Computer Programming Concepts and Visual Basic

David I. Schneider

UNIVERSITY OF PHOENIX

COLLEGE OF INFORMATION SYSTEMS AND TECHNOLOGY

Cover Art: Copyright © Stock Illustration Source, Inc. / Kelly Brother.

Excerpts taken from:

An Introduction to Programming Using Visual Basic 6.0, Fourth Edition,
by David I. Schneider
Copyright © 1999, 1998, 1997, 1995 by Prentice-Hall, Inc.
A Pearson Education Company
Upper Saddle River, New Jersey 07458

IBM is a registered trademark ® of International Business Machines Corporation
Hercules is a registered trademark ® of Hercules Computer Technology.
Visual Basic is a registered trademark ® of the Microsoft Corporation.

The information, illustrations, and / or software contained in this book, and regarding the above-mentioned programs, are provided "As Is," without warranty of any kind, express or implied, including without limitation any warranty concerning the accuracy, adequacy, or completeness of such information. Neither the publisher, the authors, nor the copyright holders shall be responsible for any claims attributable to errors, omissions, or other inaccuracies contained in this book. Nor shall they be liable for direct, indirect, special, incidental, or consequential damages arising out of the use of such information or material.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

This special edition published in cooperation with
Pearson Custom Publishing

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Please visit our web site at www.pearsoncustom.com

ISBN 0–536–60446–0

BA 990807

PEARSON CUSTOM PUBLISHING
160 Gould Street/Needham Heights, MA 02494
A Pearson Education Company

CONTENTS

SECTION 1: PROBLEM SOLVING 1
- 1.1 Program Development Cycle 3
- 1.2 Programming Tools 5

SECTION 2: FUNDAMENTALS OF PROGRAMMING IN VISUAL BASIC 15
- 2.1 Visual Basic Objects 17
- 2.2 Visual Basic Events 27
- 2.3 Numbers 34
- 2.4 Strings 42
- 2.5 Input and Output 49
- 2.6 Built-In Functions 58
- Summary 67
- Programming Projects 68

SECTION 3: GENERAL PROCEDURES 71
- 3.1 Sub Procedures, Part I 73
- 3.2 Sub Procedures, Part II 80
- 3.3 Function Procedures 87
- 3.4 Modular Design 93
- Summary 97
- Programming Projects 97

SECTION 4: DECISIONS 101
- 4.1 Relational and Logical Operators 103
- 4.2 If Blocks 106
- 4.4 Select Case Blocks 112
- 4.4 A Case Study: Weekly Payroll 119
- Summary 126
- Programming Projects 126

SECTION 5: REPETITION 129
- 5.1 Do Loops 131
- 5.2 Processing Lists of Data with Do Loops 135
- 5.3 For...Next Loops 141
- 5.4 A Case Study: Analyze a Loan 146
- Summary 153
- Programming Projects 153

SECTION 6: ARRAYS 159
- 6.1 Creating and Accessing Arrays 161
- 6.2 Using Arrays 171
- 6.3 Control Arrays 179
- 6.4 Sorting and Searching 184
- 6.5 Two-Dimensional Arrays 196
- 6.6 A Case Study: Calculating with a Spreadsheet 200

Summary . . . 209
Programming Projects . . . 209

SECTION 7: SEQUENTIAL FILES . . . 215
7.1 Sequential Files . . . 217
7.2 Using Sequential Files . . . 223
7.3 A Case Study: Recording Checks and Deposits . . . 228
Summary . . . 237
Programming Projects . . . 237

SECTION 8: RANDOM-ACCESS FILES . . . 243
8.1 User Defined Data Types . . . 245
8.2 Random-Access Files . . . 251
Summary . . . 255
Programming Projects . . . 255

SECTION 9: THE GRAPHICAL DISPLAY OF DATA . . . 257
9.1 Introduction to Graphics . . . 259
9.2 Line Charts . . . 267
9.3 Bar Charts . . . 273
9.4 Pie Charts . . . 277
Summary . . . 282
Programming Projects . . . 282

SECTION 10: ADDITIONAL CONTROLS AND OBJECTS . . . 285
10.1 List Boxes and Combo Boxes . . . 287
10.2 Nine Elementary Controls . . . 294
10.3 Five Additional Objects . . . 303
Summary . . . 316
Programming Projects . . . 317

SECTION 11: DATABASE MANAGEMENT . . . 321
11.1 An Introduction to Databases . . . 323
11.2 Relational Databases and SQL . . . 330
11.3 Three Additional Data-Bound Controls; Creating and Designing Databases . . . 338
Summary . . . 350
Programming Projects . . . 350

SECTION 12: OBJECT-ORIENTED PROGRAMMING . . . 353
12.1 Classes and Objects . . . 355
12.2 Collections and Events . . . 365
12.3 Class Relationships . . . 373
Summary . . . 380
Programming Projects . . . 381

SECTION 13: COMMUNICATING WITH OTHER APPLICATIONS . . . 383
13.1 OLE . . . 385
13.2 Accessing the Internet with Visual Basic . . . 392

13.3 Web Page Programming with VBScript 397
Summary 403

APPENDICES
A. ANSI Values 407
B. How To 409
C. Visual Basic Statements, Functions, Methods, Properties, Events, Data Types, and Operators 423
D. Visual Basic Debugging Tools 457

INDEX 465

CONGRATULATIONS…

You have just purchased access to a valuable website that will open many doors for you! The University of Phoenix has chosen to enhance and expand your course's material with a dynamic website that contains an abundance of rich and valuable online resources specifically designed to help you achieve success!

This website provides you with material selected and added to powerful online tools that have been seamlessly integrated with this textbook, resulting in a dynamic, course-enhancing learning system. These exciting tools include:

Online Study Guide
Online glossary
Links to selected, high-quality websites
And more!

You can begin to access these tremendous resources immediately!

www.pearsoncustom.com/uop: The opening screen of the University of Phoenix website includes book covers of all the Pearson Custom Publishing books in the BSBIS and BSIT programs. Click on the book cover representing your course. This will launch the online study guide for the course in which you are currently enrolled **and** the glossary of key terms for all the University of Phoenix BSBIS and BSIT courses.

CD-ROM: The accompanying CD-ROM includes key terms underlined within the online book that are linked to the World Wide Web. Use the enclosed CD-ROM to launch websites selected to reinforce your learning experience.

TAKE THE FIRST STEP ON THE ROAD TO SUCCESS TODAY!

SECTION 1

PROBLEM SOLVING

1.1 PROGRAM DEVELOPMENT CYCLE

Hardware refers to the machinery in a computer system (such as the monitor, keyboard, and CPU) and software refers to a collection of instructions, called a **program** (or **project**), that directs the hardware. Programs are written to solve problems or perform tasks on a computer. Programmers translate the solutions or tasks into a language the computer can understand. As we write programs, we must keep in mind that the computer will only do what we instruct it to do. Because of this, we must be very careful and thorough with our instructions.

Performing a Task on the Computer

The first step in writing instructions to carry out a task is to determine what the **output** should be—that is, exactly what the task should produce. The second step is to identify the data, or **input**, necessary to obtain the output. The last step is to determine how to **process** the input to obtain the desired output, that is, to determine what formulas or ways of doing things can be used to obtain the output.

This problem-solving approach is the same as that used to solve word problems in an algebra class. For example, consider the following algebra problem:

How fast is a car traveling if it goes 50 miles in 2 hours?

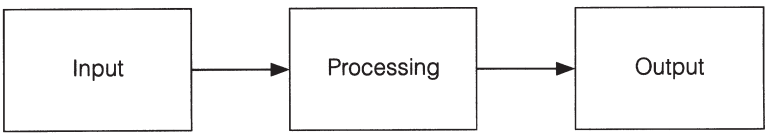
The first step is to determine the type of answer requested. The answer should be a number giving the rate of speed in miles per hour (the output). The information needed to obtain the answer is the distance and time the car has traveled (the input). The formula

rate distance / time

is used to process the distance traveled and the time elapsed in order to determine the rate of speed. That is,

rate = 50miles / 2
= 25 miles / hour

A pictorial representation of this problem-solving process is



We determine what we want as output, get the needed input, and process the input to produce the desired output.

