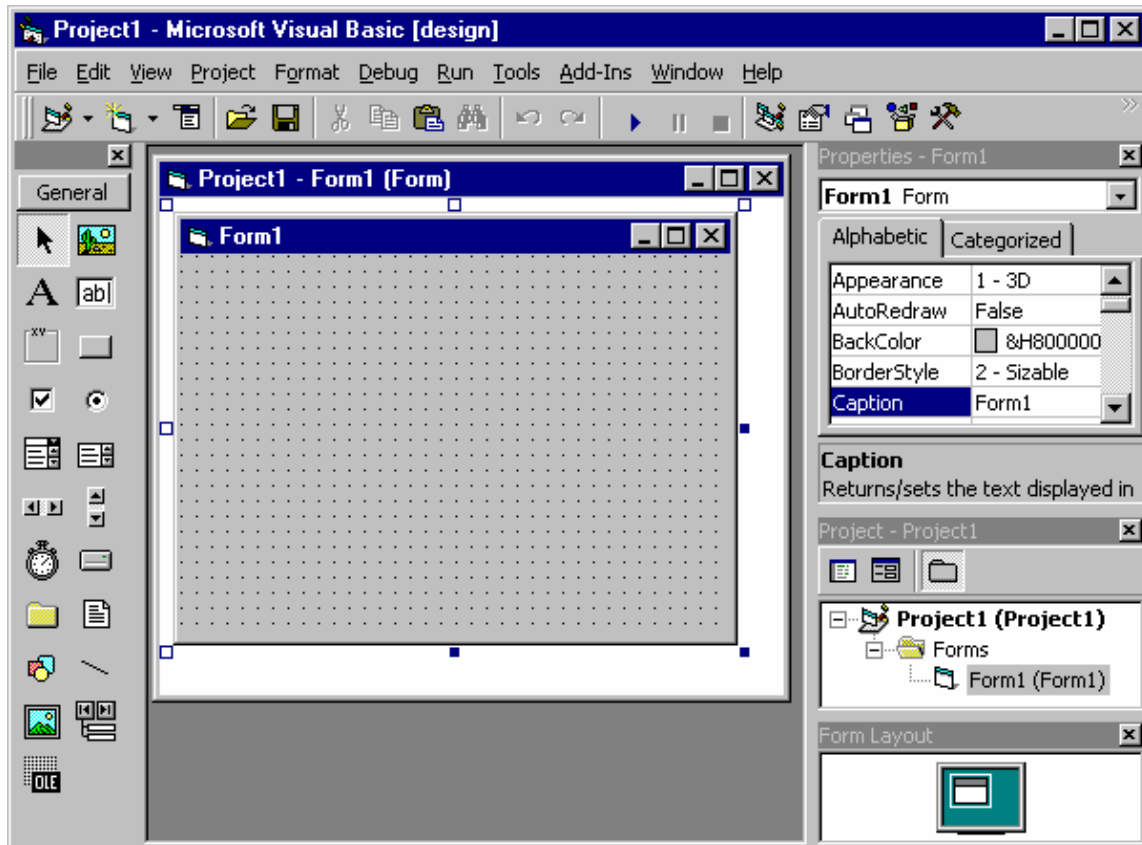


Figure 2 Visual Basic Environment

The Visual Basic development environment contains these programming tools and windows, with which you construct your Visual Basic programs:

- ◆ Menu bar
- ◆ Toolbars
- ◆ Visual Basic toolbox
- ◆ Form window
- ◆ Properties window
- ◆ Project Explorer
- ◆ Immediate window
- ◆ Form Layout window

The exact size and shape of the windows depends on how your system has been configured. In Visual Basic 6.0, you can align and attach (dock) the windows to make all the elements of the programming system visible and accessible. You'll learn how to customize your development environment in *Moving, Docking, and Resizing Windows*.

1.1.2: Loading and Running a Program

Before you can work with a Visual Basic program, you need to load the program into memory, just as you would load a word processing document in a word processor for editing.

► To load a Visual Basic program into memory and run it

1. On the **File** menu, click **Open Project**.

The **Open Project** dialog box appears. With this dialog box, you can open any existing Visual Basic program on your hard disk, attached network drive, CD-ROM, or floppy disk.

2. If necessary, use the **Look In** drop-down list box and the **Up One Level** button to locate the program you want to load. Then, double-click the program name.

The project file loads the Visual Basic user interface form, properties, and program code. (**Visual Basic project files are distinguished by the .VBP file name extension.**)

3. If the program user interface does not appear, open the Forms folder in the Project window, select the first form, and then click **View Object** in the Project window.

This is an optional but useful step, especially if you want to look at the program user interface in the Form window before you run it.

4. On the Visual Basic Standard toolbar, click **Start** to run the program.

The toolbox and several of the other windows disappear, and the Visual Basic program starts to run.

5. On the toolbar, click **End** when you want to exit the program.

1.2: Visual Basic Resources

1.2.1: Programming Tools

The location and purpose of the Visual Basic 6.0 programming tools are described in the following.

Menu bar Located at the top of the screen, the menu bar provides access to the commands that control the Visual Basic programming environment. Menus and commands work according to standard conventions used in all Windows-based programs. You can use these menus and commands by using keyboard commands or the mouse.

Toolbars Located below the menu bar, toolbars are collections of buttons that serve as shortcuts for executing commands and controlling the Visual Basic development environment. You can open special-purpose toolbars by using the **View** menu **Toolbars** command.

Windows taskbar This taskbar is located along the bottom of the screen. You can use the taskbar to switch between Visual Basic forms as your program runs and to activate other Windows-based programs.

1.2.2: Toolbox Controls

You use special tools, called controls, to add elements of a program user interface to a form. You can find these resources in the toolbox, which is typically located along the left side of the screen. (If the toolbox is not open, display it by using the **Toolbox** command on the **View** menu.) By using toolbox controls, you can add these elements to the user interface:

- ◆ Artwork
- ◆ Labels and text boxes
- ◆ Buttons
- ◆ List boxes
- ◆ Scroll bars
- ◆ File system controls
- ◆ Timers
- ◆ Geometric shapes
- ◆ Data and **OLE** controls.

See below to view an illustration of the standard contents of the toolbox.

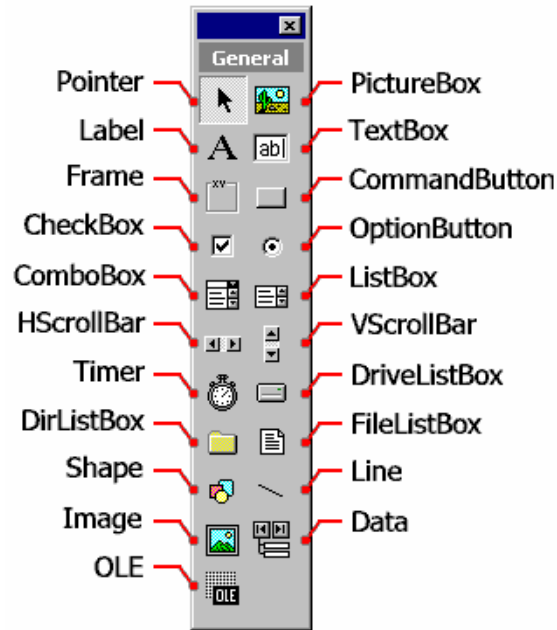


Figure3: Toolbox Controls

Visible and Invisible Controls

When a Visual Basic program runs, most toolbox controls operate like the standard objects in any Windows-based application — and they will be visible to the user. However, the toolbox also contains controls that can be used to perform special, behind-the-scenes operations in a Visual Basic program. The powerful objects you create with these controls do useful work but can be made invisible to the user when the program runs. These objects can be used for:

- ◆ Manipulating database information.
- ◆ Working with Windows-based applications.
- ◆ tracking the passage of time in your programs.

1.2.3: Form Window

When you start Visual Basic, a default form (Form1) with a standard grid (a window consisting of regularly spaced dots) appears in a pane called the Form window. You can use the Form window grid to **create the user interface** and to line up interface elements.

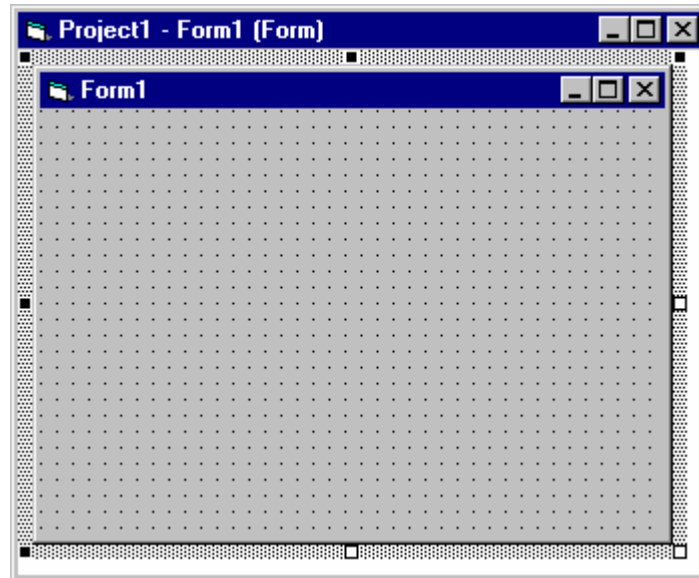


Figure 4: Form Window

Building Interface Elements

To build the interface elements, you click an interface control in the Visual Basic toolbox, and then you draw the user interface element on your form by using the mouse. This process is usually a simple matter of clicking to position one corner of the element and then dragging to create a rectangle the size you want. After you create the element — say, a text box — you can refine it by setting properties for the element. In a text box, for example, you can set properties to make the text boldface, italic, or underlined.

Adjusting Form Size

You can adjust the size of the form by using the mouse — the form can take up part or the entire screen.

Controlling Form Placement

To control the placement of the form when you run the program, adjust the placement of the form in the **Form Layout** window.

1.2.4: Properties Window

With the Properties window, you change the characteristics (property settings) of the user interface elements on a form. A property setting is a characteristic of a user interface object. For example, you can change the text displayed by a text box control to a different font, point size, or alignment. (With Visual Basic, you can display text in any font installed on your system, just as you can in Microsoft Excel or Microsoft Word.)

Displaying the Properties Window

To display the Properties window, click the **Properties Window** button on the toolbar. If the window is currently docked, you can enlarge it by double-clicking the title bar. To redock the Properties window, double-click its title bar again.

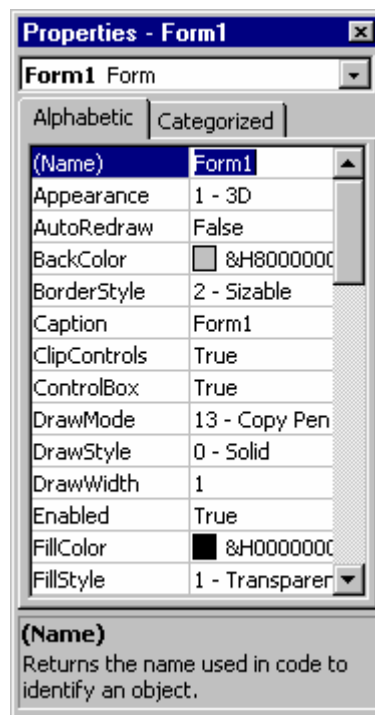


Figure 5: Properties Window

Properties Window Elements

The Properties window contains the following elements:

- ◆ A drop-down list box at the top of the window, from which you select the object whose properties you want to view or set.
- ◆ Two tabs, which list the properties either alphabetically or by category.
- ◆ A description pane that shows the name of the selected property and a short description of it.

Changing Property Settings

You can change property settings by using the Properties window while you design the user interface or by using program code to make changes while the program runs.

1.2.5: Project Window

A Visual Basic program consists of several files that are linked together to make the program run. The Visual Basic 6.0 development environment includes a Project window to help you switch back and forth between these components as you work on a project.

