

An Introduction to Visual Basic 6.0

Section 1: The nature of the language

First we'll cover the overall design of Visual Basic 6.0, including (1) What Visual Basic does, then (2) how to start and stop Visual Basic, (3) when to use the different Visual Basic windows; and (4) how the VB programming environment windows work together.

Microsoft Visual Basic 6.0 is the most modern form of the old BASIC language. Visual Basic (or VB) lets you write, edit, and test Windows applications. Visual Basic is itself a Windows application: you load and execute the VB system just as you do other Windows programs. You can use this running VB program to create other programs. So, VB is a tool that programmers can use to write, test, and run Windows applications.

First, some important terms: a "program" is a set of instructions that make the computer do something useful such as perform analysis of a set of laboratory data. (The term program is often used synonymously with "application".) "Controls" are tools in the so-called "Toolbox" window that you place on a "form" to interact with the user and control the program flow. And, a "project" is a collection of files you create that make up your Windows application. An "application" is a collection of one or more files that compile into an executable program. Although programmers often use the terms "program" and "application" interchangeably, the term *application* seems to fit the best when you're describing a Windows program because a Windows program typically consists of several files. These files work together in the form of a project. The project generates the final program that the user loads and runs from Windows by double-clicking an icon or by starting the application with the Windows Start menu.

Programming tools have evolved substantially over the past 45 years along with computer hardware. A modern programming language like Visual Basic differs greatly from programming languages of just a few years ago. The visual nature of the Windows operating system requires more advanced tools than were previously available. Before window-based environments, a programming language was a simple text-based tool with which you wrote programs. These days we need more than just a language; we need a graphical development tool that can work inside the Windows system and create applications that take advantage of all the graphical, multimedia, and online activities that Windows offers. Visual Basic is such a tool. More than a language, Visual Basic lets us generate applications that interact with every aspect of the Windows operating system.

Although Visual Basic is a comprehensive programming tool, VB has retained its BASIC language heritage. In the mid-60's, Dartmouth's Professors Kemeny and Kurtz developed the BASIC programming language for beginning programmers. BASIC was easier to use than other programming languages of the time, such as FORTRAN. Microsoft has stayed true to the origins of BASIC when they developed Visual Basic. New programmers can learn to create simple but working Windows programs in a relatively brief time.

Here are a few more terms to understand in your use of Visual Basic: "Wizards" are question-and-answer dialog boxes within VB that automate some of the tasks of creating a program. A "compiler" is a system that converts the program you write into a computer executable application. (Important note: you may wish to compile programs you write to be turned in as class assignments just for your own amusement, but compiled programs should not be turned in as class assignments. Turn in the VB projects with .vdp, .vbw and .frm extensions so that your source code can be examined.) The "Developer Studio" is Visual Basic's development environment, providing, among other resources, useful "help" information about using VB. "Code" is another name for the programming statements you write.

Visual Basic 6 contains a true compiler that creates standalone runtime “.exe” files that execute more quickly than uncompiled (line by line interpreted) VB programs. VB also includes several wizards that offer step-by-step dialog box questions that guide you through the creation of applications. VB's development platform, a development environment called the Developer Studio, supports the same features for VB as the advanced language compilers offer for Visual C++ and Visual J++. Once you learn one of Microsoft's Visual programming products, you will have some of the skills needed to use the other programming languages without as long a learning curve.

Languages: Programming languages today are not what they used to be. The language itself has not gotten less important; rather, the graphical interfaces to applications have gotten more important. Computers, of course, cannot (yet) understand natural (spoken) language. A spoken language like English is simply too ambiguous for computers to understand. For this reason, we must adapt to the machine and learn a language that the computer can understand. VB's programming language is fairly simple and uses common English words and phrases for the most part. The language is not ambiguous, however. When you write a statement in the Visual Basic language, the statement never has multiple meanings within the same context, like natural languages do.

As you learn more of Visual Basic language's vocabulary and syntax (grammar, punctuation, and spelling rules), you will use the VB language to embed instructions within applications you create. All the code you enter must work together to instruct the computer what to do. Code is the glue that ties all the graphics, text, and processes together within an application. The program code lets the application know what to do given a wide variety of possible outcomes and user actions.

Visual Basic software is sold in several versions: Visual Basic 6 comes in a Standard Edition, a Professional Edition, an Enterprise Edition, and a Working Edition. The Standard Edition is called the learning edition and provides the least expensive approach to using Visual Basic. The Standard Edition gives you a complete development environment programming language, and many of the same tools the other editions offer. If you use the Standard Edition, you have a very respectable programming tool. Some people develop only with the Standard Edition and never need anything else. Most programmers need only the Standard or Professional Edition. The Enterprise Edition provides a multitude of additional links and resources to other Microsoft software, while the Working edition (often included with VB texts on an accompanying CD-ROM) allows you to write, run and save VB programs but not compile them. Any of these versions will suffice for use in our Bio 595 class.

The VB Programming Process: If you were to use Visual Basic to create a professional applications software package, you'd follow these basic steps: (1) You start running Visual Basic. (2) Create a new application or load an existing application. When you create a new application, you might want to use Visual Basic's VB Application Wizard to write your program's initial shell, as we'll describe shortly. (3) Test your application with the debugging tools provided in Visual Basic. The debugging tools help you locate and eliminate program errors (bugs) that can appear despite your best efforts to keep them out. (A bug is a program error that you must correct – debug - before your program executes properly.) (4) Save your project, being certain that the (minimum of) three files are saved on your own hard drive (on your home machine) or backup floppy disk (in the University Laboratory, LS126) as well as on the disk you will turn in to class. (5). Quit Visual Basic. These five steps are not necessarily sequential steps, but stages that you go through and return to before completing your application. Again, for our purposes, your exercises (programs) will not require compilation.

Starting Visual Basic: You start Visual Basic from the Windows "Start" menu. The Visual Basic development environment itself usually appears on a submenu called Microsoft Visual Basic 6.0, although yours might be called something different due to installation differences. You will see additional programs listed on the Microsoft Visual Basic 6.0 submenu, but when you select Visual Basic 6.0 from the submenu, Visual Basic loads and appears on your screen. On most systems,

the “dialog box” appears as soon as you start Visual Basic. The dialog box lets you start the VB Application Wizard, edit an existing VB project, or select from a list of recent projects you've worked on, depending on the dialog box tab you click.

Once you close the dialog box, the regular Visual Basic screen appears. VB's opening screen is pretty full of windows. You are now in the Visual Basic development environment. From this development environment you will create Windows programs. Although the screen can look confusing, you can fully customize the Visual Basic screen to suit your own needs and preferences. Over time, you will adjust the screen's window sizes and hide and display certain windows so that your Visual Basic screen's start-up state will differ from that of someone else's screen.

Most of VB's windows are sizable and dockable, meaning that you can connect them together, move them, and hide them. (A dockable window is a window that you can resize and move to the sides of the screen and connect to other windows.) This first lesson in using VB might be titled "Mastering the Development Environment" because it is here that we explain the parts of the environment and how to maneuver within that environment.

Stopping Visual Basic: You can exit from Visual Basic and return to Windows the same way you exit most Windows applications: Select File / Exit, click Visual Basic's main window close button, press Alt+F4, or double-click VB's Control menu icon that appears in the upper-left corner of the screen. If you have made changes to one or more files within the currently open project (remember that a project is the collection of files that comprise your application), Visual Basic gives you one last chance to save your work before quitting to Windows. Important: never power off the computer without completely exiting Visual Basic, or you might lose your work for your current session.

Mastering the Development Environment: Learning the ins and outs of the development environment before you learn Visual Basic is somewhat like learning the parts of an automobile before you learn to drive; you might have a tendency to skip the terms and jump into the foray. If, however, you take the time to learn some of the development environment's fundamental principles, you will be better able to master Visual Basic. The Visual Basic development environment contains several important screen components. Visual Basic looks somewhat like other Windows programs on the market. Many of Visual Basic's menu bar commands work just as they do in other applications such as Word or Excel. For example, you can select Edit /Cut and Edit/Paste or cut and paste text from one location to another. These same menu bar commands appear on almost every other Windows program on the market.

Standards -The Menu Bar and Toolbar: Visual Basic's menu bar and toolbars work just as you would expect them to. You can click or press a menu bar option's hotkey (for example, Alt+F displays the File menu) to see a pull-down list of menu options that provides either commands, another level of menus, or dialog boxes. Many of the menu options have shortcut keys (often called accelerator keys) such as Ctrl+S for the File/Save option. When you press an accelerator key, you don't first have to display the menu to access the option.