In the following chapters we discuss how to write programs to carry out the preceding operations. But first we look at the general process of writing programs.

Program Planning

A recipe provides a good example of a plan. The ingredients and the amounts are determined by what is to be baked. That is, the *output* determines the *input* and the *processing*. The recipe, or plan, reduces the number of mistakes you might make if you tried to bake with no plan at all. Although it's difficult to imagine an architect building a bridge or a factory without a detailed plan, many programmers (particularly students in their first programming course) frequently try to write programs without first making a careful plan. The more complicated the problem, the more complex the plan must be. You will spend much less time working on a program if you devise a carefully thought out step-by-step plan and test it before actually writing the program.

Many programmers plan their programs using a sequence of steps, referred to as the **program development cycle**. The following step-by-step process will enable you to use your time efficiently and help you design error-free programs that produce the desired output.

1. ***Analyze:*** Define the problem.

   Be sure you understand what the program should do, that is, what the output should be. Have a clear idea of what data (or input) are given and the relationship between the input and the desired output.

2. ***Design:*** Plan the solution to the problem.

   Find a logical sequence of precise steps that solve the problem. Such a sequence of steps is called an **algorithm**. Every detail, including obvious steps, should appear in the algorithm. In the next section, we discuss three popular methods used to develop the logic plan: flowcharts, pseudocode, and top-down charts. These tools help the programmer break a problem into a sequence of small tasks the computer can perform to solve the problem.

   Planning also involves using representative data to test the logic of the algorithm by hand to ensure that it is correct.

3. ***Choose the interface:*** Select the objects (text boxes, command buttons, etc.).

   Determine how the input will be obtained and how the output will be displayed. Then create objects to receive the input and display the output. Also, create appropriate command buttons to allow the user to control the program.

4. ***Code:*** Translate the algorithm into a programming language.

   Coding is the technical word for writing the program. During this stage, the program is written in Visual Basic and entered into the computer. The programmer uses the algorithm devised in Step 2 along with a knowledge of Visual Basic.

5. ***Test and debug:*** Locate and remove any errors in the program.

   Testing is the process of finding errors in a program, and **debugging** is the process of correcting errors that are found. (An error in a program is called a **bug.**) As the program is typed, Visual Basic points out certain types of program errors. Other types of errors will be detected by Visual Basic when the program is executed; however, many errors due to typing mistakes, flaws in the algorithm, or incorrect usages of the Visual Basic language rules only can be uncovered and corrected by careful detective work. An example of such an error would be using addition when multiplication was the proper operation.

6. ***Complete the documentation:*** Organize all the material that describes the program.

   Documentation is intended to allow another person, or the programmer at a later date, to understand the program. Internal documentation consists of statements in the program that are not executed, but point out the purposes of various parts of the program. Documentation might also consist of a detailed description of what the program does and how to use the program (for instance, what type of input is expected). For commercial programs, documentation includes an instruction manual. Other types of documentation are the flowchart, pseudocode, and top-down chart that were used to construct the program. Although documentation is listed as the last step in the program development cycle, it should take place as the program is being coded.

1.2 PROGRAMMING TOOLS

This section discusses some specific algorithms and develops three tools used to convert algorithms into computer programs: flowcharts, pseudocode, and hierarchy charts.

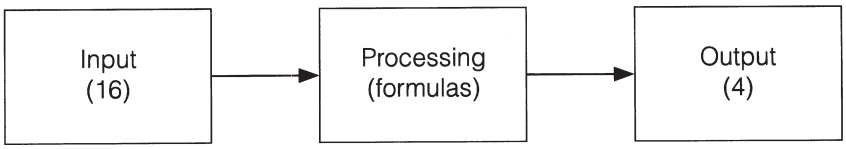
You use algorithms every day to make decisions and perform tasks. For instance, whenever you mail a letter, you must decide how much postage to put on the envelope. One rule of thumb is to use one stamp for every five sheets of paper or fraction thereof. Suppose a friend asks you to determine the number of stamps to place on an envelope. The following algorithm will accomplish the task.

1. Request the number of sheets of paper; call it Sheets. *(input)*
2. Divide Sheets by 5. *(processing)*
3. Round the quotient up to the next highest whole number; call it Stamps. *(processing)*
4. Reply with the number Stamps. *(output)*

The preceding algorithm takes the number of sheets (Sheets) as input, processes the data, and produces the number of stamps needed (Stamps) as output. We can test the algorithm for a letter with 16 sheets of paper.

1. Request the number of sheets of paper; Sheets = 16.
2. Dividing 5 into 16 gives 3.2.
3. Rounding 3.2 up to 4 gives Stamps = 4.
4. Reply with the answer, 4 stamps.

This problem-solving example can be pictured by



Of the program design tools available, the three most popular are the following:

Flowcharts: Graphically depict the logical steps to carry out a task and show how the steps relate to each other.

Pseudocode: Uses English-like phrases with some Visual Basic terms to outline the task.

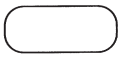
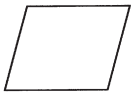
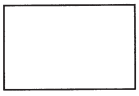
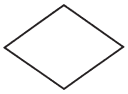
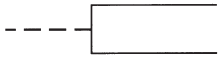
Hierarchy charts: Show how the different parts of a program relate to each other.

FLOWCHARTS

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. Figure 1-1 shows the flowchart for the postage stamp problem.

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
	Flowline	Used to connect symbols and indicate the flow of logic.
	*Terminal*	Used to represent the beginning (Start) or the end (End) of a task.
	*Input/Output*	Used for input and output operations, such as reading and printing. The data to be read or printed are described inside.
	*Processing*	Used for arithmetic and data-manipulation operations. The instructions are listed inside the symbol.
	*Decision*	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."
	Connector	Used to join different flowlines.
	Offpage Connector	Used to indicate that the flowchart continuesto a second page.
	Predefined Process	Used to represent a group of statements that perform one processing task.
	*Annotation*	Used to provide additional information about another flowchart symbol.

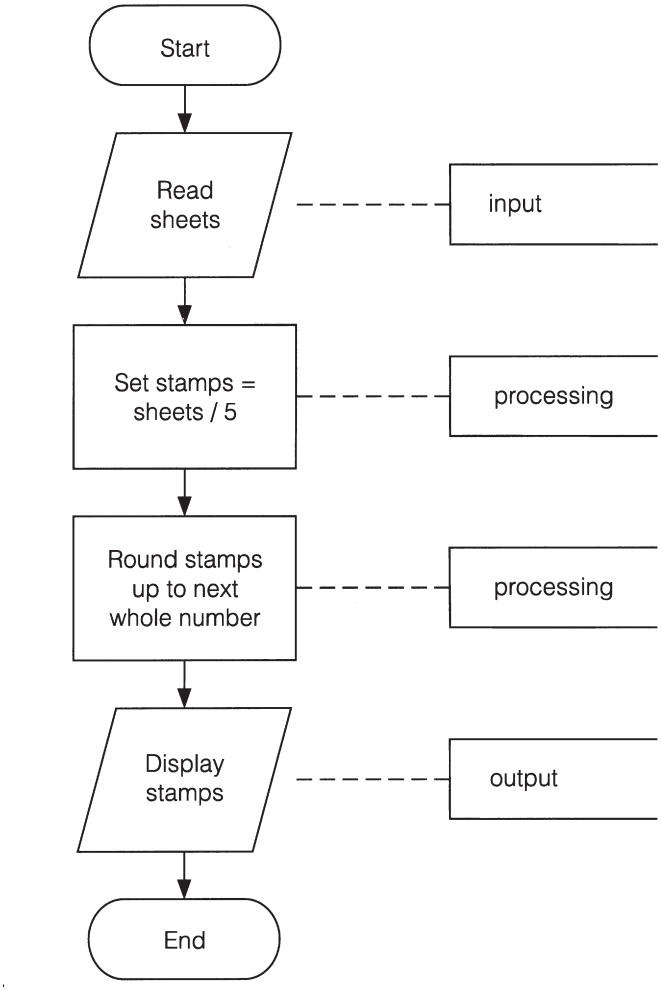


FIGURE 1-1 *Flowchart for the Postage Stamp Problem*

PSEUDOCODE

Pseudocode is an abbreviated version of actual computer code (hence, *pseudocode*). The geometric symbols used in flowcharts are replaced by English-like statements that outline the process. As a result, pseudocode looks more like computer code than does a flowchart. Pseudocode allows the programmer to focus on the steps required to solve a problem rather than on how to use the computer language. The programmer can describe the algorithm in Visual Basic-like form without being restricted by the rules of Visual Basic. When the pseudocode is completed, it can be easily translated into the Visual Basic language.