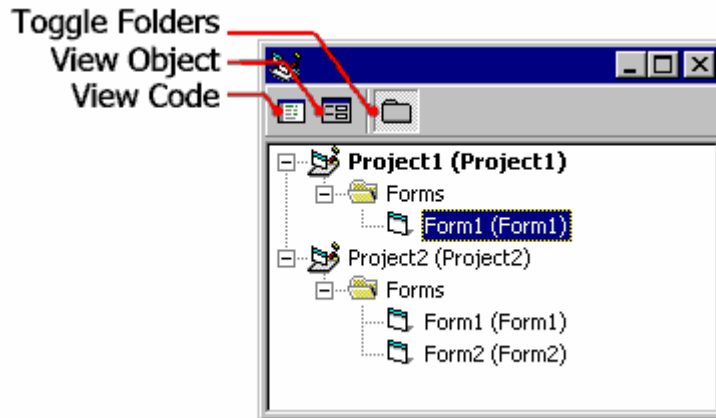


Figure 6: Project Explorer Window

Project Window Components

The Project window lists all the files used in the programming process and provides access to them with two special buttons: **View Code** and **View Object**.

Displaying the Project Window

To display the Project window, click the **Project Explorer** button on the Visual Basic toolbar. If the window is currently docked, you can enlarge it by double-clicking the title bar. To re-dock the Project window, double-click its title bar again.

Adding and Removing Files

The project file maintains a list of all the supporting files in a Visual Basic programming project. You can recognize project files by their **.vbp** file name extension.

You can add individual files to and remove them from a project by using commands on the Project menu. The changes that you make will be reflected in the Project window.

Note In Visual Basic versions 1 through 3, project files had the **.mak** file name extension. In Visual Basic versions 4, 5, and 6.0, project files have the **.vbp** file name extension.

Adding Projects

If you load additional projects into Visual Basic with the **File** menu **Add Project** command, outlining symbols appear in the Project window to help you organize and switch between projects.

1.2.6: Code Window

You can create much of your program by using controls and setting properties. However, most Visual Basic programs require additional program code that acts as the brains behind the user interface that you create. This computing logic is created using program statements — keywords, identifiers, and arguments — that clearly spell out what the program should do each step of the way.

You enter program statements in the Code window, a special text editing window designed specifically for Visual Basic program code. You can display the Code window in either of two ways:

- ◆ By clicking **View Code** in the Project window.
- ◆ By clicking the **View** menu **Code** command.

1.2.7: Form Layout Window

The Form Layout window is a visual design tool. With it, you can control the placement of the forms in the Windows environment when they are executed. When you have more than one form in your program, the Form Layout window is especially useful — you can arrange the forms onscreen exactly the way you want.

To position a form in the Form Layout window, simply drag the miniature form to the desired location in the window.

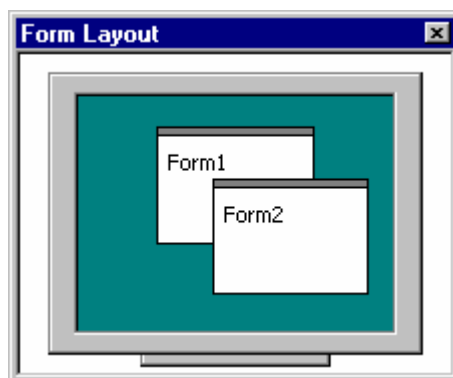


Figure 7: Form Layout Window

1.3: Developing Visual Basic Programs

1.3.1: Developing Visual Basic Programs

If you haven't written a program before, you might wonder just what a program is and how to create one in Visual Basic. This section provides an overview of the Visual Basic programming process.

A program is a set of instructions that collectively cause a computer to perform a useful task, such as processing electronic artwork or managing files on a network. A program can be quite small — something designed to calculate a home mortgage — or it can be a large application, such as Microsoft Excel.

A Visual Basic program is a Windows-based application that you create in the Visual Basic development environment. This section includes the following topics:

- ◆ Planning the Program
- ◆ Building the Program
- ◆ Testing, Compiling, and Distributing the Program

1.3.1.1: Planning the Program

The first step in programming is determining exactly what you want your program to accomplish. This sounds simple (isn't it obvious?), but without a mission statement, even the best programmer can have trouble building an application he or she is happy with.

Planning a program is a little like planning a barbecue. For a barbecue to go off smoothly, you need to prepare for it ahead of time. You need to organize the menu, invite your friends, buy the food, and (most likely) clean your house. But a barbecue can be entertaining if friends just happen to drop by and bring stuff. Programs, though, usually don't turn out the best if they're built with the stone-soup approach.

Identify your Objectives

Long before you sit down in front of your computer, you should spend some time thinking about the programming problem you are trying to solve. Up-front planning will save you development time down the road, and you'll probably be much happier with the result. One part of the planning process might be creating an ordered list of programming steps, called an **algorithm**.

Ask Yourself Questions

When you plan Visual Basic programming projects, you might find it useful to ask yourself the following questions about your program:

- ◆ **What** is the **goal (mission)** of the program I am writing?
- ◆ **Who** will use the program?
- ◆ **What** will the program **look like** when it starts?
- ◆ **What information** will the user enter in the program?
- ◆ **How** will the program process the input?
- ◆ **What information (output)** will the program produce?

When you finish this preliminary work, you'll be ready to start building the program with Visual Basic.

1.3.1.2: Building the Program

Building a Windows-based application with Visual Basic involves three programming steps: creating the user interface, setting the properties, and writing the code. And, of course, your project must be saved.

These steps are described in the following topics:

- ◆ Creating the User Interface
- ◆ Setting the Properties
- ◆ Writing Program Code
- ◆ Saving a Project

Creating the User Interface

After you have established a clear goal for your program, it's important to think about how it will look and how it will process information. The complete set of forms and controls used in a program is called the program user interface. The user interface includes all the menus, dialog boxes, buttons, objects, and pictures that users see when they operate the program. In the Visual Basic development environment, you can create all the components of a Windows-based application quickly and efficiently.

Setting the Properties

Properties are programmable characteristics associated with forms and their controls. You can set these properties either as you design your program (at design time) or while you run it (at run time). You change properties at design time by selecting an object, clicking the Properties window, and changing one or more of the property settings. To set properties at run time, you use Visual Basic program code.

Writing the Program Code

You finish building your program by typing program code for one or more user interface elements. Writing program code gives you more control over how your program works than you can get by just setting properties of user interface elements at design time. By using program code, you completely express your thoughts about how your application:

- ◆ Processes data
- ◆ Tests for conditions
- ◆ Changes the order in which Visual Basic carries out instructions.

The Visual Basic Programming Language

The Visual Basic programming language contains several hundred statements, functions, and special characters. However, most of your programming tasks will be handled by a few dozen, easy-to-remember keywords.

In this course, you'll spend a lot of time exploring the subtleties of writing useful program code that you can adapt to a variety of situations. For now, though, just keep these points in mind:

- ◆ Program code follows a particular form (syntax) required by the Visual Basic compiler.
- ◆ You enter and edit program code in the Code window, a special text editor designed to track and correct (debug) program statement errors.

Saving a Project

After you complete a program or find a good stopping point, you should save the project to disk with the **Save Project As** command on the **File** menu.

Chapter 2

Visual Basic Fundamentals

2.1 NUMERIC CONSTANTS

Numbers are referred to as *numeric constants* in Visual Basic. Most numeric constants are expressed as *integers* (whole numbers that do not contain a decimal point or an exponent), *long integers* (similar to integers with an extended range), *single-precision real quantities* (numbers that include a decimal point, an exponent, or both), or *double-precision real quantities* (similar to single-precision real quantities with an extended range and greater precision). The following rules apply to numeric constants:

1. Commas cannot appear anywhere in a numeric constant.
2. A numeric constant may be preceded by a + or a – sign. The constant is understood to be positive if a sign does not appear.
3. An integer constant occupies two bytes. It must fall within the range –32,768 to 32,767. It cannot contain either a decimal point or an exponent.
4. A long integer constant occupies four bytes. It must fall within the range –2,147,483,648 to 2,147,483,647. It cannot contain either a decimal point or an exponent.
5. A single-precision real constant occupies four bytes. It can include a decimal point and as many as seven significant figures. However, its magnitude cannot exceed approximately 3.4×10^{38} .

A single-precision real constant can include an exponent if desired. Exponential notation is similar to scientific notation, except that the base 10 is replaced by the letter E. Thus, the quantity 1.2×10^{-3} could be written as 1.2E–3. The exponent itself can be either positive or negative, but it must be a whole number; i.e., it cannot contain a decimal point.

6. A double-precision real constant occupies eight bytes. It can include a decimal point and as many as fifteen significant figures. However, its magnitude cannot exceed approximately 1.8×10^{308} .

A double-precision real constant can include an exponent if desired. Double-precision exponential notation is similar to scientific notation, except that the base 10 is replaced by the letter D. Thus, the quantity 1.6667×10^{-3} could be written as 1.6667D–3. The exponent itself can be either positive or negative, but it must be a whole number; i.e., it cannot contain a decimal point.

EXAMPLE 2.1

Several Visual Basic numeric constants are shown below. Note that each quantity (each row) can be written in several different ways.

0	+0	-0	
1	+1	0.1E+1	10E-1
-5280	-5.28E+3	-.528E4	-52.8E2
1492	0.1492D+4	1.492D+3	+14.92D2
-.0000613	-6.13E-5	-613E-7	-0.613E-4
3000000	3D6	3D+6	0.3D7

2.2 STRING CONSTANTS

A *string constant* is a sequence of characters (i.e., letters, numbers and certain special characters, such as +, -, /, *, =, \$, ., etc.), enclosed in quotation marks. Blank spaces can be included within a string. A quotation mark can also be placed within a string, but it must be written as *two adjacent* quotation marks (see the last line in the example below).

String constants are used to represent nonnumeric information, such as names, addresses, etc. There is no practical restriction on the maximum number of characters that can be included within a string constant. Thus, the maximum length of a string constant can be considered infinite.

EXAMPLE 2.2

Several string constants are shown below.

"SANTA CLAUS"	"Please type a value for C:"
"\$19.95"	"Welcome to the 21st Century"
"X1 = "	"3730425"
"The answer is "	"Do you wish to try again?"

2.3 VARIABLES

A *variable* is a name that represents a numerical quantity, a string, or some other basic data item (e.g., a date, true/false condition, etc.). The following rules apply to the naming of variables:

1. A variable name must begin with a letter. Additional characters may be letters or digits. Certain other characters may also be included, though the period and special *data-typing characters* (e.g., %, &, !, #, and \$) are *not* permitted. In general, it is good programming practice to avoid the use of characters other than letters and digits.
2. A variable name cannot exceed 255 characters. As a practical matter, however, variable names rarely approach this size.
3. Visual Basic does not distinguish between uppercase and lowercase letters. Many programmers use uppercase letters as word separators within a single variable name (e.g., FreezingPoint, TaxRate, etc.)
4. Visual Basic includes a number of *reserved words* (e.g., Dim, If, Else, Select, Case, Do, etc.). These reserved words represent commands, function names, etc. They *cannot* be used as variable names.

EXAMPLE 2.3

Several variable names are shown below.

Area	Radius	X	xmax	C3
Counter	CustomerName		Account_Number	UnpaidBalance

2.4 DATA TYPES AND DATA DECLARATIONS

Visual Basic supports all common data types, including *Integer*, *Long* (i.e., long integer), *Single*, *Double* and *String*. The language also supports other data types, such as *Boolean*, *Byte*, *Currency* and *Date* data, as well as *Variant*-type data (see below) and user-defined data types.

The `Dim` statement is used to associate variables with specific data types. This process, which is common to all modern programming languages, is known as *data declaration*, or simply *declaration*. In general terms, the `Dim` statement is written as

```
Dim variable name 1 As data type 1, variable name 2 As data type 2, etc.
```

EXAMPLE 2.4

Several variable declarations are shown below.

```
Dim Counter As Integer
Dim Area As Single
Dim StudentName As String
Dim StudentName As String * 30
Dim TaxRate As Single, Income As Double, Taxes As Double, Dependents As Integer
```

Variants

Visual Basic allows variables to be undeclared if the programmer so chooses. In such cases, the data type of the variable is determined implicitly by the value that is assigned to the variable. Such variables are referred to as *Variant-type* variables, or simply as *variants*.