The toolbar provides one-button access to many common menu commands. Instead of selecting Edit/Paste, for example, you could click the Paste toolbar button. As with most of today's Windows applications, Visual Basic supports a wide range of toolbars. Select View/Toolbars to see a list of available toolbars. Each one that is currently showing on the screen will appear with a checkmark by its name.

As you begin to work with Visual Basic, pay attention to the form location and form size coordinates at the left of the toolbar buttons. These measurements, given in “twips” (a twip is 1,440th of an inch, the smallest screen measurement you can adjust), determine where the Form window appears, and what its size will be. Twip values usually appear in pairs. The first location value describes the x-coordinate (the number of twips from the left of the screen) and the second value describes the y-coordinate (the number of twips from the top of the screen), with 0,0 indicating the upper-left

corner of the screen. The first size value describes the width of the form, and the second size value describes the height of the form. Therefore, the size coordinate pair 1000,3000 indicates that the Form window will be 1,000 twips wide and 3,000 twips tall when the program runs. As you'll learn in the next section, the Form window is the primary user screen interface window for the applications you write. The location and size coordinates describe the form's location and size when you run the application.

The Form Window: This window is your primary work area. Although the Form window first appears small relative to the rest of your screen, the Form window comprises the background of your application. In other words, if you write a Windows-based data analyzer with Visual Basic, the analyzer's buttons all reside on the Form window and when someone runs the program; the analyzer program that appears is really just the application's Form window with components placed there and tied together with code.

You will not see program code on the Form window. The Form window holds the program's interactive objects, such as command buttons, labels, text boxes, scrollbars, and other controls. The code appears elsewhere in a special window called the Code window. You can select View/Code to see the Code window. A Code window is little more than a text editor with which you write the programming statements that tie together the application. When a program is running, the window might show a simple dialog box with a few options, text boxes, and command buttons.

The programmer who creates a program's starting dialog box does so by opening a Form window, adding some controls (the items on the Form window that interact with the user—sometimes called tools), and tying the components together with some Visual Basic language code. That's exactly what you will do when writing either simple or complex Visual Basic applications. You will begin with a blank Form window and add controls to the Form window such as options and command buttons. Some applications may even require multiple Form windows.

Word, for example, allows for several Form windows in a special mode called MDI (multiple-document interface) in which you can open multiple data documents within the same application. An application that requires only a single data window is called an SDI (single document interface) application, such as the Windows Notepad application that lets the user open only one data document at a time. SDI applications might support multiple forms; however, these forms do not hold multiple data files but only provide extended support for extra dialog boxes and secondary work screens.

An application will have a different appearance in its “design-time” state than it does in its “run-time” state. It is during design time that you design, create, edit, and correct the application. When you or another user runs the application, the results of your work can be seen in the application's “run-time” state.

The “program” you will produce will consist of code, forms, menus, graphics, and help files that you create and edit to form the project (also called source code). The parts of the application that you create, such as the forms, the code, and the graphics that you prepare for output, comprise the “source” program. When you compile or run the source program, VB translates the program into an “executable” or “object” program. You cannot make changes directly to an executable program. If you see bugs when you run the program, you must change the source application and rerun or recompile the source. (Again, you generally will not be compiling your programs.)

The Toolbox: The toolbox contains the controls that you place on the Form window. All of the controls used in a typical application you might write already appear in the toolbox. You will learn shortly how to place toolbox controls on the Form window. The toolbox never runs out of controls; if you place a command button on the Form window, another awaits you on the toolbox, ready to be placed also.

The tools that appear in the standard Toolbox window are called the “intrinsic” controls because all four versions of VB support these standard tools. You can add additional controls to the toolbox as your needs grow. Some extra tools come with all editions of VB, but these extra tools do not appear on the Toolbox window until you add them through the Project/Components menu option. The controls on the left half of the dialog box, from the top down, include: pointer, label, frame, check box, combo box, horizontal scrollbar, timer, directory list box, shape, image, and OLE. The controls on the right half of the dialog box, from the top down, include: picture box, text box, command button, option button, list box, vertical scrollbar, drive list box, file list box, line, and data. More about the use of these controls later.

The Form Layout Window: This window displays the initial position and relative size of the current form shown in the Form window. An application may be a multiple-form application; a form with the title “Text Box Properties” is one type of several forms. The Form Layout window always shows where the form appears in the current Form window. If you want the form to appear at a different location from the current position, you can move the form inside the Form Layout window to move the form's location when the application is run. Notice that the form location indicators, to the right of the toolbar buttons, change when you move the form in the Form layout window. You can display the Form Layout window from the View menu, and you can hide the Form Layout window by clicking its window close button.

The Project Explorer Window: this window, often called the Project window, gives you a tree-structured view of all the files in the application you are working on. The Project Explorer window displays forms, modules (files that hold supporting code for the application), classes (advanced modules), and more. When you want to work with a particular part of the loaded application, double-click the component in the Project window to bring that component into focus. In other words, if the Project Explorer window displays three forms and you need to edit one of the forms, locate and double-click the form name in the Project window to make that form appear in the Form window.

The Properties Window: “Properties” are detailed descriptive information about a control. A different list appears in the Properties window every time you click over a different Form window tool. The Properties window describes properties (descriptive and functional information) about the form and its controls. Many properties exist for almost every object in Visual Basic. The Properties window lists all the properties of the Form window's selected control.

Visual Basic's on-line help system is quite advanced. When you want help with a control, window, tool, or command, press F1. Visual Basic analyzes what you are doing and offers help. In addition, Visual Basic supports a tremendous help resource called Books Online. (Books Online are electronic books about Visual Basic for the Visual Basic programmer.) When you select Books Online from the Help menu, Visual Basic displays a tree-structured view of books about Visual Basic that you can search and read. The online help extends to the Internet as well. If you have an Internet connection, you can also browse the latest help topics by selecting Help/Microsoft on the Web. (However, please do not use the personal copy of VB distributed on disk or from the website to the class; although a legal copy, it is not licensed for multiple user distribution and Microsoft on-line access. Instead, if you wish to access on-line resources, please do so from the LS-126 computer laboratory or Love Library stations.

Summary: Visual Basic is more than it first appears. Programmers can use Visual Basic to create advanced Windows applications. We will now look at a sample application so you can get a better understanding of how Visual Basic's components work together. Visual Basic is more than just a programming language. Nevertheless, learning VB's language portion is integral to writing advanced applications. Fortunately, the Visual Basic programming language is one of the easiest programming languages in existence. The language is simple but powerful because Visual Basic's language was based upon early BASIC, a beginner's language. VB's simplicity does not translate to

a lack of capability, however. VB is one of the most powerful Windows programming languages on the market and supports advanced programming techniques.

Section 2: Analyzing a sample program

This section describes what “events” are, how to respond to events, when to use event procedures, how to name event procedures; and when and how to use the VB Application Wizard.

VB programs, object-based as this language is, are “event-driven” programs. Programs usually offer lots of choices of what to do next at any moment during program execution: menu options, various controls, data-entry, etc. “Events” can include a keyboard key-press, mouse movement, etc. A VB program consists of a group of routines to handle events like these; when the program runs, specific event procedures execute only if that particular event actually occurs. Every control placed on a form window supports one or more events. The written code “under” that object executes as soon as the event occurs.

Analyzing a sample application: Load and operate the sample application provided on the website, "Example1", a simple application program for calculating the average value of a laboratory measurement taken five times. Note that this "project" consists of several files, which are shown in the Project window. Load the project, by double-clicking "Example1.vbp". Open the form, "Example1.frm". Clicking on plus signs in the Project window will expand the listing of contents of that folder. The " Example1.frm " name to the form was assigned by the program's author, using a recommended convention in writing VB projects: a three letter abbreviation for the type of object, in this case, a form with objects placed on it: this is the graphical user interface created for the program. Note the parenthesized name which indicates the actual Windows pathname for this file, "Form1"; the file's name within the project is " Example1." VB will assign internal names to objects you incorporate in the application program that you write: default names like "Command1" and "Label2" are assigned automatically by VB and can then be renamed by you with a name suggesting the object's purpose, like "cmdEnterdata" and lblChangevalue;" note the correspondence between these two example object types and their three letter prefixes. In this first simple example, however, the default names of objects placed on the form window have been retained. The six textboxes are named Text1 through Text6; the seven labels are called Label1 through Label7. There are two command buttons Command1 (with text reading “Find Result”) and Command2 (“Exit Program”)

Running an application: You can execute this demonstration program by running the application (use the F5 key or the pulldown “Run” menu from the command bar). Enter sample values such as 1, 2, 3, 4 and 5 for the five measured values (you can use the tab key to move from textbox to textbox); when you click the “Find Result” button, a value should appear in the box (Text6) next to “The average value is:” label. After you have exited the program by clicking on “Exit Program” to return to the Design Time state, select one of the controls, say, a button, by clicking on it once, to make that object's property list appear in the Properties window (usually in the lower right area of the screen).