The following is pseudocode for the postage stamp problem:

Program: Determine the proper number of stamps for a letter

Read Sheets	*(input)*
Set the number of stamps to Sheets / 5	*(processing)*
Round the number of stamps up to the next whole number	*(processing)*
Display the number of stamps	*(output)*

Pseudocode has several advantages. It is compact and probably will not extend for many pages as flowcharts commonly do. Also, the plan looks like the code to be written and so is preferred by many programmers.

Hierarchy Chart

The last programming tool we'll discuss is the **hierarchy chart**, which shows the overall program structure. Hierarchy charts are also called structure charts, HIPO (Hierarchy plus Input-Process-Output) charts, top-down charts, or VTOC (Visual Table of Contents) charts. All these names refer to planning diagrams that are similar to a company's organization chart.

Hierarchy charts depict the organization of a program but omit the specific processing logic. They describe what each part, or **module**, of the program does and they show how the modules relate to each other. The details on how the modules work, however, are omitted. The chart is read from top to bottom and from left to right. Each module may be subdivided into a succession of submodules that branch out under it. Typically, after the activities in the succession of submodules are carried out, the module to the right of the original module is considered. A quick glance at the hierarchy chart reveals each task performed in the program and where it is performed. Figure 1-2 shows a hierarchy chart for the postage stamp problem.

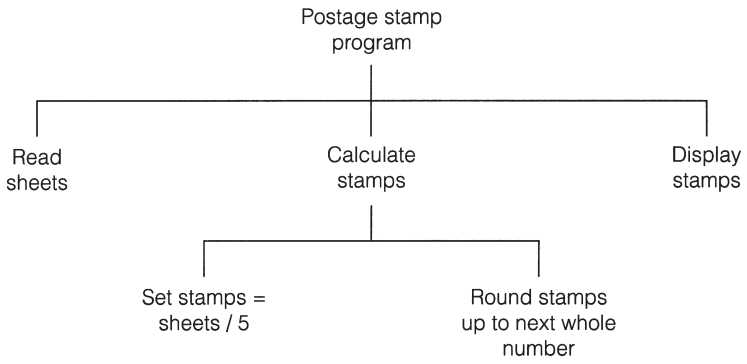


FIGURE 1-2 *Hierarchy Chart for the Postage Stamp Problem*

The main benefit of hierarchy charts is in the initial planning of a program. We break down the major parts of a program so we can see what must be done in general. From this point, we can then refine each module into more detailed plans using flowcharts or pseudocode. This process is called the **divide-and-conquer** method.

The postage stamp problem was solved by a series of instructions to read data, perform calculations, and display results. Each step was in a sequence; that is, we moved from one line to the next without skipping over any lines. This kind of structure is called a **sequence structure**. Many problems, however, require a decision to determine whether a series of instructions should be executed. If the answer to a question is "Yes," then one group of instructions is executed. If the answer is "No," then another is executed. This structure is called a **decision structure**. Figure 1-3 contains the pseudocode and flowchart for a decision structure.

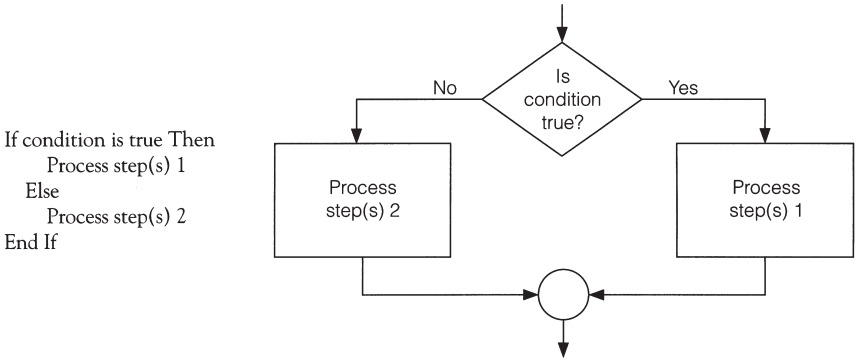


FIGURE 1-3 *Pseudocode and Flowchart for a Decision Structure*

The sequence and decision structures are both used to solve the following problem.

Direction of Numbered NYC Streets Algorithm

Problem: Given a street number of a one-way street in New York, decide the direction of the street, either eastbound or westbound.

Discussion: There is a simple rule to tell the direction of a one-way street in New York: Even numbered streets run eastbound.

Input: Street number

Processing: Decide if the street number is divisible by 2.

Output: "Eastbound" or "Westbound"

Figures 1-4 through 1-6 show the flowchart, pseudocode, and hierarchy chart for the New York numbered streets problem.

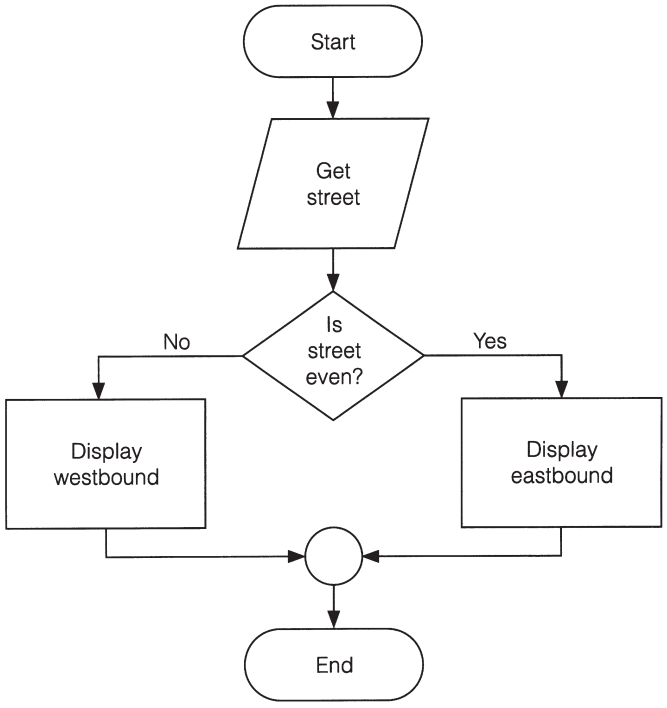


FIGURE 1-4 *Flowchart for the New York Numbered Streets Problem*

Program: Determine the direction of a numbered NYC street.

```
Get Street
If Street is even Then
      Display Eastbound
   Else
      Display Westbound
End If
```

FIGURE 1-5 *Pseudocode for the New York Numbered Streets Problem*

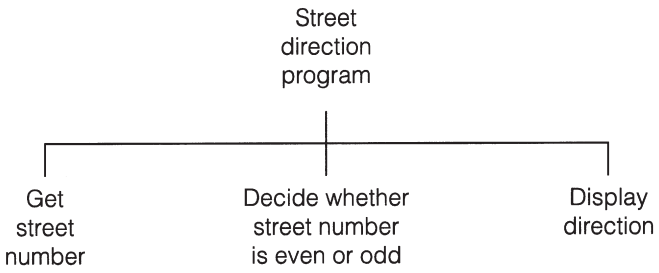


FIGURE 1-6 *Hierarchy Chart for the New York Numbered Streets Problem*

The solution to the next problem requires the repetition of a series of instructions. A programming structure that executes instructions many times is called a **loop structure**.