On the surface, the use of variants appears to simplify the program development process. This is a false perception, however, as the use of variants is computationally inefficient, and it compromises the clarity of a program. *Good programming practice suggests that the use of variants be avoided.* Use explicitly declared variables instead.

Named Constants

It is also possible to define named constants in Visual Basic. Named constants are similar to variables. However, variables can be reassigned different values within a program, whereas named constants remain unchanged throughout a program.

The `Const` statement is used to declare a named constant. This statement has the general form

```
Const constant name As data type = value
```

EXAMPLE 2.5

Here are some typical named constant declarations:

```
Const TaxRate As Single = 0.28
Const Avogadro As Double = 6.0225D+23
Const MaxCount As Integer = 100
```

The first line declares `TaxRate` to be a single-precision real constant whose value is 0.28. The second line defines *Avogadro's number* as a double-precision real constant whose value is 6.0225×10^{23} . The last line declares `MaxCount` as an integer constant whose value is 100.

Note that the values assigned to `TaxRate`, `Avogadro` and `MaxCount` will remain unchanged throughout the program.

Suffixes

Rather than declaring a data type explicitly (using a `Dim` or `Const` statement), a variable or named constant can be associated with a data type by adding a single-character *suffix* to the end of the variable/constant name. Several of the more commonly used suffixes are listed below.

<u>Suffix</u>	<u>Data Type</u>
%	integer
&	long integer
!	single
#	double
\$	string

EXAMPLE 2.6

Shown below are several variables whose data types are defined by suffixes.

<u>Variable</u>	<u>Data Type</u>
Index%	integer
Counter&	long integer
TaxRate!	single
Ratio#	double
CustomerName\$	string

2.5 OPERATORS AND EXPRESSIONS

Special symbols, called *arithmetic operators*, are used to indicate arithmetic operations such as addition, subtraction, multiplication, division and exponentiation. These operators are used to connect numeric constants and numeric variables, thus forming *arithmetic expressions*.

The standard arithmetic operators are

<i>Addition:</i>	+
<i>Subtraction:</i>	-
<i>Multiplication:</i>	*
<i>Division:</i>	/
<i>Exponentiation:</i>	^

When arithmetic operators appear within an arithmetic expression, the indicated operations are carried out on the individual terms within the expression, resulting in a single numerical value. Thus, *an arithmetic expression represents a specific numerical quantity*.

EXAMPLE 2.7

Several arithmetic expressions are presented below.

$2 * j + k - 1$	$2 * (j + k - 1)$
$first + second - third$	$(a ^ 2 + b ^ 2) ^ 0.5$
$4 * Pi * Radius ^ 3 / 3$	$(5 / 9) * (F - 32)$
$b ^ 2 - (4 * a * c)$	$(2 * x - 3 * y) / (u + v)$

Each expression represents a numerical quantity. Thus, if the variables a , b and c represent the quantities 2, 5 and 3, respectively, the expression $a + b - c$ will represent the quantity 4.

Visual Basic also includes two additional arithmetic operators:

<i>Integer division</i>	\	(backward slash)
<i>Integer remainder</i>	Mod	

In integer division, each of the two given numbers is first *rounded* to an integer; the division is then carried out on the rounded values and the resulting quotient is truncated to an integer. The integer remainder operation (Mod) provides the remainder resulting from an integer division.

EXAMPLE 2.8

The results of several ordinary division, integer division and integer remainder operations are shown below.

$13/5 = 2.6$	$13 \setminus 5 = 2$	$13 \text{ Mod } 5 = 3$
$8.6/2.7 = 3.185185$	$8.6 \setminus 2.7 = 3$	$8.6 \text{ Mod } 2.7 = 0$
$8.3/2.7 = 3.074074$	$8.3 \setminus 2.7 = 2$	$8.3 \text{ Mod } 2.7 = 2$
$8.3/2.2 = 3.772727$	$8.3 \setminus 2.2 = 4$	$8.3 \text{ Mod } 2.2 = 0$

2.6 HIERARCHY OF OPERATIONS

Questions in meaning may arise when several operators appear in an expression. For example, does the expression $2 * x - 3 * y$ correspond to the algebraic term $(2x) - (3y)$ or to $2(x - 3y)$? Similarly, does the expression $a / b * c$ correspond to $a/(bc)$ or to $(a/b)c$? These questions are answered by the *hierarchy of operations* and the *order of execution within each hierarchical group*.

The hierarchy of operations is

1. *Exponentiation.* All exponentiation operations are performed first.
2. *Multiplication and division.* These operations are carried out after all exponentiation operations have been performed. Multiplication does not necessarily precede division.
3. *Integer division.* Integer division operations are carried out after all multiplication and (ordinary) division operations.
4. *Integer remainder.* Integer remainder operations are carried out after all integer divisions operations.
5. *Addition and subtraction.* These operations are the last to be carried out. Addition does not necessarily precede subtraction.

Within a given hierarchical group, the operations are carried out from left to right.

EXAMPLE 2.9

The arithmetic expression

$$a / b * c$$

is equivalent to the mathematical expression $(a/b)c$, since the operations are carried out from left to right.

Similarly, the arithmetic expression

$$b ^ 2 - 4 * a * c$$

is equivalent to the mathematical expression $b^2 - (4ac)$. In this case, the quantity $b ^ 2$ is formed initially, followed by the product $4 * a * c$ [first $4 * a$, then $(4 * a) * c$]. The subtraction is performed last, resulting in the final numerical quantity $(b ^ 2) - (4 * a * c)$.

EXAMPLE 2.10

Suppose we want to evaluate the algebraic term

$$[2(a + b)^2 + (3c)^2] ^ {m / (n+1)}$$

A Visual Basic expression corresponding to this algebraic term is

$$(2 * (a + b) ^ 2 + (3 * c) ^ 2) ^ (m / (n + 1))$$

If there is some uncertainty in the order in which the operations are carried out, we can introduce additional pairs of parentheses, giving

$$((2 * ((a + b) ^ 2)) + ((3 * c) ^ 2)) ^ (m / (n + 1))$$

Both expressions are correct. The first expression is preferable, however, since it is less cluttered with parentheses and therefore easier to read.

2.8 DISPLAYING OUTPUT – THE Print STATEMENT

The Print statement is used to display information within the currently active form, beginning in the upper left corner. This statement is not used often in Visual Basic projects. However, it is very convenient for displaying the results of very simple programs, and it provides a way to view the results of small program segments during the development of a large project.

The Print statement consists of the keyword Print, followed by a list of output items. The output items can be numeric constants, string constants, or expressions. Successive items must be separated either by commas or semicolons. Commas result in wide separation between data items; semicolons result in less separation. Each new Print statement will begin a new line of output. An empty Print statement will result in a blank line.

EXAMPLE 2.11

A Visual Basic program contains the following statements.

```
Dim Student As String, X As Integer, C1 As Single, C2 As Single
. . . . .
Student = "Aaron"
X = 39
C1 = 7
C2 = 11
. . . . .
Print "Name:", Student, X, (C1 + C2) / 2
```

The Print statement will generate the following line of output:

```
Name:      Aaron      39      9
```

If the Print statement had been written with semicolons separating the data items, e.g.,

```
Print "Name: "; Student; X; (C1 + C2) / 2
```

then the output data would be spaced more closely together, as shown below.

```
Name: Aaron 39 9
```

Now suppose the original Print statement had been replaced by the following three successive Print statements:

```
Print "Name: "; Student
Print
Print X,, (C1 + C2) / 2
```

Notice the repeated comma in the last Print statement.

The output would appear as

```
Name: Aaron

39      9
```

The empty Print statement would produce the blank line separating the first and second lines of output. Also, the repeated comma in the last Print statement would increase the separation between the two data items.

2.9 LIBRARY FUNCTIONS

Visual Basic contains numerous *library functions* that provide a quick and easy way to carry out many mathematical operations, manipulate strings, and perform various logical operations. These library functions are prewritten routines that are included as an integral part of the language. They may be used in place of variables within an expression or a statement. Table 2.1 presents several commonly used library functions.

A library function is accessed simply by stating its name, followed by whatever information must be supplied to the function, enclosed in parentheses. A numeric quantity or string that is passed to a function in this manner is called an *argument*. Once the library function has been accessed, the desired operation will be carried out automatically. The function will then return the desired value.

Table 2.1 Commonly Used Library Functions

<i>Function</i>	<i>Application</i>	<i>Description</i>
Abs	$y = \text{Abs}(x)$	Return the absolute value of x ; $y = x $.
Cdbl, CInt, CSng, CStr, CVar, etc.	$y = \text{CInt}(x)$	Convert x to the appropriate data type (Cdbl converts to double, CInt to integer, CSng to single, etc.).
Chr	$y = \text{Chr}(x)$	Return the character whose numerically encoded value is x . For example, in the ASCII character set, $\text{Chr}(65) = "A"$.
Cos	$y = \text{Cos}(x)$	Return the cosine of x (x must be in radians).
Date	$y = \text{Date}$	Return the current system date.
Exp	$y = \text{Exp}(x)$	Return the value of e to the x power; $y = e^x$.
Format	$y = \text{Format}(x, \text{"fmt str"})$	Return the value of x in a format designated by "fmt str" (format string). Note that the format string may take on several different forms.
Int	$y = \text{Int}(x)$	Return the largest integer that algebraically does not exceed x . For example, $\text{Int}(-1.9) = -2$.
Lcase	$y = \text{Lcase}(x)$	Return the lowercase equivalent of x .
Left	$y = \text{Left}(x, n)$	Return the leftmost n characters of the string x .
Len	$y = \text{Len}(x)$	Return the length (number of characters) of x .
Log	$y = \text{Log}(x)$	Return the natural logarithm of x ; $y = \log_e(x)$, $x > 0$.
Mid	$y = \text{Mid}(x, n1, n2)$	Return the middle $n2$ characters of the string x , beginning with character number $n1$.
Right	$y = \text{Right}(x, n)$	Return the rightmost n characters of the string x .
Rnd	$y = \text{Rnd}$	Return a random number, uniformly distributed within the interval $0 \leq y < 1$.
Sgn	$y = \text{Sgn}(x)$	Determine the sign of x ; ($y = +1$ if x is positive, $y = 0$ if $x = 0$, and $y = -1$ if x is negative).
Sin	$y = \text{Sin}(x)$	Return the sine of x (x must be in radians).
Sqr	$y = \text{Sqr}(x)$	Return the square root of x ; $y = \sqrt{x}$, $x > 0$.
Str	$y = \text{Str}(x)$	Return a string whose characters comprise the value of x . For example, $\text{Str}(-2.50) = "-2.50"$.
Tan	$y = \text{Tan}(x)$	Return the tangent of x (x must be in radians).
Time	$y = \text{Time}$	Return the current system time.
Ucase	$y = \text{Ucase}(x)$	Return the uppercase equivalent of x .
Val	$y = \text{Val}(x)$	Return a numeric value corresponding to the string x , providing x has the appearance of a number. For example, $\text{Val}("-2.50") = -2.5$.

EXAMPLE 2.12

Suppose we wanted to calculate the square root of the value represented by the expression $\text{Area} / 3.141593$, using the library function `Sqr`. To do so, we could write

```
Radius = Sqr(Area / 3.141593)
```

Notice that the argument of `Sqr` is the numeric expression $(\text{Area} / 3.141593)$.

Of course, we could also have written

```
Radius = (Area / 3.141593) ^ 0.5
```

EXAMPLE 2.13

The `Int` function can be confusing, particularly with negative arguments. The values resulting from several typical function calls are shown below.

```
Int(2.3) = 2
```

```
Int(-2.3) = -3
```

```
Int(2.7) = 2
```

```
Int(-2.7) = -3
```

Remember that `Int` produces a value whose magnitude is equal to or *smaller* than its argument if the argument is *positive*, and equal to or *larger* (in magnitude) than its argument if the argument is *negative*.

Some functions, such as `Log` and `Sqr`, require positive arguments. If a negative argument is supplied, an error message will be generated when an attempt is made to evaluate the function.

EXAMPLE 2.14

A Visual Basic program contains the statements

```
x = -2.7
. . . . .
y = Sqr(x)      (Notice the negative value assigned to x.)
```

When the program is executed, the following error message will be displayed:

```
Run-time error '5':
Invalid procedure call or argument
```

The execution will then cease.