The behind-the-scenes program code for this project can be found in the Code window. This window behaves like a simple text editor, with standard functions like insert, delete, copy, cut and paste mouse-selected text. You can look at the code executed when a particular object is selected and double-clicked during Design Time. A cursory examination of the code accompanying an object like a command button are "wrapper lines" of code - the first and last lines of code. Wrapper lines are generated when a new object is placed on a form of an application project you are writing. Typical event wrapper lines are "Private Sub controlName_eventName()", ending with "End Sub"; the specific code you enter is to be sandwiched between these two lines.

Code: Event procedures (code specifying how an object is to respond to being activated) always consist of the control name, an underscore, and the procedure's event name. Standard events are

named by VB conventions. The top of a code window has a drop-down menu with all possible events listed.

When an object is placed on a form during program creation, properties are set at default values automatically. You can customize and tailor each object by clicking in the appropriate test areas of the Properties window. It is important to note that the event procedure code written for an object doesn't actually do anything during Design Time; it is executed only during Run Time when the object is selected.

You will get more detailed information about the objects (labels, textboxes and command buttons) in later sections of this document, but for now you can examine some of the features of this simple biosciences laboratory program by selecting objects on the form window and examining their properties. Select the "Exit Program" button (in Design Time mode, not while the program is running) to see how simple the coding of a command button event can be: this one simply quits the execution of the program and returns Visual Basic to its Design Time mode, using the BASIC keyword "End". More interesting is the code behind the "Find result" button. Here is a simple calculation, to find the average of the five values entered in textboxes 1 through 5. There are various ways the code to perform the averaging could have been written. The code shown here is long-winded, hopefully to improve clarity of the operation. We have created 5 variables, X1 through X5, and set each, in turn, equal to the text entered in Textboxes 1 through 5. However, the text in a textbox is not a number; the function VAL for each of these entries converts each text into a numeric value that VB can manipulate in a numeric calculation. The code line performing the calculation is written like a simple math formula: The variable average X is set equal to the sum of X1 through X5, divided by 5. The final line of code places this numeric value into the text property of textbox 6 ("Text6.text")

Generating a program from scratch: now try using the VB Application Wizard to generate a skeleton (or "shell") of a program by responding to a series of dialog boxes. Select File/New Project. Double-click the Application Wizard icon. Answer the series of questions asked by the Wizard; use default (computer-assigned) values. The Wizard allows interconnectivity with the Internet or with data from application programs like Excel. Complete the dialog and examine the program it has generated. Try the controls and menus. (This program doesn't do much, but at least it doesn't "blow up.")

Another demonstration program: Also, check out the sample application called Vcr/ Vcr.vbp. Note that a project created for you in VB can contain only one, or might contain multiple files. In this example, double-click form called frmVCR in Project Explorer window to bring up VCR in Form window. Click the plus signs in the Project window to get a full tree list of the whole project. The project window shows external disk drive filenames for each project. In this much more complex sample program, the Form name is frmVCR; but the disk file name is vcr.frm. The names of objects must begin with an alphabetical character. Double-click frmSetTime in the example. To run this program: use function key F5. Normally, when you author an application, you'll usually choose to leave the form layout window closed.

Note: Compare the "toolbox" (which lets you select controls to place on a form) and the "toolbar" (icon-based menu selections linked to menu bar above it). Click (select) a button on the form to see its properties in Properties window, toolbar (row of menu option buttons under menu bar). Properties of a button include the name, a caption, a font, a height, etc.

As we proceed to write programs, we'll write code stored in standard modules that can be recycled in later programs. Look at code in the example's VCRmodule (note the extension, ".bas"). Also, look at the code for the "Up" command button by double-clicking that button.

Section 3: Controls and Properties

This section covers (1) how to create an application from scratch, (2) how to place and size the controls, (3) how object properties are set, (4) how to name objects (what prefixes to use), and (5) how to use “tooltips”.

Creating a new application: select the “.EXE” option. You can place controls on the form window either by double-clicking a selected control to put it in the center of the forms window where you can drag it to any desired location, or clicking the control, then clicking the crosshair cursor where you want the control located on the form. You can size the control by dragging one of its eight “sizing handles.” You can move multiple commands by selecting them using the CTRL key, or drawing a rectangle around them.

Creating your application program's interface: Set the control properties; select a control and click in the Properties window, or use the drop-down window box. Scroll the Properties window. Note that the form itself has its own properties. Use an appropriate 3-letter prefix in naming the controls. Here are the standard prefixes recommended:

cbo	Combo box	lbl	Label
chk	Check box	lin	Line
cmd	Command button	lst	List box
dir	Directory list box	mnu	Menu
drv	Drive list box	ole	OLE client
fil	File list box	opt	Option button
fra	Frame	pic	Picture box
frm	Form	shp	Shape
grd	Grid	tmr	Timer
hsb	Horizontal scrollbar	txt	Text box
img	Image	vsb	Vertical scrollbar

Give your user some built-in "help": “Tooltips” are helpful little clues about the function of a control entered as text in the control’s “ToolTipText” property. The tip appears as the mouse is run over the button. Many properties are easiest to specify from the dropdown list box (as when choices are restricted to True or False states). Ellipses (...) following choices indicate that a dialog box will be displayed if this choice is selected. Properties can be listed alphabetically (default) or grouped by category (like Appearance, Behavior, etc.). A “named literal” or constant is a label for a fixed value used in your program.

To create your first application from scratch (without the Wizard's help): (1)create a new project (File | New Project, .EXE option); (2) change the form’s “Name” property to “frmFirst” and the caption to “My First Application” (this shows up as the title bar); (3) expand the form to 7380 h by 7095 w (in “twips”, the units of distance on the screen); (4) select the Label control; (5) drag a Label control centered toward the top of the Form window; (6) change the label’s Name property to “lblFirst” and its Caption to “VB is fun” or “Hello, World!” or “Bio 595 Rules!”; (7) set the font size (Font property) to 24 Bold, and Alignment to 2-Center; (8) set BorderStyle property to FixedSingle; resize the height of the label if necessary; (9) add a Command button; (10) set the button’s Name to cmdExit; note that the displayed caption has the “x” underlined, permitting the Alt+x keyboard command as a way to activate the button (in addition to clicking on it with the mouse); (11) now write a command procedure for the command button; double-click the button to open its Code window. The window will show default “wrapper” lines (first and last lines of code) as follows:

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

Enter the word “End” between these “wrapper lines”. As we mentioned earlier, this statement terminates Basic program execution, so your application will quit when this Event Handler is executed (when the button is clicked). (12) Save your program on a floppy disk if you are running these exercises on one of the LS126 computers; you are entitled to save your programs anywhere you wish if you are running these exercises on your own computer! (13) Run the program!

Section 4: Examining Labels, Buttons and Text Boxes

In this section we explain in more detail the three types of objects used in our first example – labels, textboxes, and command buttons – as well as the “form” – the user interface upon which these objects are placed. Specifically, we cover (1) how to set the “focus” order of objects like command buttons or textboxes; (2) when the “cancel” property triggers events; (3) how to set a command button’s default property; (4) what important common properties are; and, (5) how to adjust label sizes.

Control focus: (this term refers to the control that is active at any moment of program execution) The window or form currently highlighted (blue title bar) has the focus; the control with highlighted caption or outline has the focus. This determines what control the next keystroke will activate. Mouse and hotkeys need no focus (you can change the focus just by selecting another button or window). The “Enter” or hotkey keys act on only the active window, even if other windows can recognize and act on these events.

A command button’s “cancel” property determines which command button gets a simulated “click” event when the “Esc” key (escape) is struck. A command button’s “default” property; the button with default set to “True” gets the first Enter. Only one button can have this setting of True at a time. Tab and Shift+Tab allow the user to move the focus from control to control when the program is running. The tab order is determined initially by the order in which commands are created on the form; the order can be rearranged by changing the order of the “TabIndex” property.