We need a test (or decision) to tell when the loop should end. Without an exit condition, the loop would repeat endlessly (an infinite loop). One way to control the number of times a loop repeats (often referred to as the number of passes or iterations) is to check a condition before each pass through the loop and continue executing the loop as long as the condition is true. See Figure 1-7.

```
Do While condition is true
      Process step(s)
Loop
```

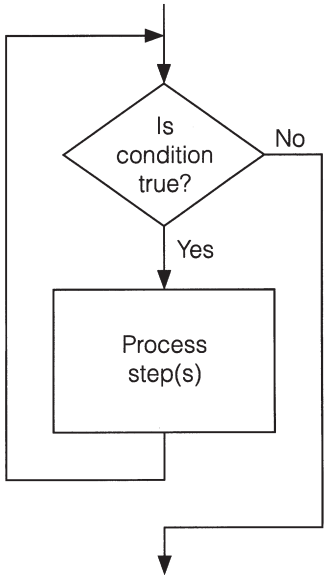


FIGURE 1-7 *Pseudocode and Flowchart for a Loop*

Class Average Algorithm

Problem: Calculate and report the grade-point average for a class.

Discussion: The average grade equals the sum of all grades divided by the number of students. We need a loop to read and then add (accumulate) the grades for each student in the class. Inside the loop, we also need to total (count) the number of students in the class. See Figures 1-8 to 1-10.

Input: Student grades

Processing: Find the sum of the grades; count the number of students; calculate average grade = sum of grades / number of students.

Output: Average grade

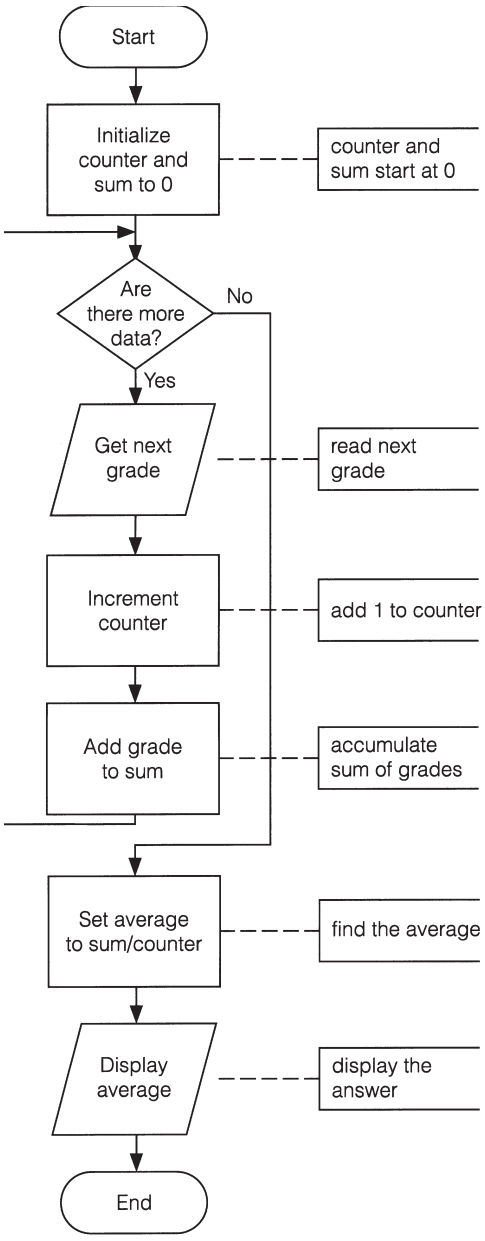


FIGURE 1-8 *Flowchart for the Class Average Problem*

```
Program: Determine the average grade of a class
Initialize Counter and Sum to 0
Do While there are more data
   Get the next Grade
   Add the Grade to the Sum
   Increment the Counter Loop
Compute Average = Sum / Counter
Display Average
```

FIGURE 1-9 *Pseudocode for the Class Average Problem*

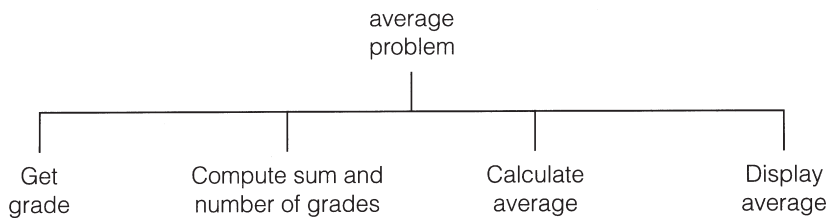


FIGURE 1-10 *Hierarchy Chart for the Class-Average Problem*

COMMENTS

1. Tracing a flowchart is like playing a board game. We begin at the Start symbol and proceed from symbol to symbol until we reach the End symbol. At any time, we will be at just one symbol. In a board game, the path taken depends on the result of spinning a spinner or rolling a pair of dice. The path taken through a flowchart depends on the input.
2. The algorithm should be tested at the flowchart stage before being coded into a program. Different data should be used as input, and the output checked. This process is known as **desk checking**. The test data should include nonstandard data as well as typical data.
3. Flowcharts, pseudocode, and hierarchy charts are universal problem-solving tools. They can be used to construct programs in any computer language, not just Visual Basic.
4. Flowcharts are used throughout this text to provide a visualization of the flow of certain programming tasks and Visual Basic control structures. Major examples of pseudocode and hierarchy charts appear in the case studies.
5. There are four primary logical programming constructs: sequence, decision, loop, and unconditional branch. Unconditional branch, which appears in some languages as Goto statements, involves jumping from one place in a program to another. Structured programming uses the first three constructs but forbids the fourth. One advantage of pseudocode over flowcharts is that pseudocode has no provision for unconditional branching and thus forces the programmer to write structured programs.
6. Flowcharts are time-consuming to write and difficult to update. For this reason, professional programmers are more likely to favor pseudocode and hierarchy charts. Because flowcharts so clearly illustrate the logical flow of programming techniques, however, they are a valuable tool in the education of programmers.

7. There are many styles of pseudocode. Some programmers use an outline form, whereas others use a form that looks almost like a programming language. The pseudocode appearing in the case studies of this text focuses on the primary tasks to be performed by the program and leaves many of the routine details to be completed during the coding process. Several Visual Basic keywords, such as, Print, If, Do, and While, are used extensively in the pseudocode appearing in this text.
8. Many people draw rectangles around each item in a hierarchy chart. In this text, rectangles are omitted to encourage the use of hierarchy charts by making them easier to draw.

SECTION 2

FUNDAMENTALS OF PROGRAMMING IN VISUAL BASIC

2.1 VISUAL BASIC OBJECTS

Visual Basic programs display a Windows style screen (called a **form**) with boxes into which users type (and edit) information and buttons that they click to initiate actions. The boxes and buttons are referred to as **controls**. Forms and controls are called **objects**. In this section, we examine forms and four of the most useful Visual Basic controls.

Note: If Visual Basic has not been installed on your computer, you can install it by following the steps outlined on the first page of Appendix B.

Invoking Visual Basic 6.0: To invoke Visual Basic, click the Start button, point to Programs, point to Microsoft Visual Basic 6.0, and click on Microsoft Visual Basic 6.0 in the final list.

With all versions of Visual Basic 6.0, the center of the screen will contain the New Project window of Figure 2-1. The main part of the window is a tabbed dialog box with three tabs—New, Existing, and Recent. (If the New tab is not in the foreground, click on it to bring it to the front.) The number of project icons showing are either three (with the Working Model and Learning Editions) or thirteen (with the Professional and Enterprise Editions).