Similarly, the statement

```
y = Log(x)
```

will produce the same error message when the program is executed.

EXAMPLE 2.15

The Format function allows a data item to be displayed in many different forms. Several possibilities are shown below. Many other variations are possible.

<i>Expression</i>	<i>Result</i>	
Print Format(17.66698, "##.##")	17.67	
Print Format(7.66698, "##.##")	7.67	(note the leading blank space)
Print Format(0.66667, "##.###")	.667	(note the leading blank spaces)
Print Format(0.66667, "#0.###")	0.667	(note the leading blank space)
Print Format(12345, "##,###")	12,345	
Print Format(12345, "##,###.00")	12,345.00	
Print Format("Basic", "&&&&&&&")	Basic	
Print Format("Basic", "@@@@@@")	Basic	(note the leading blank spaces)
Print Format(Now, "mm-dd-yyyy")	1-20-2001	
Print Format(Now, "mm/dd/yy")	1/20/01	
Print Format(Now, "hh:mm:ss am/pm")	04:47:51 pm	

Note that Now is a predefined Visual Basic variable that represents the current date and time, as determined by the computer's real-time clock.

2.10 PROGRAM COMMENTS

Comments provide a convenient means to *document* a program (i.e., to provide a program heading, to identify important variables, to distinguish between major logical segments of a program, to explain complicated logic, etc.). A comment consists of a single apostrophe ('), followed by a textual message. Comments can be inserted anywhere in a Visual Basic program. They have no effect on the program execution.

EXAMPLE 2.16

A Visual Basic program includes the following statements:

```
'Program to Calculate the Roots of a Quadratic Equation
. . . . .
X1 = (-b + root) / (2 * a)      'calculate the first root
X2 = (-b - root) / (2 * a)      'calculate the second root

Print X1, X2
```

The entire first line is a comment, which serves as a program heading. On the other hand, the last two lines each have a comment attached at the end of an executable statement. Note that each comment begins with a single apostrophe.

Chapter 3

Branching and Looping

3.1 COMPARISON OPERATORS

In order to carry out branching operations in Visual Basic, we must be able to express conditions of equality and inequality. To do so, we make use of the following *comparison operators*:

Equal:	=
Not equal:	<>
Less than:	<
Less than or equal to:	<=
Greater than:	>
Greater than or equal to:	>=

3.2 LOGICAL OPERATORS

In addition to the relational operators, Visual Basic contains several *logical operators*. They are

And will result in a condition that is true if both expressions are true

Or will result in a condition that is true if either expression is true, or if they are both true

Xor (exclusive or) will result in a condition that is true *only* if one of the expressions is true and the other is false.

Not is used to reverse the value of a logical expression (e.g., from true to false, or false to true).

3.3 BRANCHING WITH THE If -Then BLOCK

An If -Then block is used to execute a single statement or a block of statements on a conditional basis. There are two different forms. The simplest is the single-line, single-statement If -Then, which is written as

```
If logical expression Then executable statement
```

The *executable statement* will be executed only if the *logical expression* is true. Otherwise, the statement following If -Then will be executed next. Note that the executable statement must appear on the same line as the logical expression; otherwise, an End If statement will be required (see below).

EXAMPLE 3.1

A typical situation utilizing an If -Then statement is shown below.

```
If x < 0 Then x = 0  
Sum = Sum + x
```

Here is a more general form of an If -Then block:

```
If logical expression Then  
    . . . . .  
    executable statements  
    . . . . .  
End If
```

EXAMPLE 3.2

The following If -Then block permits a single group of statements to be executed conditionally.

```
IF income <= 14000 THEN  
    tax = 0.2 * pay  
    net = pay - tax  
END IF
```

The assignment statements will be executed only if the logical expression `income <= 14000` is true.

3.4 BRANCHING WITH If-Then-Else BLOCKS

An If-Then-Else block permits one of two different groups of executable statements to be executed, depending on the outcome of a logical test. Thus, it permits a broader form of branching than is available with a single If-Then block.

```
In general terms, an If-Then-Else block is written as
```

```
  If logical expression Then
    . . . . .
    executable statements
    . . . . .
  Else
    . . . . .
    executable statements
    . . . . .
  End If
```

EXAMPLE 3.3

A typical If-Then-Else sequence is shown below. This sequence allows us to calculate either the area and circumference of a circle or the area and circumference of a rectangle, depending on the string that is assigned to the variable form.

```
pi = 3.141593
If (form = "circle") THEN           'circle
  area = pi * radius ^ 2
  circumference = 2 * pi * r
Else                                 'rectangle
  area = length * width
  circumference = 2 * (length + width)
End If
```

A more general form of the If-Then-Else block can be written as

```
  If logical expression 1 Then
    . . . . .
    executable statements
    . . . . .
  ElseIf logical expression 2 Then
    . . . . .
    executable statements
    . . . . .
  repeated ElseIf clauses
    . . . . .
  Else
    . . . . .
    executable statements
    . . . . .
  End If
```

EXAMPLE 3.4 ROOTS OF A QUADRATIC EQUATION

The roots of the quadratic equation $ax^2 + bx + c = 0$ can be determined using the well-known formulas

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

provided the quantity $b^2 - 4ac$ is positive.

If $b^2 - 4ac$ is zero, we have a single (repeated) real root, determined as

$$x = -b / 2a$$

If $b^2 - 4ac$ is negative, we have two complex roots. In this case,

$$x_1 = \frac{-b + \sqrt{4ac - b^2} i}{2a}$$

$$x_2 = \frac{-b - \sqrt{4ac - b^2} i}{2a}$$

where i represents the *imaginary number* $\sqrt{-1}$

In Visual Basic, we can accommodate these three situations with a group of If-Then-ElseIf-Else shown below.

```
'Roots of a Quadratic Equation
Dim a, b, c, d, x, x1, x2, real, imag           'variant data types
d = (b ^ 2 - 4 * a * c)
If d > 0 Then                                  'real roots
    x1 = (-b + Sqr(d)) / (2 * a)
    x2 = (-b - Sqr(d)) / (2 * a)
ElseIf d = 0 Then                              'repeated root
    x = -b / (2 * a)
Else                                           'complex roots
    real = -b / (2 * a)
    imag = Sqr(-d) / (2 * a)
End If
```

3.5 SELECTION: Select Case

One way to select a block of statements from several competing blocks is to use a series of If-Then-Else or If-Then-ElseIf-Else blocks. This can be tedious, however, if the number of competing blocks is moderately large. The Select Case structure frequently offers a simpler approach to this type of situation.

The most common form of the Select Case structure is written in general terms as

```
Select Case expression

  Case value1
    executable statements

  Case value2
    executable statements

    . . . . .

  Case Else
    executable statements

End Select
```

EXAMPLE 3.5

Here is a Visual Basic program segment that makes use of a Select Case structure.

```
'Raise x to a Selected Power
Dim x, z, n As Integer
Select Case n                'select a group of statements
Case 1                      'x ^ 1
  z = x
Case 2                      'x ^ 2
  z = x ^ 2
Case 3                      'x ^ 3
  z = x ^ 3
Case Else                   'error
  MsgBox("ERROR - Please try again")
End Select
```

3.6 LOOPING WITH For-Next

The For-Next structure is a block of statements that is used to carry out a looping operation; that is, to execute a sequence of statements some predetermined number of times. The structure begins with a For-To statement and ends with a Next statement. In between are the statements to be executed.

In its simplest form, a For-Next structure is written as

```
For index = value1 To value2
    . . . . .
    executable statements
    . . . . .
Next index
```

EXAMPLE 3.6

A typical For-To loop structure is shown below.

```
sum = 0
For i = 1 To 10
    sum = sum + i
Next i
```

This structure will result in 10 passes through the loop. During the first pass, *i* will be assigned a value of 1; *i* will then increase by 1 during each successive pass through the loop, until it has reached its final value of 10 in the last pass. Within each pass, the current value of *i* is added to *sum*. Hence, the net effect of this program segment is to determine the sum of the first 10 integers (i.e., $1 + 2 + \dots + 10$).

A more general form of the For-Next structure can be written as

```
For index = value1 To value2 Step value3
    . . . . .
    executable statements
    . . . . .
Next index
```

EXAMPLE 3.7

The loop structure

```
sum = 0
For count = 2.5 To -1 STEP -0.5
    sum = sum + count
Next count
```

will cause *count* to take on the values 2.5, 2.0, 1.5, . . . , 0.0, -0.5, -1.0. Hence, the final value of *sum* will be 6.0 (because $2.5 + 2.0 + 1.5 + 1.0 + 0.5 + 0.0 - 0.5 - 1.0 = 6.0$). Note that this structure will generate a total of eight passes through the loop.

3.7 LOOPING WITH Do-Loop

In addition to For-Next structures, Visual Basic also includes Do-Loop structures, which are convenient when the number of passes through a loop is not known in advance (as, for example, when a loop is required to continue until some logical condition has been satisfied).

A Do-Loop structure always begins with a Do statement and ends with a Loop statement. However, there are four different ways to write a Do-Loop structure. Two of the forms require that a logical expression appear in the Do statement (i.e., at the beginning of the block); the other two forms require that the logical expression appear in the Loop statement (at the end of the block).

The general forms of the Do-Loop structure are shown below.

First form:

```
Do While logical expression
. . . . .
executable statements
. . . . .
Loop
```

Second form:

```
Do Until logical expression
. . . . .
executable statements
. . . . .
Loop
```

Third form:

```
Do
. . . . .
executable statements
. . . . .
Loop While logical expression
```

Fourth form:

```
Do
. . . . .
executable statements
. . . . .
Loop Until logical expression
```

The first form continues to loop as long as the *logical expression* is true, whereas the second form continues to loop as long as the *logical expression* is *not* true (until the *logical expression* becomes true). Similarly, the third form continues to loop as long as the *logical expression* is true, whereas the fourth form continues to loop as long as the *logical expression* is *not* true.

Note that there is a fundamental difference between the first two forms and the last two forms of the Do-Loop block. In the first two forms, the logical test is made at the *beginning* of each pass through the loop; hence, it is possible that there will not be *any* passes made through the loop, if the indicated logical condition is not satisfied. In the last two forms, however, the logical test is not made until the *end* of each pass; therefore, at least one pass through the loop will always be carried out.

EXAMPLE 3.8

Consider the following two Do-Loop structures.

```
Private Sub Command1_click()
Dim Name As String
Do While Name<>"Ali"
Name=InputBox("EnterYournameor Ali To quit")
If Name<>"Ali" Then Print Name
Loop
End Sub
```

```
Private Sub Command1_click()
Dim Name As String
Do
Name=InputBox("EnterYournameor Ali To quit")
If Name<>"Ali" Then Print Name
Loop While Name<>"Ali"
End Sub
```

EXAMPLE 3.9

```
sum = 0
count = 1
Do While count <= 10
    sum = sum + count
    count = count + 1
Loop
```

This structure will result in 10 passes through the loop. Note that count is assigned a value of 1 before entering the loop. The value of count is then incremented by 1 during each pass through the loop. Once the value of count exceeds 10, the execution will cease.

Here is another way to accomplish the same thing.

```
sum = 0
count = 1
Do
    sum = sum + count
    count = count + 1
Loop While count <= 10
```

If we choose to use an Until clause rather than a While clause, we can write the control structure in either of the following ways.

```
sum = 0
count = 1
Do Until count > 10
    sum = sum + count
    count = count + 1
Loop

sum = 0
count = 1
Do
    sum = sum + count
    count = count + 1
Loop Until count > 10
```

Note that the logical expression in these two structures (count > 10) is the opposite of the logical expression in the first two structures (count <= 10).

Control can be transferred out of a Do-Loop block using the Exit Do statement. This statement is analogous to Exit For, which is used with For-Next blocks. Thus, when an Exit Do statement is encountered during program execution, control is transferred out of the Do-Loop block to the first executable statement following Loop.