Command buttons: to add a button to an application, locate and size the button; set its name and caption; then add code to the button’s Click event procedure. The click event is the one (out of a potential 15 different events) usually accommodated. All properties can be set at design time; some of these can be set at runtime. The common command button properties are:

BackColor	Specifies the command button's background color. Click the BackColor's palette down arrow to see a list of colors and click Categorized to see a list of common Windows control colors. Before the command button displays the background color, you must change the style property from 0-Standard to 1-Graphical.
Cancel	Determines whether the command button gets a Click event if the user presses the esc key.
Caption	Holds the text that appears on the command button.
Default	Determines if the command button responds to an Enter keypress even if another control has the focus.
Enabled	Determines whether the command button is active. You can change the Enabled property at runtime with code when a command button is no longer needed and you want to gray out the command button.
Font	Produces a Font dialog box in which you can set the caption's font name, style, and size.
Height	Holds the height of the command button in twips.
Left	Holds the number of twips from the command button's left edge to the Form window's left edge.
MousePointer	Determines the shape of the mouse cursor when the user moves the mouse over the command button.
Picture	Holds the name of an icon graphic image that appears on the command button as long as the Style property is set to 1-Graphical.
Style	Determines whether the command button appears as a standard Windows command button (if set to 0 -standard) or a command button with a color and possible picture (if set to 1 -Graphical).
TabIndex	Specifies the order of the command button in the focus order.

TabStop	Determines whether the command button can receive the focus.
ToolTipText	Holds the text that appears as a tooltip at runtime.
Top	Holds the number of twips from the command button's top edge to the Form window's top edge.
Visible	Determines whether the command button appears or is hidden from the user. (Invisible controls cannot receive the focus until the running code changes the Visible property to True.)
Width	Holds the width of the command button in twips.

Labels: these objects contain the text displayed on a form. If a label is given text too long (number of characters) or too large (font size) to fit within the label boundaries, the text may be truncated or the text field may move downwards or to the right, depending on property settings. Set `AutoSize` and `WordWrap` properties to `True` to account for this possibility. You don't need to worry about this if the label isn't going to change during program execution. Here are common label properties:

Alignment	Determines whether the label's caption appears left-justified, centered, or right-justified within the label's boundaries.
AutoSize	Enlarges the label's size properties, when <code>True</code> , if you assign a caption that is too large to fit in the current label's boundaries at runtime.
BackColor	Specifies the label's background color. Click the <code>BackColor</code> 's palette down arrow to see a list of colors and click <code>Categorized</code> to see a list of common Windows control colors.
BackStyle	Determines whether the background shows through the label or if the label covers up its background text, graphics, and color.
BorderStyle	Determines whether a single-line border appears around the label.
Caption	Holds the text that appears on the label.
Enabled	Determines whether the label is active. Often, you'll change the <code>Enabled</code> property at runtime with code when a label is no longer needed.
Font	Produces a <code>Font</code> dialog box in which you can set the caption's font name, style, and size.
ForeColor	Holds the color of the label's text.
Height	Holds the height of the label's outline in twips.
Left	Holds the number of twips from the label's left edge to the Form window's left edge.
MousePointer	Determines the shape of the mouse cursor when the user moves the mouse over the label.
TabIndex	Specifies the order of the label in the focus order. Although the label cannot receive the direct focus, the label can be part of the focus order.
ToolTipText	Holds the text that appears as a tooltip at runtime.
Top	Holds the number of twips from the label's top edge to the Form window's top edge.
Visible	Determines whether the label appears or is hidden from the user.
Width	Holds the width of the label in twips.
WordWrap	Determines whether the label expands to fit whatever text appears in the caption.

Text boxes: these objects receive user keyboard input. Note that text boxes don't have a "Caption" property. (Text can be included in the box before the program runs, so captions aren't needed.) Common text box properties are:

Alignment	Determines whether the text box's text appears left-justified, centered, or right-justified within the text box's boundaries.
BackColor	Specifies the text box's background color. Click the <code>BackColor</code> property's palette down arrow to see a list of colors and click <code>Categorized</code> to see a list of common Windows control colors.

BorderStyle	Determines whether a single-line border appears around the text box.
Enabled	Determines whether the text box is active. You can change the Enabled property at runtime with code when a text box is no longer needed.
Font	Produces a Font dialog box in which you can set the Text property's font name, style, and size.
ForeColor	Holds the color of the text box's text.
Height	Holds the height of the text box's outline in twips.
Left	Holds the number of twips from the text box's left edge to the Form window's left edge.
Locked	Determines whether the user can edit the text inside the text box that appears.
MaxLength	Specifies the number of characters the user can type into the text box.
MousePointer	Determines the shape of the mouse cursor when the user moves the mouse over the text box.
MultiLine	Lets the text box hold multiple lines of text or sets the text box to hold only a single line of text. Add scrollbars if you wish to put text in a multiline text box so a user can scroll through the text.
PasswordChar	Determines the character that appears in the text box when the user enters a password (keeps prying eyes from knowing what the user enters into a text box).
ScrollBars	Determines whether scrollbars appear on the edges of a multiline text box.
TabIndex	Specifies the order of the text box in the focus order.
TabStop	Determines whether the text box can receive the focus.
Text	Holds the value of the text inside the text box. The Text property changes at runtime as the user types text into the text box. If you set an initial Text property value, that value becomes the default value that appears in the text box when the user first sees the text box.
ToolTipText	Holds the text that appears as a tooltip at runtime.
Top	Holds the number of twips from the text box's top edge to the Form window's top edge.
Visible	Determines whether the text box appears or is hidden from the user.
Width	Holds the width of the text box in twips.

Form properties: the form is the “stage” upon which your program runs. The important form properties are as follows:

BackColor	Specifies the form's background color. Click the BackColor's palette down arrow to see a list of colors and click Categorized to see a list of common Windows control colors.
BorderStyle	Determines how the Form window appears. The BorderStyle property specifies whether the user can resize the form and also determines the kind of form you wish to display.
Caption	Displays text on the form's title bar at runtime.
ControlBox	Determines whether the form appears with the Control menu icon. The Control menu appears when your application's user clicks the Control menu icon.
Enabled	Determines whether the form is active. Often, you'll change the Enabled property at runtime with code when a form is no longer needed. Generally, only multiform applications, such as MIDI applications, need to modify a form's Enabled property.
Font	Produces a Font dialog box in which you can set the text's font name, style, and size.
ForeColor	Holds the color of the form's text.
Height	Holds the height of the form's outline in twips.

Icon	Describes the icon graphic image displayed on the taskbar when the user minimizes the form.
Left	Holds the number of twips from the form's left edge to the screen's left edge.
MaxButton	Specifies whether a maximize window button appears on the form.
MinButton	Specifies whether a minimize window button appears on the form.
MousePointer	Determines the shape of the mouse cursor when the user moves the mouse over the form.
Moveable	Specifies whether the user can move the form at runtime.
Picture	Determines a graphic image that appears on the form's background at runtime.
ScaleMode	Determines whether the form's measurements appear in twips, pixels (the smallest graphic dot image possible), inches, centimeters, or other measurements.
ShowInTaskbar	Determines whether the form appears on the Windows taskbar.
StartUpPosition	Determines the state (centered or default) of the form at application startup.
Top	Holds the number of twips from the form's top edge to the Form window's top edge.
Visible	Determines whether the form appears or is hidden from the user.
Width	Holds the width of the form in twips.
WindowState	Determines the initial state (minimized, maximized, or normal) in which the window appears at runtime.

Section 5: Getting Code into a Visual Basic Program

Next, we discuss (1) the data types supported by VB; (2) the declaration of variables; (3) the assignment of data to variables; (4) data type mismatch problems; and (5) the use of operators.

A VB program consists of forms, controls placed on the forms, and VB code. The code manipulates data and performs I/O (input and output operations). Some code consists of a lot of small event procedures in form modules and other code in standard (non-form related) modules. Sometimes a program might have no form (the form's visible = false).

Definitions: A "form module" is the module file containing one or more forms used in an application, and the code (e.g. event procedures) that goes with each form. A "standard module" is a file holding code not connected to a form. VB 5, this program's predecessor, was an SDI (single document interface) program, supporting only one open window at a time; VB 6 is an MDI (multiple document interface) application and can have (like Microsoft Word) more than one open window. Even an SDI application can have multiple forms.

Data types in VB: these include numeric, string (e.g. text), and special (e.g. logical). Most controls also permit "variant" data. Data can be converted from one type to another by various functions. Conversion can be done implicitly. Visual Basic Data Types include the following:

<u>Data Type</u>	<u>Description and Range</u>
Boolean	A data type that takes on one of two values only: True or False. True and False are Visual Basic reserved words, meaning that you cannot use them for names of items you create.
Byte	Positive numeric values without decimals that range from 0 to 255.
Currency	Data that holds dollar amounts from -\$922,337,203,685,477.5808 to \$922,337,203,685,477.5807. The four decimal places ensure that proper rounding can occur. VB follows the Windows International settings and adjusts currency amounts according to your country's requirements. Don't include the dollar sign when entering Currency values.