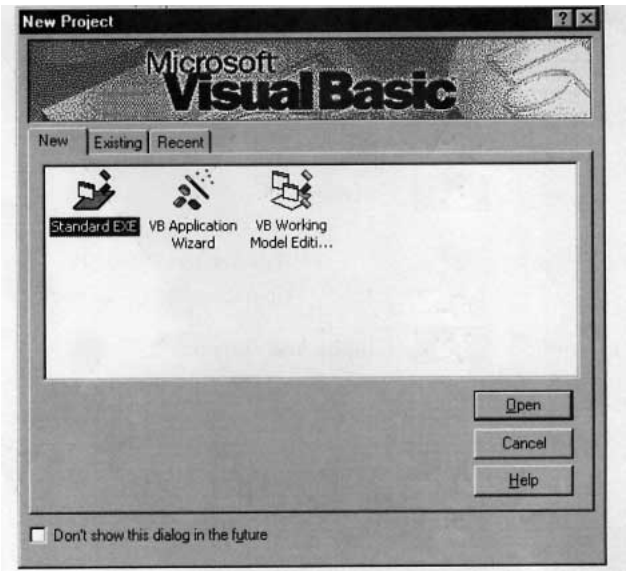


FIGURE 2-1 *New Project Window from the Working Model Edition of VB 6.0*

Double-click the Standard EXE icon to bring up the initial Visual Basic screen in Figure 2-2. The appearance of this screen varies slightly with the different versions of Visual Basic.

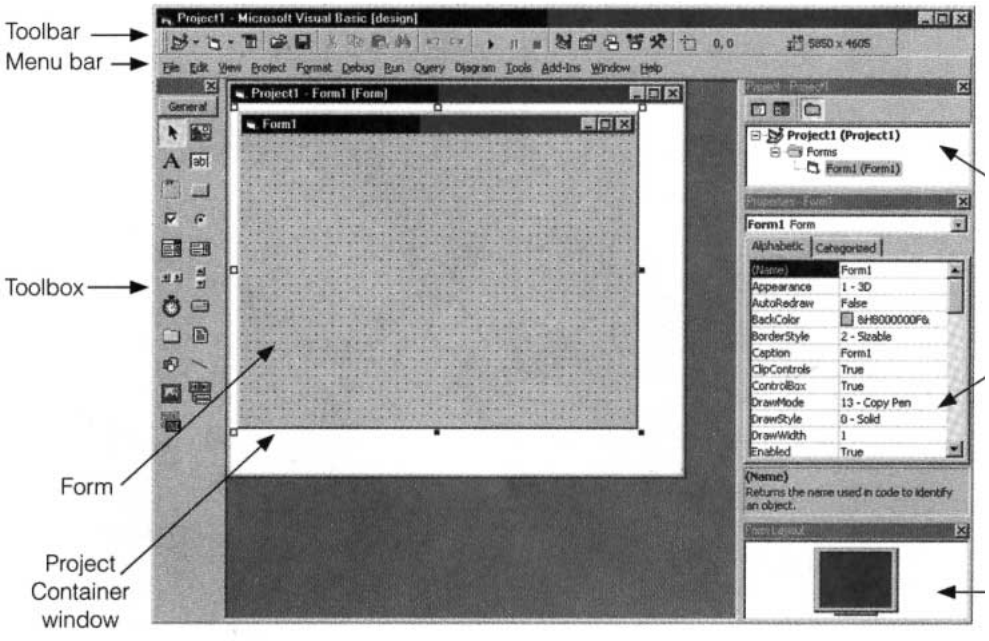


FIGURE 2-2 *The Initial Visual Basic Screen*

The **Menu bar** of the Visual Basic screen displays the commands you use to work with Visual Basic. Some of the menus, like File, Edit, View, and Window, are common to most Windows applications. Others, such as Project, Format, and Debug, provide commands specific to programming in Visual Basic.

The **Toolbar** is a collection of icons that carry out standard operations when clicked. For example, the fifth icon, which looks like a diskette, can be used to save the current program to a disk. To reveal the function of a Toolbar icon, position the mouse pointer over the icon for a few seconds.

The large stippled **Form window**, or **form** for short, becomes a Windows window when a program is executed. Most information displayed by the program appears on the form. The information usually is displayed in controls that have been placed on the form. The **Form Layout window** allows you to position the location of the form at run time relative to the entire screen using a small graphical representation of the screen.

The **Project Explorer window** is seldom needed for our purposes until Section 12. The **Properties window** is used to change how objects look and react.

The icons in the **Toolbox** represent controls that can be placed on the form. The four controls discussed in this section are text boxes, labels, command buttons, and picture boxes.

Text boxes: You use a text box primarily to get information, referred to as **input**, from the user.

Labels: You place a label to the left of a text box to tell the user what type of information to enter into the text box. You also use labels to display output.

Command buttons: The user clicks a command button to initiate an action.

Picture boxes: You use a picture box to display text or graphics output.

A TEXT BOX WALKTHROUGH

1. Double-click on the text box icon. (The text box icon consists of the letters ab and a vertical bar cursor inside a rectangle and is the fourth icon in the Toolbox.) A rectangle with eight small squares, called **sizing handles**, appears at the center of the form. See Figure 2-3.

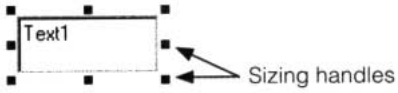


FIGURE 2-3 *A Text Box with Sizing Handles*

2. Click anywhere on the form outside the rectangle to remove the handles.
3. Click on the rectangle to restore the handles. An object showing its handles is (said to be) **selected**. A selected object can have its size altered, location changed, and other properties modified.
4. Move the mouse arrow to the handle in the center of the right side of the text box. The cursor should change to a double arrow (↔). Hold down the left mouse button, and move the mouse to the right. The text box is stretched to the right. Similarly, grabbing the text box by one of the other handles and moving the mouse stretches the text box in another direction. For instance, you use the handle in the upper-left corner to stretch the text box up and to the left. Handles also can be used to make the text box smaller.
5. Move the mouse arrow to any point of the text box other than a handle, hold down the left mouse button, and move the mouse. You can now drag the text box to a new location. Using Steps 4 and 5, you can place a text box of any size anywhere on the form.

   Note: The text box should now be selected; that is, its sizing handles should be showing. If not, click anywhere inside the text box to select it.
6. Press the delete key, Del, to remove the text box from the form. Step 7 gives an alternative way to place a text box of any size at any location on the form.
7. Click on the text box icon in the Toolbox. Then move the mouse pointer to any place on the form. (When over the form, the mouse pointer becomes a pair of crossed thin lines.) Hold down the left mouse button, and move the mouse on a diagonal to generate a rectangle. Release the mouse button to obtain a selected text box. You can now alter the size and location as before.

   Note: The text box should now be selected; that is, its sizing handles should be showing. If not, click anywhere inside the text box to select it.
8. Press F4 to activate the Properties window. (You can also activate the properties window by clicking on it or clicking on the Properties window icon in the Toolbar.) See Figure 2-4. The first line of the Properties window (called the **Object box**) reads "Text1 TextBox". Text1 is the current name of the text box. The two tabs permit you to view the list of properties either alphabetically or grouped into categories. Text boxes have 43 properties that can be grouped into 7 categories. Use the up- and down-arrow keys (or the up- and down-scroll arrows) to glance through the list. The left column gives the property and the right column gives the current setting of the property. We discuss four properties in this walkthrough.