EXAMPLE 3.10

```
sum = 0
count = 1
Do While count <= 10
    sum = sum + count
    If sum >= 10 Then
        Exit Do
    count = count + 1
Loop
```

Chapter 4

Visual Basic Control Fundamentals

4.1 VISUAL BASIC CONTROL TOOLS

In Chapter 1 we saw that the *Visual Basic Toolbox* (shown below in Fig. 4.1) contains a collection of control tools, such as labels, text boxes and command buttons. These controls, together with customized menus, allow us to build a broad variety of graphical user interfaces. In this chapter we will focus on several of the more commonly used *Toolbox* control tools. These tools, together with the material covered in the previous two chapters, will allow us to write complete Visual Basic programs.

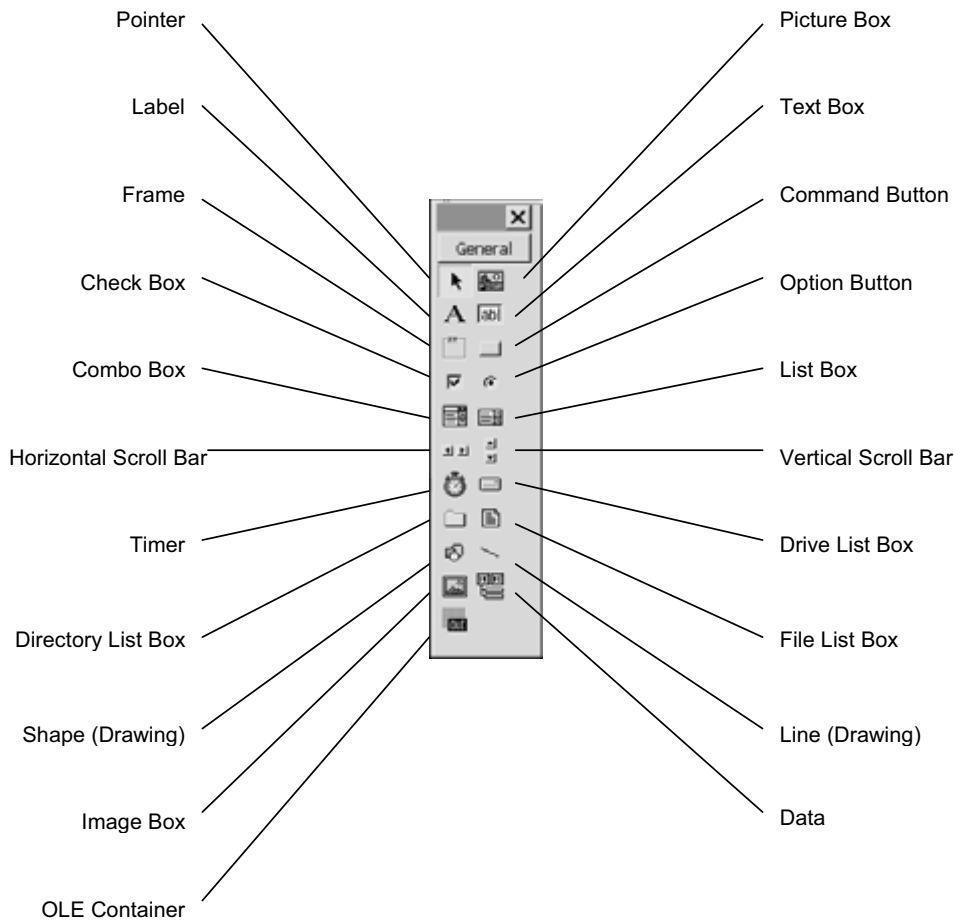


Fig. 4.1 The Visual Basic toolbox

Here, in alphabetical order, is a brief description of each control tool:

Check Box



Provides a means of specifying a Yes/No response. Within a group of check boxes, any number of boxes can be checked, including none. (See also the Option Box description.)

Combo Box



Combines the capabilities of a text box and a list box. Thus, it provides a collection of text items, one of which may be selected from the list at any time during program execution. Text items can be assigned initially, or they can be assigned during program execution. In addition, the user can enter a text item at any time during program execution.

Command Button



Provides a means of initiating an event action by the user clicking on the button.

Data



Provides a means of displaying information from an existing database.

Directory List Box



Provides a means of selecting paths and directories (folders) within the current drive.

Drive List Box



Provides a means of selecting among existing drives.

File List Box



Provides a means of selecting files within the current directory.

Frame



Provides a container for other controls. It is usually used to contain a group of option buttons, check boxes or graphical shapes.

Horizontal Scroll Bar



Allows a horizontal scroll bar to be added to a control (if a horizontal scroll bar is not included automatically).

Image Box



Used to display graphical objects, and to initiate event actions. (Note that an Image Box is similar to a Picture Box. It redraws faster and can be stretched, though it has fewer properties than a Picture Box.)

Label



Used to display text on a form. The text cannot be reassigned during program execution, though it can be hidden from view and its appearance can be altered. (See also the Text Box description.)

Line

Used to draw straight-line segments within forms. (See also the Shape tool description.)

List Box

Provides a collection of text items. One text item may be selected from the list at any time during program execution. Text items can be assigned initially, or they can be assigned during program execution. However, in contrast to a text box, the user cannot enter text items to a list box during program execution. (Note that a combo box combines the features of a list box and a text box).

OLE Container

Allows a data object to be transferred from another Windows application and embedded within the Visual Basic application.

Option Button

Provides a means of selecting one of several different options. Within a group of option buttons, one and only one can be selected. (See also the Check Box description.)

Picture Box

Used to display graphical objects or text, and to initiate event actions. (Note that a Picture Box is similar to an Image Box. It has more properties than an Image Box, though it redraws slower and cannot be stretched.)

Pointer

The pointer is not really a control tool, in the true sense of the word. When the pointer is active, the mouse can be used to position and resize other controls on the design form, and to double-click on the controls, resulting in a display of the associated Visual Basic code.

Shape

Used to draw circles, ellipses, squares and rectangles within forms, frames or picture boxes. (See also the Line tool description.)

Text Box

Provides a means of entering and displaying text. The text can be assigned initially, it can be reassigned during program execution, or it can be entered by the user during program execution. (See also the Label Box and the Combo Box descriptions.)

Timer

Allows events to occur repeatedly at specified time intervals.

Vertical Scroll Bar

Allows a vertical scroll bar to be added to a control (if a vertical scroll bar is not included automatically).

4.2 CONTROL TOOL CATEGORIES

The control tools can be grouped into the following overall categories: (Keep in mind that some control tools have multiple uses and are not restricted to the categories listed below.)

Entering Text

Text Box
Combo Box

Drawing

Line Button
Shape Button

Displaying Text

Label
Text Box
List Box
Combo Box

Selecting Among Alternatives

Check Box
Option Button
Frame
List Box

Displaying Graphics

Image Box
Picture Box
Frame

Viewing Windows

Frame
Horizontal Scroll Bar
Vertical Scroll Bar

Managing Files

File List Box
Drive List Box
Directory List Box

Accessing Existing Data

Data

Initiating Events

Command Button

Linking with Other Objects

OLE

Executing Timed Events

Timer

4.3 WORKING WITH CONTROLS

A control can be *added* to the Form Design Window two different ways:

1. By clicking on the desired control tool within the Toolbox, then clicking on the control's location within the Form Design Window.
2. By double-clicking on the desired control tool within the Toolbox, automatically placing the control at the center of the Form Design Window.

A control can be *relocated* within the Form Design Window by dragging the control to its desired location (hold down the left mouse button and drag).

A control can be *resized* within the Form Design Window by dragging one of its edges or corners.

A control can be *removed* from the Form Design Window by highlighting the control (i.e., by clicking on it) and then pressing the Delete key.

4.4 NAMING FORMS AND CONTROLS

When an object (i.e., a form or control) is added to the Form Design Window, a generic default name (e.g., Form1, List1, List2, Text1, etc.) is automatically assigned to that object. Each name includes a generic identifier

(Form, List, Text, etc.) that identifies the type of object, followed by a number that identifies the order in which that particular object type has been added to the Form Design Window. Thus, List1 is the name of the first list box added to the Form Design Window, List2 is the name of the second list box, and so on.

The default names work well for simple applications. For more complicated applications, however, it may be preferable to assign different names that suggest the purpose of each object. Thus, Students and Addresses may be preferable to List1 and List2.

Microsoft suggests that such programmer-assigned names include a three-letter prefix suggesting the type of object. Hence, we might use lstStudents and lstAddresses rather than Students and Addresses, if each object is a list box. (The use of prefixes is unnecessary when the default names are used, since the names themselves indicate the object type.)

Microsoft recommends the following prefixes for programmer-defined object names:

<u>Object</u>	<u>Prefix</u>	<u>Object</u>	<u>Prefix</u>
Combo Box	cbo	Label	lbl
Check Box	chk	Line	lin
Command Button	cmd	List Box	lst
Data	dat	Menu	mnu
Directory List Box	dir	OLE	ole
Drive List Box	drv	Option Button	opt
File List Box	fil	Picture Box	pic
Frame	fra	Shape	shp
Form	frm	Text Box	txt
Horizontal Scroll Bar	hsb	Timer	tmr
Image Box	img	Vertical Scroll Bar	vsb

4.5 ASSIGNING PROPERTY VALUES TO FORMS AND CONTROLS

The properties associated with each object type are unique, though some, such as Name, BackColor (i.e., background color), Height and Width, are common to many different object types. The meaning of most properties is readily apparent. Some, however, require further explanation, particularly certain unique properties that are required for an object's special behavior. For information about such properties, you should consult the on-line help (press F1 or click on the Help menu), related example projects, or printed reference material.

Moreover, each object will have a unique set of values assigned to its properties. These values may be assigned at *design time* (i.e., when the object is first defined, before the application is executed), or at *run time* (i.e., while the application is executing).

Design-time assignments are made by selecting a property from the list of properties shown in the Properties Window (see Fig. 1.5), and then either choosing an appropriate value from the adjoining list of values or entering a value from the keyboard. These property values will apply when the application first begins to run. In general terms, a property assignment is written as

$$\textit{object_name.property} = \textit{value}$$

where *object_name* refers to the name of the form or control, *property* refers to the associated property name, and *value* refers to an assignable item, such as a number or a string. The net effect is to assign the value on the right-hand side of the equal sign to the property on the left. Such assignments can provide initial values to properties that were formerly undefined, or they may replace previous assignments.

EXAMPLE 4.1 ASSIGNING VALUES TO PROPERTIES

Each of the following commands assigns a run-time value to a text box property.

```
Text1.Text = "Welcome to Visual Basic"  
txtMessage.Text = "Welcome to Visual Basic"  
txtMessage.Height = 300
```

The first line assigns the string "Welcome to Visual Basic" to the Text property associated with text box Text1 (default name). The second line assigns this same string to the Text property associated with text box txtMessage (programmer-assigned name). The last line assigns a numerical value to the Height property associated with txtMessage.

Each of these commands either assigns a new value during program execution, or replaces a previously assigned initial value (i.e., a value assigned during design time).

4.6 EXECUTING COMMANDS (EVENT PROCEDURES AND COMMAND BUTTONS)

An *event procedure* is an independent group of commands that is executed whenever an "event" occurs during program execution. Typically, an event occurs when the user takes some action, such as clicking on a control icon, or dragging an icon to another location. Many (though not all) Visual Basic controls have event procedures associated with them.

Each event procedure begins with a **Sub** statement, such as `Private Sub Command1_Click()`, and ends with an `End Sub` statement. Between the `Sub` and `End Sub` statements is a group of instructions, such as those discussed in the last two chapters, that are executed when the user initiates the corresponding event. The parentheses in the `Sub` statement may contain *arguments* – special variables that are used to transfer information between the event procedure and the "calling" routine.

Command buttons are often used to execute Visual Basic event procedures. Thus, when the user clicks on a command button during program execution, the statements within the corresponding event procedure are carried out. The statements within the event procedure may involve the properties of controls other than the command button. For example, a command-button event procedure may result in new values being assigned to the properties of a label or a text box.

EXAMPLE 4.2 A SAMPLE EVENT PROCEDURE

A typical event procedure is shown below:

```
Private Sub Command1_Click()  
Label1.Caption = "Hello, " & Text1.Text & "! Welcome to Visual Basic."  
Label1.BorderStyle = 1  
Label1.Visible = True  
End Sub
```

From the first line (i.e., the `Sub` statement), we see that this event procedure is associated with command button `Command1`, and it is a response to a click-type event. The three assignment statements within the event procedure will be executed whenever the user clicks the mouse on command button `Command1` during program execution.