Date	Holds date and time values. The date can range from January 1100, to December 31, 9999.
Decimal	A new data type not yet supported in Visual Basic.. The Decimal data type represents numbers with 28 decimal places of accuracy.
Double	Numeric values that range from -1.79769313486232E+308 to 1.797693134QG232E+308. The double data type is often called double-precision.
Integer	Numeric values with no decimal point or fraction that range from -32,768 to 32,767.
Long	Integer values with a range beyond that of Integer data values. Long data values range from -2,147,483,648 to 2,147,483,647. Long data values consume more memory storage than integer values, and they are less efficient. The Long data type is often called long integer.
Object	A special data type that holds and references objects such as controls and forms.
Single	Numeric values that range from -3.402823E+38 to 3.402823E+38. The Single data type is often called single-precision.
String	Data that consists of 0 to 65,400 characters of alphanumeric data. Alphanumeric means that the data can be both alphabetic and numeric. String data values may also contain special characters such as ^, %' and @. Both fixed-length strings and variable-length strings exist.
Variant	Data of any data type and used for control and other values for which the data type is unknown.

Scientific notation: The letters E and D designate exponent and double precision exponent respectively. Literals are by nature constants. Strings are literals of text in quotation marks. ("" is an empty, or null string.) Time and date are embedded as literals between pound (#) signs. Use data type suffix characters to specify appropriate type: "&" for long, "!" for single, "#" for double, and "@" for currency.

Variables are named locations in computer memory that hold data. A variable must be declared, identifying the data type it is to hold. Use the "Dim" statement to declare variables. The format is *Dim VarName as DataType*. Use Option Explicit statement to tell VB you will declare variables and their data types (otherwise, VB assumes variant data type for variables). Dim in an event procedure means the variable is local. Dim in a general module makes the variable global. "Public" instead of Dim makes the variable(s) visible in every module of the project. It is recommended (but not required) that standard 3-letter prefixes be used in variable names, identifying the data type. These are:

Prefix	Data Type	Example
bln	Boolean	blnIsOverTime
byt	Byte	bytAge
cur	Currency	curHourlyPay
dtm	Date	dteFirstBegan
dbl	Double	dblMicroMeasurement
int	Integer	intCount
log	Long	lngStarDistance
obj	Object	objSoundClip
sng	Single	sngYearSales
str	String	strLastName
vnt or var	Variant	vntControlValue

Some examples: Dim IntLength As Integer, Dim sngPrice As Single, Dim dblStructure As Double, Dim strFirstName As String. To define a fixed length string, Dim strFirstName As String * 20. Dims can be grouped: Dim A As Integer, C As Integer.

Assignment statement: this puts data into a variable. The format is VarName = Expression. The expression here can be a literal, variable, or math expression. You must match the data type and declared variable type. For example, sngTemp = 42.1 is valid, but sngTemp = "Hot!" is not. Note that text boxes or command buttons have properties which, as variables, can be reset by code during program execution.

Math operators: + (add), - (subtract), * (multiply), / (divide), ^ (raise to power), & (concatenate strings). Precedence: first ^, then * and /, then + and -, from left to right. Use parentheses to avoid ambiguity. Example of a string concatenation: strFullName = strFirstName & " " & strLastName (the & " " inserts a space between the first and last names).

Section 6: Message and Input Boxes

This section covers runtime I/O (for input/output) interaction, that is, the data flow between the user and the computer. Topics covered here include (1) comparing the properties of message boxes and text boxes; (2) BASIC functions; (3) testing values returned by message boxes; (4) adding remarks; and (5) receiving input box messages.

Functions: these include intrinsic or built-in functions such as common math operations, string manipulations, and I/O operations. Functions accept arguments, and return a result for use by the program. Arguments are the data given to a function. The two functions we'll cover here first are MsgBox() and InputBox(). The presence of the parentheses indicates the name identifies a function, not a variable. A message box gives the user output information; while an input box asks the user for input information.

MsgBox(): assign these functions to an integer variable so that the button clicked by a user can be saved. The format is:

an IntVariable = MsgBox(strMsg [intType] [, strTitle])

Here, strMsg is a text string (either variable or constant), the message to be displayed; intType is the optional value describing desired options in the box; and strTitle is the Message box title- if omitted, the Project's name is used.

The buttons displayed in a message box:

<u>Named Literal</u>	<u>Value</u>	<u>Description</u>
vbOKOnly	0	Displays the OK button.
vbOKCancel	1	Displays the OK and Cancel buttons.
vbAbortRetryIgnore	2	Displays the Abort, Retry, and Ignore buttons.
vbYesNoCancel	3	Displays the Yes, No, and Cancel buttons.
vbYesNo	4	Displays the Yes and No buttons.
vbRetryCancel	5	Displays the Retry and Cancel buttons.

The icons displayed in a message box:

<u>Named Literal</u>	<u>Value</u>	<u>Description</u>
vbCritical	16	Displays Critical Message icon.
vbQuestion	32	Displays Warning Query icon.
vbExclamation	48	Displays Warning Message icon.
vbInformation	64	Displays Information Message icon.

The default buttons displayed in a message box:

<u>Named Literal</u>	<u>Value</u>	<u>Description</u>
----------------------	--------------	--------------------

vbDefaultButton1	0	The first button is the default.
vbDefaultButton2	256	The second button is the default.
vbDefaultButton3	512	The third button is the default.

The options selected use the intType value in the MsgBox() function, and determine whether an icon is displayed, and whether the message box is application specific or system specific. If application specific, the box must be answered before the program proceeds; if system specific, the box must be closed even to switch to another Windows application. Example of a MsgBox function: intPress = MsgBox("Are you ready for the report?", vbQuestion + vbYesNoCancel, "Report Request")

MsgBox () return values. These integers report back which button was clicked.

<u>Named Constant</u>	<u>Value</u>	<u>Description</u>
vbOK	1	The user clicked the OK button.
vbCancel	2	The user clicked the Cancel button.
vbAbort	3	The user clicked the Abort button.
vbRetry	4	The user clicked the Retry button.
vbIgnore	5	The user clicked the Ignore button.
vbYes	6	The user clicked the Yes button.
vbNo	7	The user clicked the No button.

When functions are entered in VB's Code Window, a pop-up help window immediately shown the function's format; Function argument choices are automatically shown in drop-down lists.

Comments: remarks added to a program constitute an essential part of the documentation that the program's author should provide to the user. Critical information to be provided include the program's author, date, what the general program is to accomplish, the intent or goal of each procedure, and any explanations deemed necessary to show difficult logical functioning of the program (the program's algorithm, or strategy for solving the problem). Begin remark lines with the Rem statement, or with an apostrophe. These can be added to the right of code lines on the same line.

InputBox testing: the format of this function is

```
strVariable = InputBox( strprompt [, (strTitle) [, strDefault] [, intXpos, intYpos]]])
```

Here strPrompt is inside the displayed box; strTitle is the title inside the input box's title bar, strDefault is a default string value VB displays for the default answer (which can be changed by the user). IntXpos and IntYpos give the input box location on the form, in twips from top and left edges; if omitted, box is centered on the form. Example:

```
strCompName = InputBox("What is the name of the Company?", "company request, "XYZ, Inc.")
```

Section 7: Comparison operators

This section explains (1) the comparison operators, (2) IF statements; (3) ELSE branching; and (4) SELECT CASE statements.

Comparison operators: these yield "true" or "false" results in comparing data values with one another. The comparison operators can compare variables, literals, control values, etc. Comparisons work either with numeric or alphabetic values, allowing comparisons of strings. Strings are equated to their ASCII code values. Thus, "B" > "C" (comparing ASCII codes 65 and 66) and "Smith, K." < "Smith, L." Comparisons must be of consistent data types or a mismatch error will be generated.

<u>Operator</u>	<u>Usage</u>	<u>Description</u>
>	IblLab.Caption > Goal	The <i>greater than</i> operator returns True if the value on the left side of > is numerically or alphabetically greater than the value on the right.
<	Bill < 2000.00	The <i>less than</i> operator returns True if the value on the left side of < is numerically or alphabetically less than the value on the right.
=	Age = Limit	The <i>equal to</i> operator (sometimes called the equal operator) returns True if the values on both sides of = are equal to each other.
>=	FirstName >= "Mike"	The <i>greater than or equal to</i> operator returns True if the value on the left side of >= is numerically or alphabetically greater than or equal to the value on the right.
<=	Num c= IblAmt.Caption	The <i>less than or equal to</i> operator returns True if the value on the left side of c= is numerically or alphabetically less than or equal to the value on the right.
<>	txtAns.Text <> "Yes"	The <i>not equal to</i> operator returns True if the value on the left side of <> is numerically or alphabetically unequal to the value on the right.