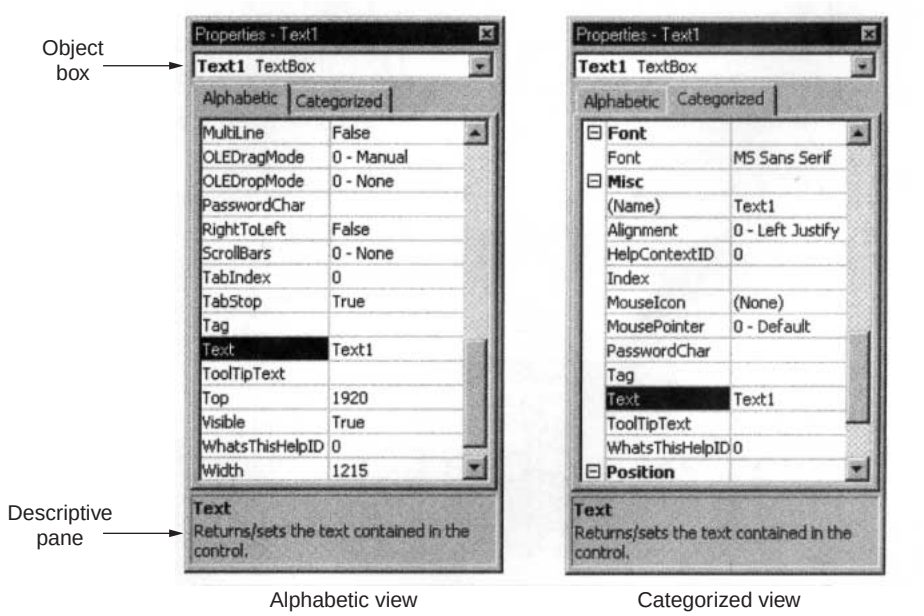


FIGURE 2-4 *Text Box Properties Window*

9. Move to the Text property with the up- and down-arrow keys. (Alternatively, scroll until the property is visible and click on the property.) The Text property is now highlighted. The Text property determines the words in the text box. Currently, the words are set to "Text1" in the **Settings box** on the right.
10. Type your first name. As you type, your name replaces "Text1" in both the Settings box and the text box. See Figure 2-5. (Alternatively, you could have clicked on the Settings box and edited its contents.)

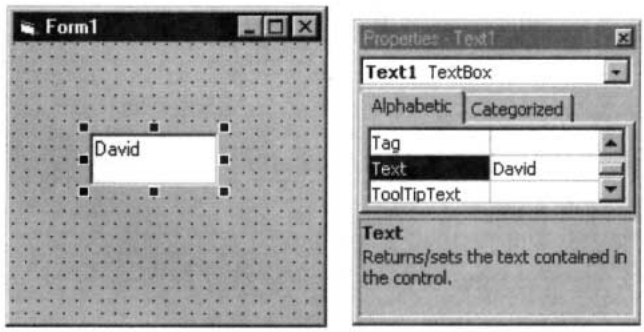


FIGURE 2-5 *Setting the Text Property to David*

11. Click at the beginning of your name in the Settings box and add your title, such as Mr., Ms., or The Honorable. (If you mistyped your name, you can easily correct it now.)
12. Press Shift+Ctrl+F to move to the first property that begins with the letter F. Now use the down-arrow key or the mouse to highlight the property ForeColor. The foreground color is the color of the text.
13. Click on the down arrow in the right part of the Settings box, and then click on the Palette tab to display a selection of colors. See Figure 2-6. Click on one of the solid colors, such as blue or red. Notice the change in the color of your name.

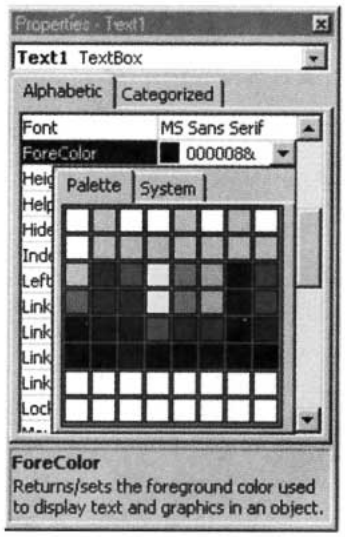


FIGURE 2-6 *Setting the ForeColor Property*

14. Highlight the Font property with a single click of the mouse. The current font is named MS Sans Serif.
15. Click on the ellipsis (...) box in the right part of the settings box to display a dialog box. See Figure 2-7. The three lists give the current name (MS Sans Serif), current style (Regular), and current size (8) of the font. You can change any of these attributes by clicking. Click on Bold in the style list,and click on 12 in the size list. Now click on the OK button to see your name displayed in a larger bold font.
16. Click on the text box and resize it to be about 3 inches wide and 1 inch high.

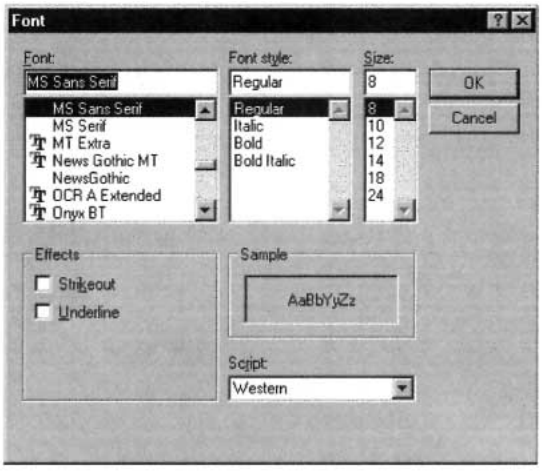


FIGURE 2-7 *The Font Dialog Box*

Visual Basic programs consist of three parts—interface, values of properties, and code. Our interface consists of a form with a single object, a text box. We have set a few properties for the text box—the text (namely, your name), the foreground color, the font style, and the font size. In Section 2.2, we see how to place code into a program. Visual Basic endows certain capabilities to programs that are independent of any code. We will now run the existing codeless program and experience these capabilities.

17. Press F5 to run the program. (Alternatively, a program can be run from the menu by pressing Alt/R/S or by clicking on the Start icon, the twelfth icon on the Toolbar.) Notice that the dots have disappeared from the form.
18. The cursor is at the beginning of your name. Press the End key to move the cursor to the end of your name. Now type in your last name, and then keep typing. Eventually, the words will scroll to the left.
19. Press Home to return to the beginning of the text. You have a full-fledged word processor at your disposal. You can place the cursor anywhere you like to add or delete text. You can drag the cursor across text to create a block, place a copy of the block in the clipboard with Ctrl+C, and then duplicate it anywhere with Ctrl+V.
20. To terminate the program, press Alt+F4. Alternatively, you can end a program by clicking on the End icon, the fourteenth icon on the Toolbar, or clicking on the form's close button.
21. Select the text box, activate the Properties window, select the MultiLine property, click on the down-arrow button, and finally click on True. The MultiLine property has been changed from False to True.
22. Run the program, and type in the text box. Notice that now words wrap around when the end of a line is reached. Also, text will scroll up when it reaches the bottom of the text box.
23. End the program.
24. Press Alt/F/V, or click on the Save Project icon to save the work done so far. A Save File As dialog box appears. See Figure 2-8. Visual Basic creates two disk files to store a program. The first, with the extension .frm, is entered into the Save File As dialog box and the second, with the extension .vbp, into a Save Project As dialog box. Visual Basic refers to programs as **projects**.

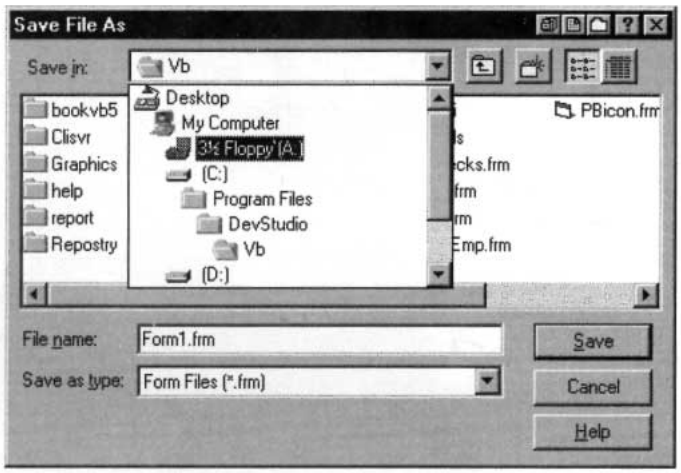


FIGURE 2-8 *The Save File As Dialog Box*

25. Type a file name, such as *testprog* into the "File name" box. The extension .frm automatically will be appended to the name. Do not press the Enter key yet. (Pressing the Enter key has the same effect as clicking Save.) The selection in the "Save in" box tells where your program will be saved. Alter it as desired. (***Suggestion:*** If you are using a computer in a campus computer lab, you probably should use a diskette to save your work. If so, place the diskette in a drive, say, the A drive, and select 3 1/2 Floppy (A:) in the "Save in" box.)