To enter an event procedure, double-click on the appropriate command button within the Form Design Window or click once on the command button (to activate it), and then select the Code Editor by clicking on the leftmost button within the Project Window toolbar (see Fig. 4.2). You may then enter the required Visual Basic commands within the corresponding event procedure .

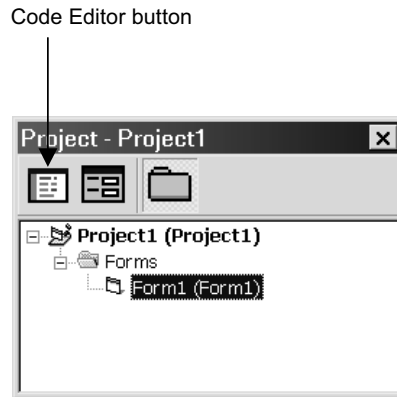


Fig. 4.2 The Visual Basic Project Window

In the next section we will see how the Code Editor is used to enter an event procedure and associate that event procedure with a command button.

4.7 DISPLAYING OUTPUT DATA (LABELS AND TEXT BOXES)

The most straightforward way to display output data is with a label or a text box. A label can only display output data, though a text box can accept input data as well as display output data. For now, however, we will work only with output data.

Both of these controls process information in the form of a string. This is not a serious limitation, however, because numeric values can easily be converted to strings via the Str function .

To display output using a label, the basic idea is to assign a string containing the desired output information to the label's Caption property. Similarly, when displaying output using a text box, a string containing the desired output information is assigned to the text box's Text property. The following example illustrates the technique.

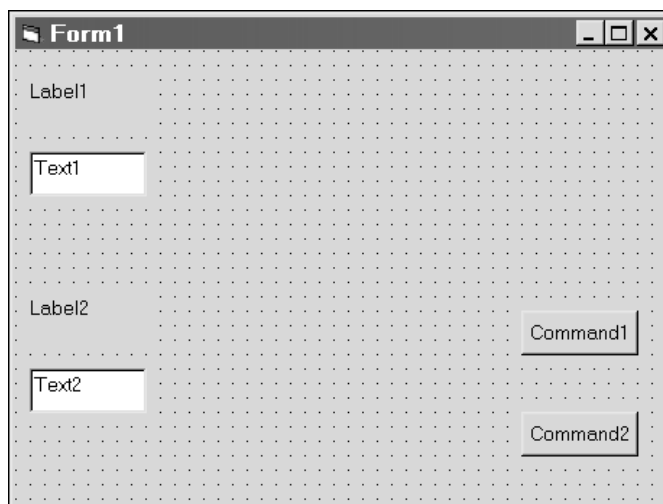


Fig. 4.3

Chapter 5

WORKING WITH MENUS AND DIALOG BOXES

This chapter focuses on processing input from another source in the user interface: menu commands and dialog boxes. In this chapter, you will learn how to:

- ◆ Add menus to your programs by using the Menu Editor.
- ◆ Process menu choices by using program code.
- ◆ Use the **CommonDialog** ActiveX control to display standard dialog boxes.

5.1: Creating Menus

Menus, which are located on the menu bar of a form, contain a list of related commands. When you click a menu title in a Windows-based program, a list of menu commands should always appear in a well-organized list.

Most menu commands run immediately after they are clicked. For example, when the user clicks the **Edit** menu **Copy** command, Windows immediately copies information to the Clipboard. However, if ellipsis points (...) follow the menu command, Visual Basic displays a dialog box that requests more information before the command is carried out.

This section includes the following topics:

- Using the Menu Editor
- Adding Access and Shortcut Keys
- Processing Menu Choices

5.1.1: Using The Menu Editor

The Menu Editor is a Visual Basic dialog box that manages menus in your programs. With the Menu Editor, you can:

- Add new menus
- Modify and reorder existing menus
- Delete old menus
- Add special effects to your menus, such as access keys, check marks, and keyboard shortcuts.

See the Figure below for Menu editor Window

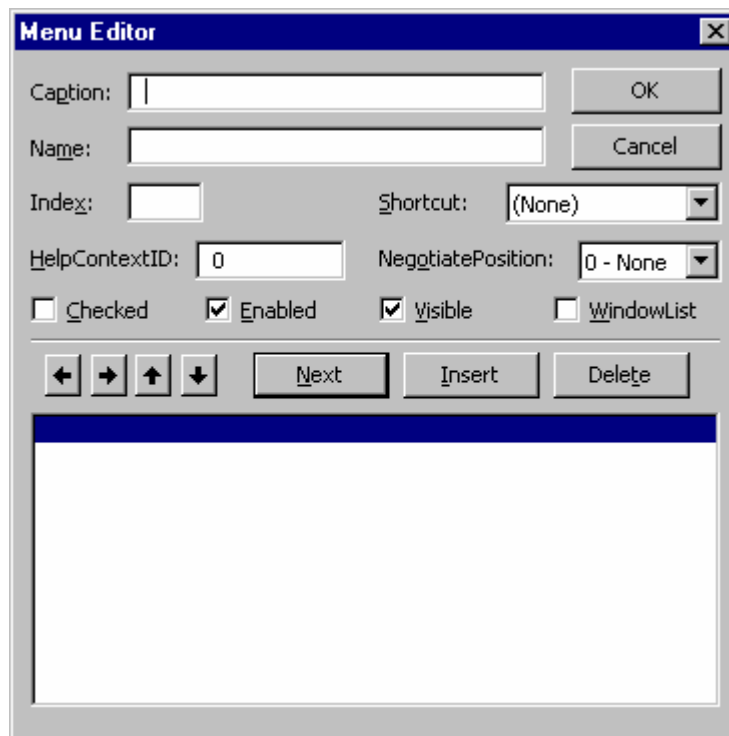


Figure 1: The Menu Editor Window

Creating Menu Command Lists

To build lists of menu commands, you first need to create the menus and then add them to the program menu bar.

► To create a list of menu commands on a form

1. Click the form itself (not an object on the form).
2. On the Visual Basic toolbar, click the Menu Editor icon, or select **Menu Editor** from the **Tools** menu.
3. In the **Caption** text box, type the menu caption (the name that will appear on the menu bar), and then press TAB.
4. In the **Name** text box, type the menu name (the name the menu has in the program code).
By convention, programmers use the *mnu* object name prefix to identify both menus and menu commands.
5. To add the menu to your program menu bar, click **Next**.

The Menu Editor clears the dialog box for the next menu item. As you build your menus, the structure of the menus and commands appear at the bottom of the dialog box.

6. In the **Caption** text box, type the caption of your first menu command.
7. Press tab, and then type the object name of the command in the **Name** text box.
8. With this first command highlighted in the menu list box, click the right arrow button in the Menu Editor.

In the **Menu** list box, the command moves one indent (four spaces) to the right. Click the right arrow button in the Menu Editor dialog box to move items to the right, and click the left arrow button to move items to the left.

9. Click **Next**, and then continue to add commands to your menu.

The position of list box items determines what they are:

List box item	Position
Menu title	Flush left
Menu command	One indent
Submenu title	Two indents
Submenu command	Three indents

► To add more menus

1. When you're ready to add another menu, click the left arrow button to make the menu flush left in the **Menu** list box.
2. To add another menu and menu commands, repeat Steps 3 through 9 in the preceding procedure.
3. When you're finished entering menus and commands, click **OK** to close the Menu Editor. (Don't accidentally click **Cancel** or all your menu work will be lost.)

The Menu Editor closes, and your form appears in the programming environment with the menus you created.

Adding Event Procedures

After you add menus to your form, you can use event procedures to process the menu commands. Clicking a menu command on the form in the programming environment displays the event procedure that runs when the menu command is chosen. You'll learn how to create event procedures that process menu selections in Processing Menu Choices.

5.1.2: Adding Access and Shortcut Keys

Visual Basic makes it easy to provide access key and shortcut key support for menus and menu commands.

Access and Shortcut Keys

The access key for a command is the letter the user can press to execute the command when the menu is open. The shortcut key is the key combination the user can press to run the command without opening the menu. Here's a quick look at how to add access and shortcut keys to existing menu items:

Add an access key to a menu item Start the Menu Editor. Prefix the access key letter in the menu item caption with an **ampersand (&)**.

Add a shortcut key to a menu command Start the Menu Editor. Highlight the command in the menu list box. Pick a key combination from the Shortcut drop-down list box.

Creating Access and Shortcut Keys

You can create access keys and shortcut keys either when you first create your menu commands or at a later time.

The following illustration shows the menu commands associated with two menus, **File** and **Clock**. Each menu item has an access key ampersand character, and the **Time** and **Date** commands are assigned shortcut keys. See figure below.

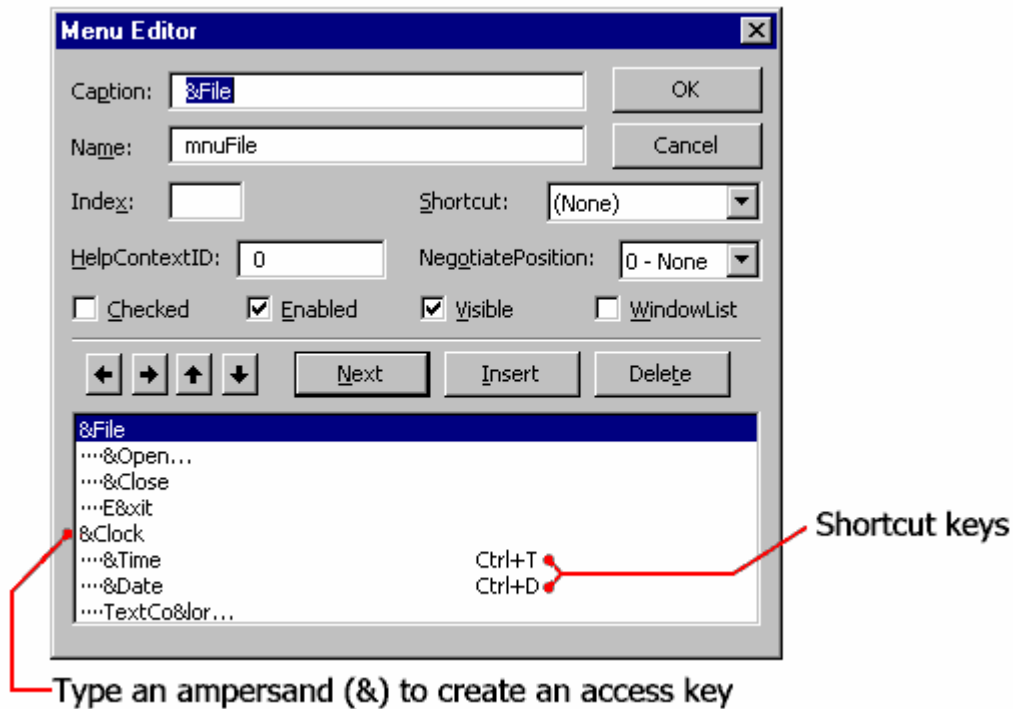


Figure 2: Menu Editor Window showing how to create shortcut keys

5.1.3: Processing Menu Choices

After you place menu items on the menu bar, they become objects in the program. To make the menu objects do meaningful work, you need to write event procedures for them. Typically, menu event procedures:

- ◆ Contain program statements that display or process information on a form.
- ◆ Modify one or more object properties.

For example, the event procedure for a command named **Time** might use the **Time keyword** to display the current system time in a text box.

Processing the selected command might require additional information (you might need to open a file on disk, for example). If so, you can display a dialog box to receive user input by using a common dialog box. You'll learn this technique in the next section.

Disabling a Menu Command

In a typical Windows application, not all menu commands are available at the same time. In a typical **Edit** menu, for example, the **Paste** command is available only when there is data on the Clipboard. When a command is disabled, it appears in dimmed (gray) type on the menu bar. You can disable a menu item by:

- ◆ Clearing the **Enabled** check box for that menu item in the Menu Editor.
- ◆ Using program code to set the item's Enable property to False. (When you're ready to use the menu command again, set its Enable property to True.)