The IF statement: this uses the comparison operator results to test data. "IF" will execute subsequent lines of code if the result is true, or other lines of code if the result is false. The format is:

```
IF comparisonTest THEN
  One or more lines of VB code (a "block" of code)
ELSE
  One or more lines of different VB code
END IF
```

ELSE here is optional and is code executed if the evaluation result is False. Either the first block of code or the second executes (they're "mutually exclusive")

The logical operators: these allow compound comparisons.

<u>Operator</u>	<u>Usage</u>	<u>Description</u>
And	If (A > B) And (C < D)	Produces True if both sides of the And are true. Therefore, A must be greater than B and C must be less than D. Otherwise, the expression produces a false result.
Or	If (A > B) Or (C < D)	Produces True if either side of the Or is true. Therefore, A must be greater than B, or C must be less than D. If both sides the Or are false, the entire expression produces a false result.
Not	If Not (strAns = "Yes ")	Produces the opposite true or false result. Therefore, if strAns holds Yes ", the Not turns the true result to false.

SELECT CASE: this statement allows multiple choices to be made; it simplifies otherwise deeply nested If-Then-Else statements. Its format is

```
Select Case expression
  Case value
    One or more VB statements
  Case value
    One or more VB statements
  Case value
```

One or more VB statements etc.

End Select

Here *expression* must evaluate to a numeric or string value.

Other formats of "Select Case":

Select Case *expression*

Case is *relation*:

One or more VB statements

Case is *relation*:

One or more VB statements

Case Else:

One or more VB statements

End Select

Instead of comparing Expression values for an exact case match, this second format allows separate comparison tests. Still another option uses the format:

Select Case *expression*

Case expr1 to expr2:

One or more VB statements

Case expr3 to expr4:

One or more VB statements

Case expr5 to expr6:

One or more VB statements

End Select

Here the Case lines are given a range of values instead of one exact match.

Section 8: Looping

This section describes (1) loops, (2) how to code DO loops; (3) why there are multiple ways to loop; (4) the use of For...To...Next; and (5) How the EXIT statement interrupts program execution.

DO WHILE loop: Like If...Then..., Do While works with the six comparison expressions listed earlier.

The format is

Do While (comparison test)

Block of VB statements

Loop

The loop is repeated as long as the comparison test is true. So, something in the block of VB code had better change a variable value within the comparison test, or an "infinite loop" will occur. As soon as the comparison test becomes false, the loop terminates. The loop terminates as soon as the test is false, so the loop will never execute if the comparison is initially false.

In this example, the DO WHILE loop executes as long as comparison test is true.

```
Dim strAge As String
```

```
Dim intAge As Integer
```

```
Dim intPress As Integer
```

```
' Get the age in a string variable
```

```
strAge = InputBox("How old are you?", "intAge Ask")
```

```
' Check for the Cancel command button
```

```
If (strAge = "") Then
```

```
End ' Terminates the application
```

```
End If
```

```
' Cancel was not pressed, so convert Age to integer
```

```
' The Val() function converts strings to integers
```

```
intAge = Val(strAge)
```

```
' Loop if the age is not in the correct range
```

```

Do While ((intAge < 10) Or (intAge > 99))
' The user's age is out of range
intPress = MsgBox("Your age must be between " & _
    "10 and 99", vbExclamation, "Error!")
    strAge = InputBox("How old are you?", "Age Ask")
' Check for the Cancel command button
If (strAge = "") Then
    End ' Terminate the program
End If
    intAge = Val(strAge)
Loop

```

An almost identical example: the DO UNTIL loops until comparison test becomes true.

```

Dim strAge As String
Dim intAge As Integer
Dim intPress As Integer
' Get the age in a string variable
strAge = InputBox("How old are you?", "Age Ask")
' Check for the Cancel command button
If (strAge = "") Then
    End ' Terminate the program
End If
' Cancel was not pressed, so convert Age to integer
intAge = Val(strAge)
' Loop if the age is not in the correct range
Do Until ((intAge >= 10) And (intAge <= 99))
' The user's age is out of range
intPress = MsgBox("Your age must be " & _
    "between 10 and 99", vbExclamation, "Error!")
strAge = InputBox("How old are you?", "Age Ask")
' Check for the Cancel command button
If (strAge = "") Then
    End ' Terminate the program
End If
    intAge = Val(strAge)
Loop

```

Yet another variation of the same example: using the DO...LOOP WHILE to check the comparison at the bottom of the loop.

```

Dim strAge As String
Dim intAge As Integer
Dim intPress As Integer
Do
strAge = InputBox("How old are you?", "Age Ask")
' Check for the Cancel command button
If (strAge = i"") Then
    End ' Terminate program
End If
    intAge = Val(strAge)
If ((intAge < 10) Or (intAge > 99)) Then
' The user's age is out of range
intPress = MsgBox("Your age must be between " & _
    '10 and 99", vbExclamation, "Error!")

```

```
End If
Loop While ((intAge < 10) Or (intAge > 99))
```

In this example, the FOR...TO...NEXT structure is used to add the numbers from 1 to 100.

```
IntSum = 0
For intNumber = 1 To 100
IntSum = IntSum + intNumber
Next intNumber
```

This example uses a FOR loop to calculate compound interest.:

```
Sub cmdIntr_Click ()
' Use a For loop to calculate a final total
' investment using compound interest.
' intNum is a loop control variable
' snglRate is the annual interest rate
' intTerm is the Number of years in the investment
' curlnitlnv is the investor's initial investment
' snglInterest is the total interest paid
Dim snglRate As Single, snglInterest As Single
Dim intTerm As Integer, intNum As Integer
Dim curlnitlnv As Currency
snglRate = .08
intTerm = 5
' Watch out...The Code window might convert the
' following literals, 1000.00 and 1.0, to double
' precision literals with the suffix # to ensure
' accuracy.
curlnitlnv = 1000.00
snglInterest = 1.0 ' Begin at one for first compound
' Use loop to calculate total compound amount
For intNum = 1 To intTerm
snglInterest = snglInterest * (1 + snglRate)
Next intNum
' Now we have total interest,
' calculate the total investment
' at the end of N years
lblFinalInv.Caption = curlnitlnv * snglInterest
End Sub
```

Section 9: Combining Code and Controls

This section demonstrates how forms and the code imbedded in them are combined with external code to make a complete application. Themes addressed here include (1) where to put the initial form; (2) control arrays; (3) use of default properties; (4) adding external code modules; and (5) custom functions.

Control arrays: these are groups of controls of the same type. Text boxes and labels can be grouped together and given most of the same properties (except for positions, names and captions). Note that a locked property is protected against change by the user (e.g. a box where the result of a calculation is displayed).

Note that even before the main program code of an application has been written, the interface created for the project can be run even though the application has not been programmed for any

necessary calculations. The general appearance and nature of the GUI can be reviewed without the calculation code yet in place.

Unload: this statement removes a form from memory and returns all properties to the original design-time state. The format is *Unload Me* (here 'me' specifies the currently active form).

Format: this sets numeric or string values to the desired specification (such as two decimal places to the right, etc.). The format of the format statement is

Format (expression, strFormat)

Example:

```
txtEnding.Text=Format(curlnitInv*sngInterest,$###,##0.00")
```

Error checking: to insure correct input of values, you can use a message box to report any problems on the screen. You could convert an entered text box text into numbers, check ranges of entered values, then use the message box to tell the user of any apparent errors. You could then set the focus on the control with the bad value so the user sees where the error is. Or, you could use an error checking function to test values, call it `ErrorCheck()`; if it is set (= 1), exit the subroutine for the calculation. (A function returns a value while the subroutine does not.)

Add a module to a project (e.g. for error checking): add a module (default name module1, filename module1.bas) Checking all text box values for correct range, returning a 1 value for `ErrorCheck()` function if any errors are found:

```
Public Function ErrorCheck() As Integer
```

```
(various If...Then... tests to determine if user input is in range
```

```
End Function.
```

Section 10: List Boxes and Combo Boxes

This section covers list controls (List boxes and Combo boxes) and data and control arrays. The specific topics include (1) adding list boxes; (2) differences between list and combo boxes; (3) initializing lists; (4) declaring and using arrays, and (5) the usefulness of control arrays.