26. Click the Save button when you are ready to go on. (Alternatively, press Tab several times until the Save button is highlighted and then press Enter.) The Save Project As dialog box appears.
27. Type a file name into the File name box. You can use the same name, such as *testprog*, as before. Then proceed as in Steps 25 and 26. (The extension .vbp will be added.)
28. Press Alt/F/N to begin a new program. (As before, select Standard EXE.)
29. Place three text boxes on the form. (Move each text box out of the center of the form before creating the next.) Notice that they have the names Text1, Text2, and Text3.
30. Run the program. Notice that the cursor is in Text1. We say that Text1 has the **focus**. (This means that Text1 is the currently selected object and any keyboard actions will be sent directly to this object.) Any text typed will display in that text box.
31. Press Tab once. Now, Text2 has the focus. When you type, the characters appear in Text2.
32. Press Tab several times and then press Shift+Tab a few times. With Tab, the focus cycles through the objects on the form in the order the objects were created. With Shift+Tab, the focus cycles in the reverse order.
33. End the program.
34. Press Alt/F/O, or click on the Open Project icon to reload your first program. When a dialog box asks if you want to save your changes, click the No button or press N. An Open Project dialog box appears on the screen. Click on the Recent tab to see a list of the programs most recently opened or saved. Your first program and its location should appear at the top of the list. (***Note:*** You can also find any program by clicking on the Existing tab and using the dialog box to search for the program.)
35. Click on the name of your first program and then click on the Open button. Alternatively, double-click on the name. (You also have the option of typing the name into the File Name box and then clicking the Open button.)

A Command Button Walkthrough

1. Press Alt/F/N and double-click on Standard EXE to start a new program. There is no need to save anything.
2. Double-click on the command button icon to place a command button in the center of the form. (The rectangular-shaped command button icon is the sixth icon in the Toolbox.)
3. Activate the Properties window, highlight the Caption property, and type "Please Push Me". See Figure 2-9. Notice that the letters appear on the command button as they are typed. The button is too small.

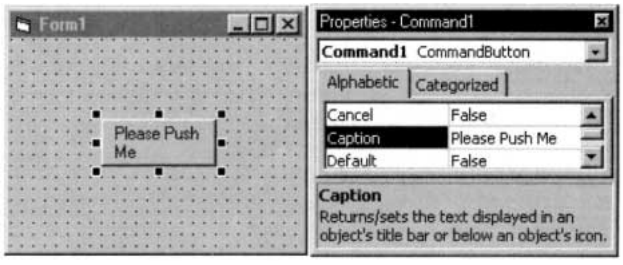


FIGURE 2-9 *Setting the Caption Property*

4. Click on the command button to select it, and then enlarge it to accommodate the phrase “Please Push Me” on one line.
5. Run the program, and click on the command button. The command button appears to move in and then out. In Section 2.2, we write code that is activated when a command button is pushed.
6. End the program, and select the command button.
7. From the Properties window, edit the Caption setting by inserting an ampersand (&) before the first letter, P. Notice that the ampersand does not show on the button. However, the letter following the ampersand is now underlined. See Figure 2-10. Pressing Alt+P while the program is running executes the same code as clicking the command button. Here, P is referred to as the **access key** for the command button. (The access key is always specified by the character following the ampersand.)

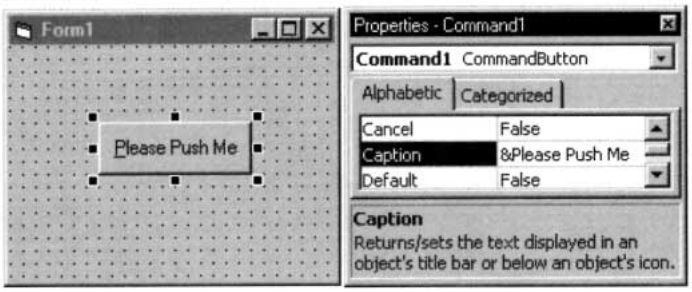


FIGURE 2-10 *Designating P as an Access Key*

A Label Walkthrough

1. Press Alt/F/N and double-click on Standard EXE to start a new program. There is no need to save anything.
2. Double-click on the label icon to place a label in the center of the form. (The label icon, a large letter A, is the third icon in the Toolbox.)
3. Activate the Properties window, highlight the Caption property, and type “Enter Your Phone Number”. Such a label would be placed next to a text box into which the user will enter a phone number.
4. Click on the label to select it, and then widen it until all words are on the same line.
5. Make the label narrower until the words occupy two lines.
6. Activate the Properties window, and double-click on the Alignment property. Double-click two more times and observe the label’s appearance. The combination of sizing and alignment permits you to design a label easily.
7. Run the program. Nothing happens, even if you click on the label. Labels just sit there. The user cannot change what a label displays unless you write code to allow the change.
8. End the program.

A Picture Box Walkthrough

1. Press Alt/F/N and double-click on Standard EXE to start a new program. There is no need to save anything.

2. Double-click on the picture box icon to place a picture box in the center of the form. (The picture box icon is the second icon in the Toolbox. It contains a picture of the sun shining over a desert.)
3. Enlarge the picture box.
4. Run the program. Nothing happens and nothing will, no matter what you do. Although picture boxes look like text boxes, you can't type in them. However, you can display text in them with statements discussed later in this section, you can draw lines and circles in them with statements discussed in Section 9, and you can insert pictures into them.
5. End the program and click the picture box to select it.
6. Activate the Properties window, and double-click on the Picture property. A Load Picture dialog box appears. See Figure 2-11.

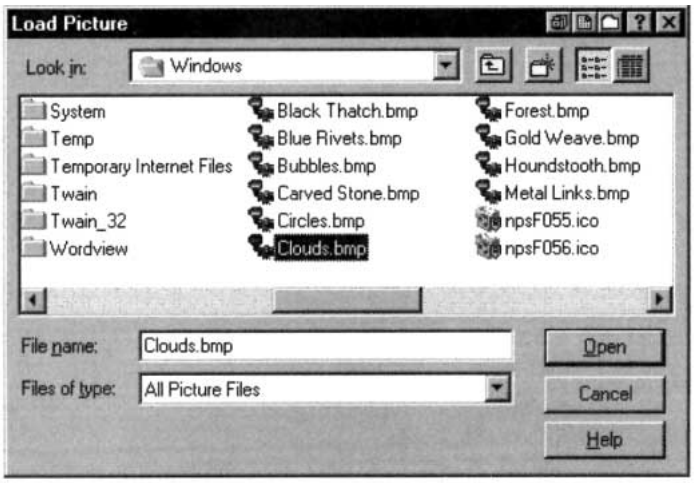


FIGURE 2-11 *The Load Picture Dialog Box*

7. Select the Windows folder and then double-click on one of the picture files. Good candidates are Clouds.bmp, shown in Figure 2-12, and Setup.bmp. (Also, the CD accompanying this textbook contains several picture files in the folder Pictures.)