5.2: Creating Dialog Boxes

A **dialog box** is simply a form in a program that contains input controls designed to receive information. To make your programming faster, Visual Basic includes an ActiveX control, named **CommonDialog**.

With this control, you can easily display **six standard dialog boxes** in your programs. These dialog boxes handle routine tasks such as opening files, saving files, and picking fonts. If the dialog box you want to use is not included in this ready-made collection of objects, you can create a new one by adding a second form to your program. This section includes the following topics:

- ◆ Using the CommonDialog Control
- ◆ Common Dialog Object Event Procedures

5.2.1: Using the Common Dialog Control

Before you can use the **CommonDialog** control, you need to verify that it is in your toolbox. If you don't see the CommonDialog icon, follow this procedure to add it to the toolbox.

► **To add the CommonDialog control to the toolbox**

1. From the **Project** menu, click **Components**.
2. Click the **Controlstab**.
3. Ensure that the **Selected Items Only** box is not checked.
4. Place a check mark next to **Microsoft Common Dialog Control**, and then click **OK**.

Creating a Dialog Box

Follow this procedure to create a dialog box with the **CommonDialog** control.

► **To create a common dialog object on your form**

1. In the toolbox, double-click the **CommonDialog** control.
2. When the common dialog object appears on your form, drag it to an out-of-the-way location.

Note: You cannot resize a common dialog object, and it **disappears** when your program runs. The common dialog object itself displays nothing — its only purpose is to display a standard dialog box on the screen when you use a method in program code to request it.

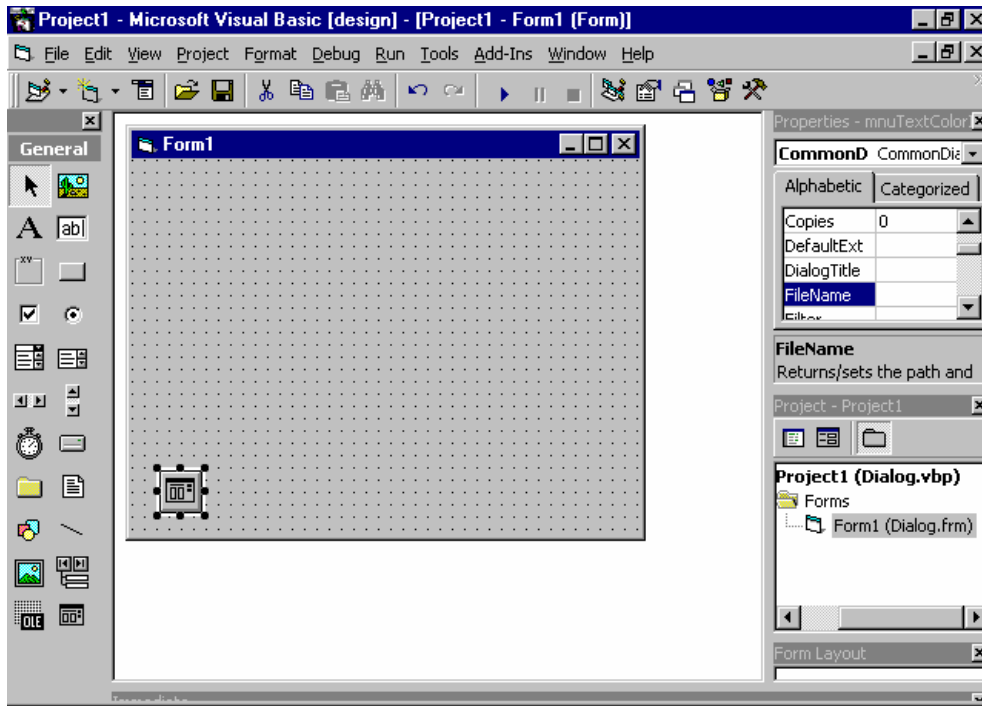


Figure 3: A Common Dialog object on a form

This table lists the name and purpose of the six standard dialog boxes that the common dialog object provides and the methods you use to display them:

Dialog Box	Purpose	Method
Open	Gets the drive, folder name, and file name for an existing file that is being opened.	ShowOpen
Save As	Gets the drive, folder name, and file name for a file that is being saved.	ShowSave
Print	Provides user-defined printing options.	ShowPrinter
Font	Provides user-defined font type and style options.	ShowFont
Help	Provides online user information.	ShowHelp
Color	Provides user-defined color selection from a palette.	ShowColor

5.2.2: Common Dialog Object event Procedures

To display a standard dialog box in a program, you need to call the common dialog object. You do this by using the appropriate object method in an event procedure. If necessary, you also use program code to set one or more common dialog object properties before the call. (For example, if you are using the **Open** dialog box, you might want to control what type of files is displayed in the list box.) Finally, your event procedure needs to process the choices made by the user when they complete the standard dialog box.

This section presents two simple event procedures, one that manages an **Open** dialog box and one that uses information received from a **Color** dialog box.

The following topics are included in this section:

- Creating an Open Dialog Box
- Creating a Color Dialog Box

5.2.2.1: Creating an Open Dialog Box

The following code window shows an event procedure named `mnuOpenItem_Click`. You can use this event procedure to display an **Open** dialog box when the user clicks the **Open** command on the **File** menu. The event procedure assumes that you have already created a **File** menu containing **Open** and **Close** commands and that you want to open Windows metafiles (.wmf). See the piece of code given below.

```
Private Sub mnuOpenItem_Click()  
    CommonDialog1.Filter = "Metafiles (*.WMF)|*.WMF"  
    CommonDialog1.ShowOpen  
    Image1.Picture = LoadPicture(CommonDialog1.FileName)  
    mnuCloseItem.Enabled = True  
End Sub
```

The event procedure uses these properties and methods:

Object	Property/Method	Purpose
Common Dialog	ShowOpen	Displays the dialog box.
Common Dialog	Filter	Defines the file type in the dialog box.
Menu	Enabled	Enables the Close menu command, which users can use to <u>unload</u> the picture.
Image	Picture	Opens the selected file.

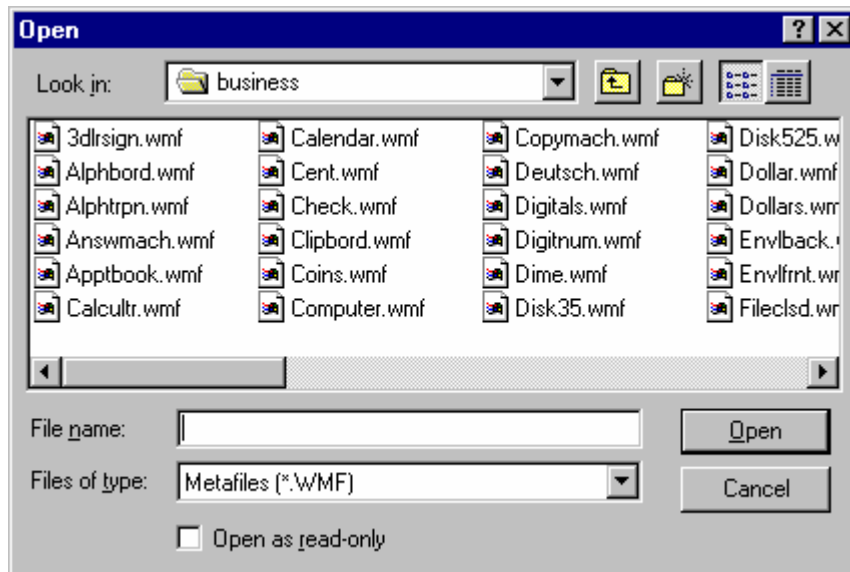


Figure 4:Open Dialog Box

5.2.2.2: Creating a Color Dialog Box

If you need to update the color of a user interface element while your program runs, you can prompt the user to pick a new color with the **Color** dialog box displayed by using the **Common Dialog** object. The color selections provided by the **Color** dialog box are controlled by the **Flags** property, and the **Color** dialog box is displayed with the **ShowColor** method.

This code window shows an event procedure that you can use to change the color of a label while your program runs. The value used for the **Flags** property — which in this case prompts Visual Basic to display a standard palette of color selections — is a hexadecimal (base 16) number. (To see a list of other potential values for the **Flags** property, search for *CommonDialog constants* in the Visual Basic online Help.) The event procedure assumes that you have already created a menu command named **TextColor** with the Menu Editor. See the code given below

```
Private Sub mnuTextColorItem_Click()
    CommonDialog1.Flags = &H1&
    CommonDialog1.ShowColor
    Label1.ForeColor = CommonDialog1.Color
End Sub
```

The figure below shows the color dialog box.

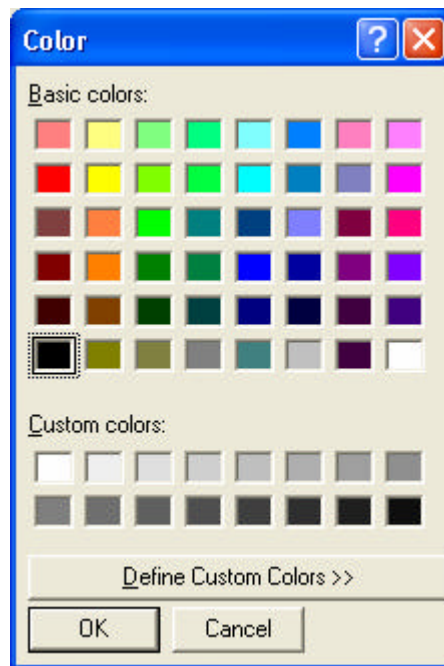


Figure 5: Color Dialog Box

EXERCISE: Creating a File Menu and Common Dialog Object Step by Step

In this exercise, you use the Menu Editor to create a **File** menu with **Open**, **Close**, and **Exit** commands for your program. You also assign access keys and shortcut keys to the commands, so you can run them from the keyboard.

► To create a File menu

1. Start Visual Basic and open a new, standard Visual Basic application.
2. On the toolbar, click **Menu Editor** to open the Menu Editor dialog box.
3. In the **Caption** text box, type **&File**.
4. In the **Name** text box, type **mnuFile**, and then click **Next**.

By placing the & character before the letter F, you specify F as the menu access key.

► To assign access and shortcut keys

1. In the **Caption** text box, type **&Open....**
2. In the **Name** text box, type **mnuOpenItem**.
3. To indent the selected (highlighted) command, click the right arrow button.
4. In the Shortcut drop-down list, click CTRL+O for a shortcut key, and then click **Next**.
5. In the **Caption** text box, type **&Close**.
6. In the **Name** text box, type **mnuCloseItem**.
7. In the Shortcut drop-down list, click CTRL+C for a shortcut key, and then click **Next**.
8. In the **Caption** text box, type **E&xit**.
9. In the **Name** text box, type **mnuExitItem**.
10. In the Shortcut drop-down list, click CTRL+X for a shortcut key, and then click **OK**.

► **To save your project**

1. From the **File** menu, click **Save Project As**.
2. Save your form and project to disk under the name "Picture". Visual Basic will prompt you for two file names — one for your form file (Picture.frm), and one for your project file (Picture.vbp).

► **To create a common dialog object**

1. Verify that the **CommonDialog** control is in your project toolbox. If it isn't, add it now by using the **Project** menu **Components** command.
2. To add a common dialog object to your form, double-click the **CommonDialog** control in the toolbox, and then drag the object to the lower right-hand side of the form.

► **To create the image object**

1. Click the **Image** control and create a large image object in the middle of your form.

When you run your program, the image object displays picture files with *.jpg extension on a form.

2. On the form, click **Image1**. To restore the Properties window to full size, double-click the Properties window title bar.

If you cannot see the Properties window, click **Properties** on the toolbar to display it.

3. Click the **Stretch** property and set it to **True**.

When you run your program, **Stretch** makes the metafile fill the entire image object.

4. On the toolbar, click **Save Project** to save these changes to your program.

► **To write event procedures**

1. In the Project window, click **View Code**, click the Code window **Object** drop-down list box, and then click **mnuOpenItem**.