List Boxes: Call up the List Box control from the Tools | Options | Editor Format selection on the pulldown menu. Size the box on the form appropriately for the width of the data; scrollbars are provided automatically if necessary. The box can have 1, 2 or 3 columns (set using the Columns property). Note the List Box properties:

Property	Description
BackColor	Specifies the list box's background color.
Columns	Determines the number of columns. If 0, the list box scrolls vertically in a single column. If 1 or more, the list box items appear in the number of columns specified (one or more columns) and a <i>horizontal scrollbar</i> appears so you can see all the items in the list.
ForeColor	Specifies the list box's text color.
Height	Indicates the height of the list box in twips.
IntegralHeight	Determines whether the list box can display partial items, such as the upper half of an item that falls toward the bottom of the list box.
List	Holds, in a drop-down property list box, values that you can enter into the list box at design time. You can enter only one at a time, and most programmers usually prefer to initialize the list box at runtime.
MultiSelect	The state of the list box's selection rules. If 0 -None (the default), the user can select only one item by clicking with the mouse or by pressing the Spacebar over an item. If 1 -Simple, the user can select more than one item by clicking with the mouse or by pressing the Spacebar over

	items in the list. If 2 -Extended, the user can select multiple items using Shift+click and Shift+arrow to extend the selection from a previously selected item to the current one. Ctrl+click either selects or deselects an item from the list.
Sorted	Determines whether the list box values are automatically sorted. If False (the default value), the values appear in the same order in which the program added the items to the list.
style	Determines whether the list box appears in its usual list format or with check boxes next to the selected items.

Command buttons can work together with a list box, signaling which list elements are selected. List box “methods” let user identify selections, add items or remove items from list. Here are the common list box methods:

Method	Description
AddItem	Adds a single item to the list box.
Clear	Removes all items from the list box.
List	A string array that holds items from within the list box.
ListCount	The total number of list box items.
RemoveItem	Removes a single item from the list box.

Example of the format used for methods: 1stOneCol.AddItem “Joseph” adds that name to list. AddItem is used for addition to the list; RemoveItem to delete; these can be subscripted, counting from the first item as element 0. Clear removes all list items. Items in a list can be assigned to variables (example: strVar1 = 1stOneCol.list(3) Use ListCount to determine the number of items in a list (example: intNum = 1stOneCol.ListCount); such a value could then be used as the end value in a For...To...Next loop. “Selected” tells whether an item has been chosen by the user (if True, the item was selected). The List Box property MultiSelect permits more than 1 item to be selected.

Combo Boxes: These behave like List Boxes but also permit addition of items to the list. The three types of combo Boxes include (1) Drop-down; a single line with a down arrow; (2) simple, looking like a list box, but allowing items to be added; (3) drop-down list box, a single line equivalent of list box, with no ability to add items. These three can be set via Style combo box property. The fundamental combo box properties include:

Property	Description
BackColor	The combo box's background color.
ForeColor	The combo box's foreground text color.
Height	The height, in twips, of the closed combo box.
IntegralHeight	Determines whether the combo box can display partial items, such as the upper half of an item that falls toward the bottom of the combo box.
List	A drop-down property list box in which you can enter values into the combo box at design time. You can enter only one at a time, and most programmers prefer to initialize the combo box at runtime.
Sorted	Determines whether the combo box values are automatically sorted. If False (the default value), the values appear in the same order in which the program added the items to the combo box.
Style	Determines the type of combo box your application needs. If 0-DropDown Combo, the combo box is a drop-down combo box. If 1 -simple Combo, the combo box turns into a simple combo box that remains open to the height you use at design time. If 2 - DropDown List, the combo box turns into a drop-down list box that remains closed until the user is ready to see more of the list.

Items entered via a combo box don't automatically get added to the list without some necessary code. For example, `cboBox.AddItem cboBox.Text`. Or, you could add a command button to add the user entry.

Data arrays: arrays are lists of subscripted variables; example: `curDivSales(5)` is the sixth saved value in an array. Use Option Base 1 statement to begin subscripts with (1) instead of (0). Use Dim or Public to declare array variables, specifying the array size in parentheses. Example: `Dim sngLabData(10) As Single`. Subscript indexing makes it easy to process all the data in an array.

Control Arrays: these are lists of controls with the same name, distinguished with subscripts. This simplifies entering properties for multiple controls with mostly the same formats. For one control, the event procedure might be declared like this: `Private Sub cmdTotal_click()`. But an array would look like this: `Private Sub cmdTotal_click(Index As Integer)`. Then properties of members of the array could be addressed individually, e.g., `cmdTotal(Index).Caption="Here's a different caption."`

Section 11: Additional Controls

This section describes selection controls, scrollbars, and the timer; included are (1) option buttons; (2) check box styles; (3) scrollbar properties; and (4) the timer control.

Option buttons: (like the Mac's radio buttons) only one can be selected at a time (within a frame). A "frame" is a rectangular area on a form where controls are placed. The frame itself is a control upon which other controls can be grouped and placed. Frames then group controls for placement on the larger form. Use a control array for consistent properties like alignment.

Colors: eight common colors can be identified by name, although millions of different shades can be programmed. The literals (with the color they code for represented by the name preceded by the prefix "vb" (designating a built-in visual Basic literal) are `vbBlack`, `vbRed`, `vbGreen`, `vbYellow`, `vbBlue`, `vbMagenta`, `vbCyan`, and `vbWhite`.

Check boxes: like radio buttons, but not mutually exclusive (multiple selections can be checked). Box can be grayed out if unavailable. Available style values include `checked`, `unchecked`, `grayed`, and `unchecked and checked graphical`.

ScrollBars: properties are set to control relative position within range of values. Properties include:

<u>Property</u>	<u>Description</u>
LargeChange	Amount scrollbar changes (scrolls up or down or left or right) with each click in scrollbar's shaft area
Max	Maximum number of units at highest setting (default = 32767)
Min	Minimum number of units at lowest setting (default = 1)
SmallChange	Amount scrollbar changes with click at either end of scrollbar
Value	Unit of measurement represented by scrollbar

The VB Clock timer control is shown on a form at design time, but not at run time. Set Interval property from 1 to 65535, the time in milliseconds; generates a timer event. `tmrClock_Timer()` event. The timer is accurate to few tenths of a %.

Section 12: Dialog Boxes

This section covers (1) common dialog box controls; (2) types of dialog boxes; (3) design-time properties; (4) font selection; (5) being consistent with dialog boxes.

Common dialog box: these are not displayed on the form until programmed to appear by code, and depending on what the code “needs” at the time. The File Open dialog box is similar to that seen in applications like Excel or Word, allowing pathname selection, selecting another drive, etc. There is also a File Save dialog box. Using a common dialog box instead of File or Directory List boxes insures consistent appearance with commercial applications software, giving user immediate familiarity with what’s expected. Note however that these displays are the human interface only; you must still provide the code that senses input to the dialog box and takes appropriate action.

Other common dialog boxes: the Color Dialog box, with about fifty colors displayed, gives access to custom colors; the Font Dialog box gives user choice of what’s installed on the computer in use; the Print Dialog box, allows selection of a printer; and, the Help Dialog box links to help files that you must generate.

Adding the common dialog box control to the toolbox: (1) select Project | Components to display Components dialog box; (2) scroll to Microsoft Common Dialog box, select; (3) control will appear; (4) Double-click common dialog box to add control to form. Note that other controls can be evoked from the dialog box, such as ActiveX, Calendar, and others. Control on your form doesn’t look as it will at runtime, because the properties you set determine this. An additional Properties dialog box appears also: the tabs display choices (Open/Save As, Color, Font, Print, Help). A Filter property allows selection of the extension type to be requested, so only selected types of files are displayed (like *.asc, using wild card, for any ASCII files).

Common Dialog Box methods: specify either ShowColor, ShowFont, ShowHelp, ShowOpen, ShowPrinter, or ShowSave. For a Common Dialog Box named cdbFile, the statement would be cdbFile.ShowSave. Example given: displaying a File Open dialog box:

```

cdbDialog.Title = "File Open"
cdbDialog.Filter = "*.txt"           ' show only text files
cdbDialog.FileName = "*.txt"        ' default
cdbDialog.ShowOpen                  ' now display the dialog box

```

Section 13: Database Basics

This section covers file access coding, including (1) file terms; (2) opening and closing files; (3) writing to a file; (4) reading from a file; (5) using the Data control; (6) bound controls; and (7) using the Data Form Wizard.

Accessing disk files: a "file" is a collection of data, (either/or or both/and), textual or database format. Files with exactly the same filename must reside on different disks or in different directories. Easiest to use are text or ASCII files; other files are encoded in binary (machine-readable only) formats. You must direct VB to the exact location of the file you want to access; you can do this using the File Open dialog box (allowing quick selection of file to be accessed, drive, subdirectory, etc.)