2. In the mnuOpenItem_Click event procedure, type the following code:

```
CommonDialog1.Filter = "JPEG FILES (*.JPG)|*.JPG"  
CommonDialog1.ShowOpen  
Image1.Picture = LoadPicture(CommonDialog1.FileName)  
mnuCloseItem.Enabled = True
```

3. In the **Object** drop-down list box, click **mnuCloseItem**, and then type the following code:

```
Image1.Picture = LoadPicture("")  
mnuCloseItem.Enabled = False
```

4. In the **Object** drop-down list box, click **mnuExitItem**, and then type **End** in the event procedure.

5. On the toolbar, click **Save Project** to save your changes.

► **To run the program**

1. On the Visual Basic toolbar, click **Start**.

Visual Basic loads the program and the form with its **File** menu.

2. From the **File** menu, click **Open**.

3. When the **Open** dialog box appears, load a picture file from your computer.

The picture selected should appear correctly sized in your image object.

4. From the **File** menu, click **Close**.

Your program should clear the picture file and turn off the **Close** command.

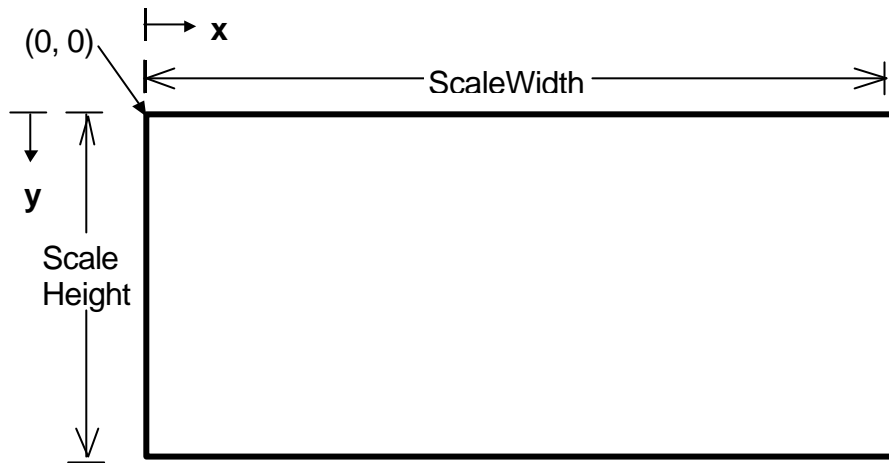
5. Try using the access keys and the shortcut keys to run the **File** menu commands. When you're finished, click the **File** menu **Exit** command.

Graphics Techniques with Visual Basic

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

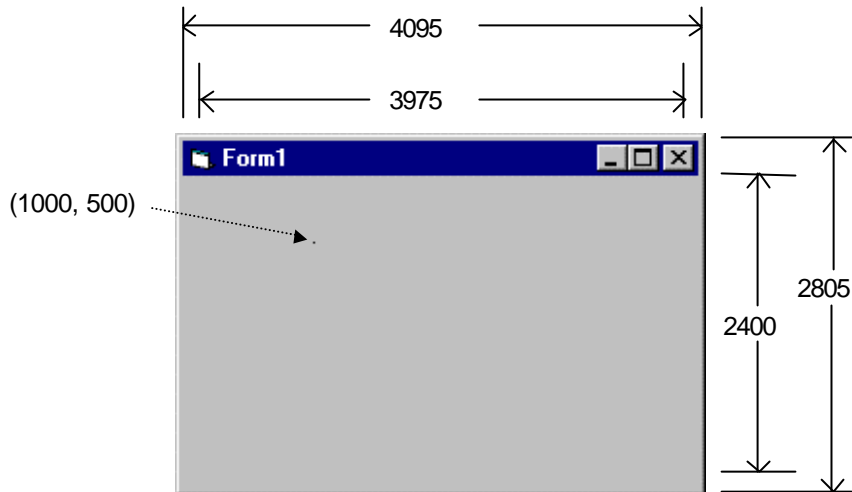
where **ObjectName** is the object name, **(x, y)** is the selected point, and **Color** is the point color (discussed in the next section). If the ObjectName is omitted, the current form is assumed to be the object. If Color is omitted, the object's **ForeColor** property establishes the color. PSet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

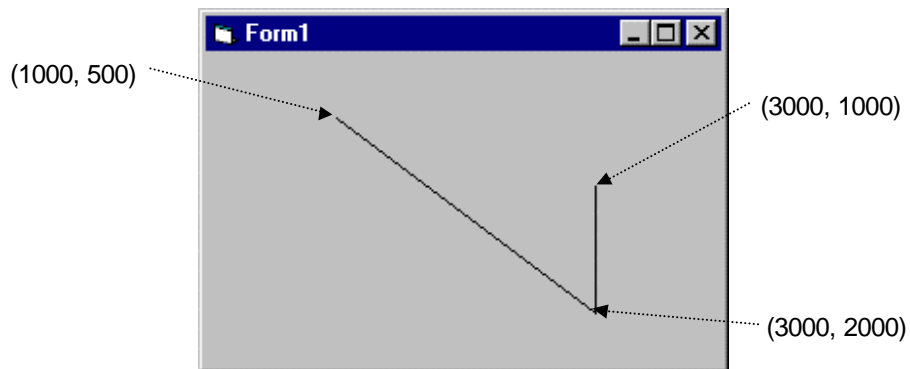
```
ObjectName.Line (x1, y1) - (x2, y2), Color, BF
```

- Line Method Examples:

Using the previous example form, the commands:

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

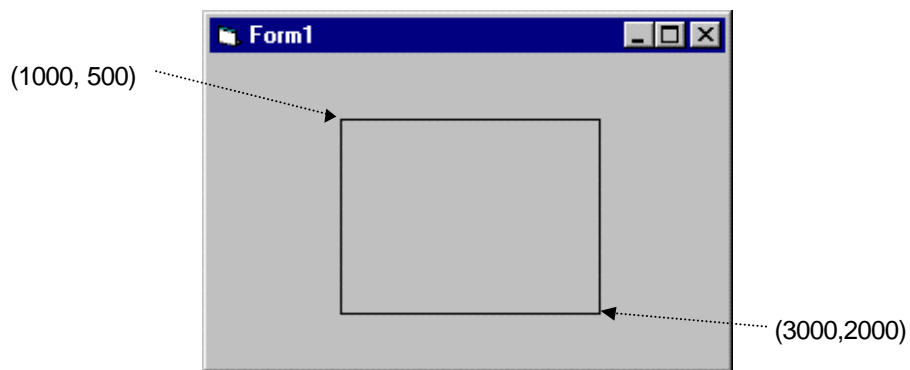
draws these line segments:



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
```

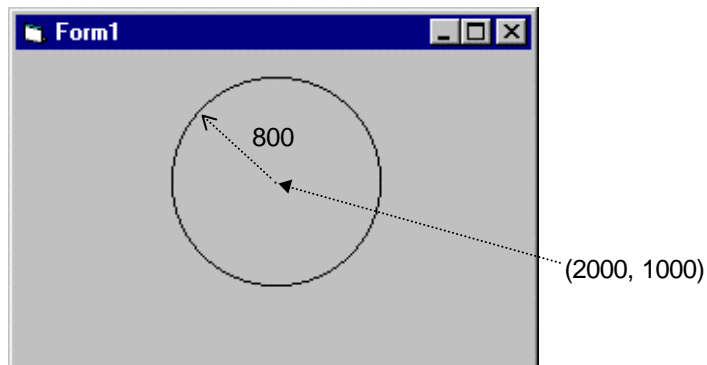
This command will draw a circle with center **(x, y)** and radius **r**, using **Color**.

- 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.

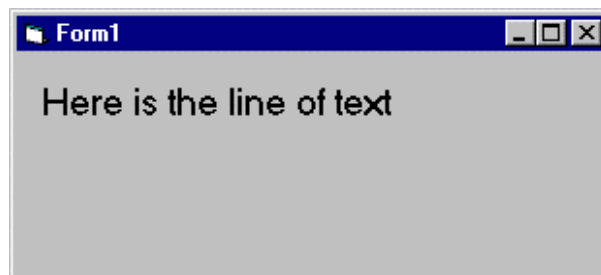
Information will print beginning at the object's **CurrentX** and **CurrentY** value. The color used is specified by the object's **ForeColor** property and the font is specified by the object's **Font** characteristics.

-
- 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.

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	0xFFFFFFFF	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 **DbClick** 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.

Example(1)

```
Private Sub Form_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
    Form1.BackColor = vbRed
End Sub
```

```
Private Sub Form_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    Form1.BackColor = vbBlue
End Sub
```

- **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(2)

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
Form1.Cls
Line (100, 100)-(X, Y)
End Sub
```

Example(3)

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
Line (0, 0)-(X, Y)
End Sub
```

Example(4)

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
Dim i
For i = 1 To 100
Circle (1000 + X, 1000 + Y), 500, vbRed
Cls
Next i
End Sub
```

Example(5)

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
Dim i
For i = 1 To 100
Circle (2000, 2000), 100 + i, vbRed
Next i
Circle (2000 + X, 2000 + Y), 100, vbRed
Cls
End Sub
```

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.

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.

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 until 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 priori, 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.

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
```

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:

$$I = \text{Int}((\text{Imax} - \text{Imin} + 1) * \text{Rnd}) + \text{Imin}$$

- 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

```

SOUNDS AND MULTIMEDIA

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
```

ξ. 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.

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 - Multimedia 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

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.

ADVANCE KEYS , MASHED EDIT CONTROL, CHART , RICH TEXT BOX AND SLIDER

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

- 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.

SelfFontName Set the font name for the selected text.

SelfFontSize Set the font size for the selected text.

SelfFontColor 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.

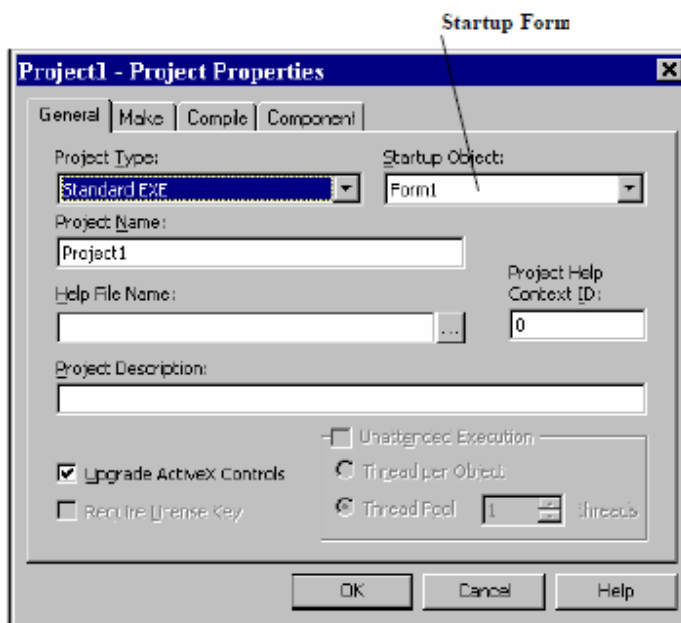
SelLength Length of select range of values.

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



- 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

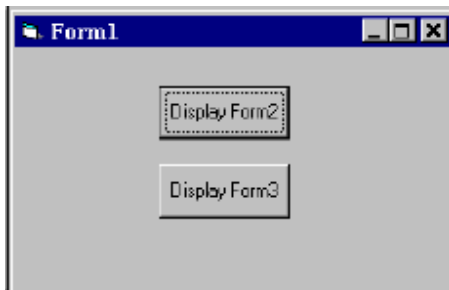
Unload Form1 ‘ Unloads Form1 from memory and hides it.

- 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:

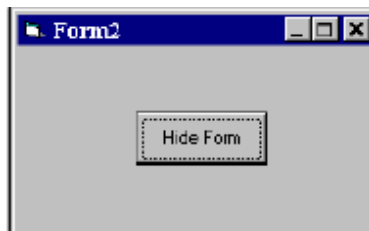
Start a new application. Put two command buttons on the form (**Form1**). Set one’s **Caption** to **Display Form2** and the other’s **Caption** to **Display Form3**. The form will look like this



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.