Open file: the format is

```
Open strFileName [For Mode] As ([#]intFileNumber).
```

The FileName given must reside on the default drive unless the full pathname is specified. Use the CurDir() function to determine the current directory, then stick this string into the full pathname. Here the “Mode” specifies what you wish to do with the accessed data: Append (add new data to the end of a file), Input (to read data from the file - that is, input to your program); and Output (write data into the file - that is, output from your program). FileNumber is an integer between 1 and 255. FileNumber (or channel) remains associated with the data file until that file is closed. VB can have multiple channels open at one time.

Close file: this finishes the writing of any file data into the file (from the buffer), and releases file number for subsequent use. "Close" without arguments closes all open files The specific format is

Close [#]intFileNumber [,...,[#intFileNumber]

Delete file: this "Kill" statement permanently erases a file from the disk. Its format is:

Kill "c:\Dat\MyData.DAT"

This erases MyData.DAT in the file folder on C called Dat.

Write to a file: The Write# command writes any type of data to a file opened in "Output" or "Append" modes; this works on strings and numbers in all combinations. The command format is:

Write #intFileNumber [,ExpressionList]

Here intFileNumber is associated with file opened for output. If no values are given, a CR LF (carriage return and line feed) are provided. Items are separated by commas; the end of each written line is terminated with a CR LF; quotation marks are added around all strings in the file. Logical values are bracketed by pound sign (e.g. #True#). A Write # line is ended with a semicolon to get next Write# to continue on the same line of the data file.

Example:

Write #3, intIDno, intAge, strPatName, strDrName,curBalance

might record a single line of data to the file that would look like

1047,31,"Sam Jones","J. D. Smith, MD", "2344.75"

Such a "Write #" statement could be within as a For...To...Next... loop where each of the integer and string variables could be subscripted with an indexing variable like i (e.g. intIDno(i), etc.).

Write# will add to the end of a file if the file is opened in Append mode. Otherwise, opening in Output mode will cause the data to overwrite existing data in the file.

Input data to a file: Input# reads data into your program from a stored file. The format is

Input #intFileNumber [,EpressionList]

This is the reverse of the write# statement. You must match incoming data with the appropriate data type. Attempts to read more data into your program than exists in the file will generate an error message. Use the EOF() function to test for the end of the file: the format is Eof(intFileNumber). This function returns True if at end of file, or False if not. A good program design is to use Do Until the end of file is reached: Do Until (EOF(intLineNumber) = True) Use a counter in the loop to 'remember' number of data values retrieved.

Line Input#: This reads data from a file into the program in form of a single string variable; the line of data is expected to end with CR LF (the way most file records end). Then the data can be parsed.

Database processing: A Database is a program used for storage, retrieval, reporting and organizing of data. Database programs include Access, dBASE, FoxPro, Lotus, and Paradox. The popular spreadsheet program Excel can also be used as a database management program. You can access data stored in these DB programs using the Data Control or the Data Form Wizard features of VB 6.

Section 14: Menus and Visual Basic

This section explains (1) the use of the Menu Editor; (2) how to add a menu bar to your program; (3) how to add submenus; (4) how to name menu options; and (5) where to put the code for menu events.

Menus are control objects comparable to buttons or text boxes; the menu bar has properties you can set from the Properties window, like any object; but the Menu Editor is easier to use than manual control of your menu's design through the Properties window. To invoke the Menu Editor, select Tools/Menu Editor. The editor creates a menu but you must write the event procedures

which link menu selections (by a mouse click on the menu item) to particular actions. The editor lets you add a menu bar, pull-down menu commands, separator bars (grouping menu options), submenus and checked items to your application.

Try it: add a standard menu bar to a new application. Open the menu editor. Each menu bar command must be given a Caption and a Name. Add the following options one by one, using the tab key to move from Caption to Name property: File (caption = &File, name = mnuFile), Edit (&Edit, mnuEdit), View (&View, mnuView) and Help (&Help, mnuHelp). The ampersand prefixes assign accelerator keystrokes for the commands. The Menu Editor's lower section displays the menu and options as they are added. Look at the result by running the program after the menu bar is created (although nothing actually happens without further code if you try to click on a menu command).

Add menu options: follow standard naming conventions by using the 'mnu' prefix, then the name of the menu command, followed by the pull-down menu item. Return to the Menu Editor and insert a pull-down command to appear under File. Select the &Edit line in the lower portion of the Editor box, then click on 'insert'; a blank line will appear. Use the right arrow button to indent (four dots - an ellipsis -appears when this is done). The left arrow removes a level of indentation, and the up-down arrows allow scrolling to different menu items. Enter the caption &New, then tab to enter the name mnuFileNew. Note the naming convention: the prefix mnu, and the Menu command under which this pulldown command is to appear. Repeat inserts to add Open (&Open, mnuFileOpen), Close (&Close, mnuFileClose) and Exit (&Exit, mnuFileExit) to the menu. Finally, add a separator bar above the Exit command under File. Insert and indent, using a hyphen (-) as the caption property and mnuFileBar1 as the name.

Try two more options: Add a checked object. Add an indented option (caption = Highlighted, name = mnuViewHighlighted) under View. Add a submenu off File/Open to give two choices: Binary (&Binary, mnuFileOpenBinary) and Text (&Text, mnuFileOpenText).

Link the menu selections to event procedures: Remember that the menu objects must be linked to code which will respond to the button click. In design time with the Forms window displayed, double click on the pulldown Exit selection under the File menu to open a code window showing the 'wrapper' lines Private Sub mnuFileExit_Click() and End Sub. Enter Unload Me as one line, then the keyword End. Run this program to verify that selection of Exit from the File pulldown menu will terminate program execution.

Section 15: Debugging Programs

This section explains (1) programming errors to avoid; (2) using breakpoints to debug your program; (3) examining variables at runtime; (4) single stepping through a program; and (5) using the Immediate window to change variable values in the middle of a program.

What kinds of errors can occur in the program you write, and how do you go about "debugging" your program? A common bug is the 'syntax error', a misspelled VB keyword or a grammar error. Note that the Select Tools/Options dialog box allows these errors to be reported immediately upon their entry. Sometimes the error as reported and highlighted is a bit misleading (e.g. 'Compile' errors can refer to a variety of not-so-intuitive mistakes). Look in the immediate vicinity of a highlighted line to find the error. Another bug category is a 'runtime error' which is detected only when the program is executed. The 'divide by zero' is such an error because the zero value taken by a variable isn't known until the calculation is attempted during program execution. The runtime error dialog box displayed in this case gives four button choices: continue (program execution if possible; may be dimmed and inoperable with fatal errors), end (stop program execution, return to design time mode), debug (enter the debugger window), and help (provide details about the nature of the detected error type). Selecting Debug can possibly allow a fix and a resumption of the program.

The most difficult type of error to fix is a so-called 'logic error', a flaw in the program's design. These are best corrected by using the debugger, which can allow you to trace program execution. Note the VB Debug pulldown menu choices. The debugger allows insertion of "breakpoints", temporary halts in program execution to allow the user to examine program status, or to change values of program variables. In 'break mode', your interrupted program retains all variable values at the moment of interruption, allowing you to examine data values from one line of code. You often can determine the source of an error by comparing these actual values of variables with the values you *thought* they would have.

You can set a breakpoint during runtime program execution using the control + break keys, or the Run pulldown menu 'break' feature during program execution. In design mode or break mode, you can set a breakpoint on a particular line of code where you want execution to halt. Also, the Debug/Add watch command lets you specify an expression that will cause a break when the expression becomes true.

What can you do at a breakpoint? Move your mouse to a variable to get a tooltip-like pop-up message giving the current value (at breakpoint) of that variable. Or, select the Debug/Quick Watch option, a dialog box showing the breakpoint line, variable name, and current value. Another option is to select the Watch dialog box, to track multiple variables that are continuously updated as the program is executed (to the next occurrence of the breakpoint). Yet another option is to display the Debug toolbar, a floating toolbar (View/Toolbars/Debug).

Single stepping through code: once a breakpoint is reached, you can execute a single line at a time with Debug/Step into command. Step Over allows jumping over repeated calls to a subroutine. Step Out exits a current procedure without single stepping. In the Immediate Window, you can enter VB commands directly, and have them executed. Print statements executed here will cause specified variables to be printed to the immediate window.

Please note: Some of this material was adapted from Greg Perry's "Teach Yourself Visual Basic 6 in 24 Hours", Indianapolis, IN: SAMS Publishing, 450 pp., 1998.