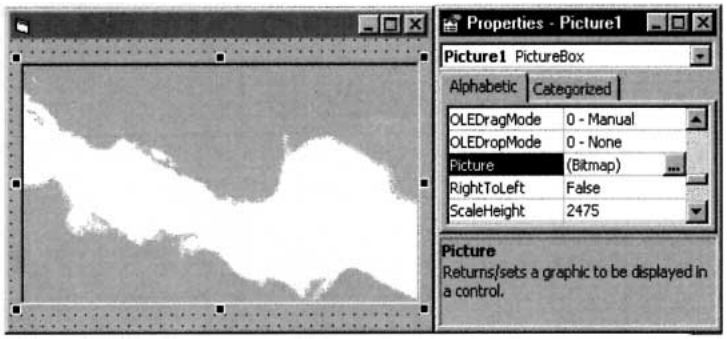


FIGURE 2-12 *A Picture Box Filled with the Clouds.bmp Picture*

8. Click on the picture box and press Del to remove the picture box.

COMMENTS

1. When selecting from a list, double-clicking has the same effect as clicking once and pressing Enter.
2. On a form, the Tab key cycles through the objects that can get the focus, and in a dialog box, it cycles through the items.
3. The form itself is also an object and has properties. For instance, you can change the text in the title bar with the Caption property. You can move the form by dragging the title bar of its Project Container window.
4. The name of an object is used in code to refer to the object. By default, objects are given names like Text1 and Text2. You can use the Properties window to change the Name property of an object to a more suggestive name. (The Name property is always the first property in the list of properties. An object's Name must start with a letter and can be a maximum of 40 characters. It can include numbers and underline (_) characters, but can't include punctuation or spaces.) Also, Microsoft recommends that each name begin with a three-letter prefix that identifies the type of the control. See the table below. Beginning with Section 2.2, we will use suggestive names and these prefixes whenever possible.

Object	Prefix	Example
command button	cmd	cmdComputeTotal
form	frm	frmPayroll
label	lbl	blInstructions
picture box	pic	picClouds
text box	txt	txtAddress

5. The Name and Caption properties of a command button are both initially set to something like Command1. However, changing one of these properties does not affect the setting of the other property. Similarly for the Name and Caption properties of forms and labels, and for the Name and Text properties of text boxes.
6. The color settings appear as strings of digits and letters preceded by &H and trailed with &. Don't concern yourself with the notation.
7. Here are some fine points on the use of the Properties window.
   (a) Press Shift+Ctrl+*letterkey* to highlight the first property that begins with that letter. Successive pressings highlight successive properties that begin with that letter.
   (b) To change the selected object from the Properties window, click on the down-arrow icon at the right of the Object box of the Properties window. Then select the new object from the drop-down list.
8. Some useful properties that have not been discussed are the following:
   (a) BorderStyle: Setting the BorderStyle to "0 – None" removes the border from an object.
   (b) Visible: Setting the Visible property to False hides an object when the program is run. The object can be made to reappear with code.
   (c) BackColor: Specifies the background color for a text box, label, picture box, or form. Also specifies the background color for a command button having the Style property set to "1 – Graphical." (Such a command button can display a picture.)
   (d) BackStyle: The BackStyle property of a label is opaque by default. The rectangular region associated with the label is filled with the label's back-

ground color and caption. Setting the background style of a label to transparent causes whatever is behind the label to remain visible; the background color of the label essentially becomes "see through."

(e) Font: Can be set to any of Windows' fonts, such as Courier and Times New Roman. Two unusual fonts are Symbol and Wingdings. For instance, with the Wingdings font, pressing the keys for %, &, ', and J yield a bell, a book, a candle, and a smiling face, respectively. To view the character sets for the different Windows' fonts, click on the Start button, and successively select Programs, Accessories, and Character Map. Then click on Character Map or press the Enter key. After selecting a font, hold down the left mouse button on any character to enlarge it and obtain the keystroke that produces that character.

9. When you click on a property in the Properties window, a description of the property appears just below the window. Additional information about many of the properties can be found in Appendix C. With the Learning, Professional, and Enterprise Editions of VB6.0 you can obtain very detailed (and somewhat advanced) information about a property by clicking on the property and pressing F1 for Help.
10. Most properties can be set or altered with code as the program is running instead of being preset from the Properties window. For instance, a command button can be made to disappear with a line such as Command1.Visible = False. See Section 2.2 for details.
11. The BorderStyle and MultiLine properties of a text box can be set only from the Properties window. You cannot alter them during run time.
12. Of the objects discussed in this section, only command buttons have true access keys.
13. If you inadvertently double-click an object in a form, a window containing two lines of text will appear. (The first line begins Private Sub.) This is a code window and is discussed in the next section. To remove this window, click on its Close button.
14. To enlarge (or decrease) the Project Container window, position the mouse cursor anywhere on the right or bottom edge and drag the mouse. To enlarge (or decrease) the form, select the form and drag one of its sizing handles. Alternatively, you can enlarge either the Project Container window or the form by clicking on its Maximize button.
15. We will always be selecting the Standard EXE icon from the New Project window.

2.2 VISUAL BASIC EVENTS

When a Visual Basic program is run, a form and its controls appear on the screen. Normally, nothing happens until the user takes an action, such as clicking a control or pressing the Tab key. Such an action is called an **event**.

The three steps to creating a Visual Basic program are as follows:

1. Create the interface; that is, generate, position, and size the objects.
2. Set properties; that is, set relevant properties for the objects.
3. Write the code that executes when the events occur.

This section is devoted to Step 3.

Code consists of statements that carry out tasks. Visual Basic has a repertoire of over 200 statements and we will use many of them in this text. In this section, we limit ourselves to statements that change properties of objects while a program is running.

Properties of an object are changed in code with statements of the form

```
objectName.property = setting
```

where *objectName* is the name of the form or a control, *property* is one of the properties of the object, and *setting* is a valid setting for that object. Such statements are called **assignment statements**. They assign values to properties. Here are three other assignment statements.

The statement

```
txtBox.Font.Size = 12
```

sets the size of the characters in the text box named txtBox to 12.

The statement

```
txtBox.Font.Bold = True
```

converts the characters in the text box to boldface.

The statement

```
txtBox.Text = ""
```

clears the contents of the text box; that is, it invokes the blank setting.

Most events are associated with objects. The event *clicking cmdButton* is different from the event *clicking picBox*. These two events are specified cmdButton_Click and picBox_Click. The statements to be executed when an event occurs are written in a block of code called an **event procedure**. The structure of an event procedure is

```
Private Sub objectName_event()
  statements
End Sub
```

The word Sub in the first line signals the beginning of the event procedure, and the first line identifies the object and the event occurring to that object. The last line signals the termination of the event procedure. The statements to be executed appear between these two lines. (***Note:*** The word Private indicates that the event procedure cannot be invoked by an event from another form. This will not concern us until much later in the text. The word *Sub* is an abbreviation of *Subprogram*.) For instance, the event procedure

```
Private Sub cmdButton_Click()
  txtBox.Text = ""
End Sub
```

clears the contents of the text box when the command button is clicked.

An Event Procedure Walkthrough

The form in Figure 2-13, which contains a text box and a command button, will be used to demonstrate what event procedures are and how they are created. Three event procedures will be used to alter the appearance of a phrase that is typed into the text box. The event procedures are txtPhrase_LostFocus, txtPhrase_GotFocus, and cmdBold_Click.

Object	Property	Setting
frmWalkthrough	Caption	Demonstration
txtPhrase	Text	(blank)
cmdBold	Caption	Make Phrase Bold

FIGURE 2-13 *The Interface for the Event Procedure Walkthrough*

1. Create the interface in Figure 2-13. The Name properties of the form, text box, and command button should be set as shown in the Object column. The Caption property of the form should be set to Demonstration, the Text property of the text box should be made blank, and the Caption property of the command button should be set to Make Phrase Bold.
2. Double-click on the text box. A window, called the **Code window**, appears. See Figure 2-14. Just below the title bar are two drop-down list boxes. The left box is called the **Object box** and the right box is called the **Procedure box**. (When you position the mouse pointer over one of these list boxes, its type appears.)

FIGURE 2-14 *A Code Window*

3. Click on the down-arrow button to the right of the Procedure box. The drop-down menu that appears contains a list of all possible event procedures associated with text boxes. See Figure 2-15.

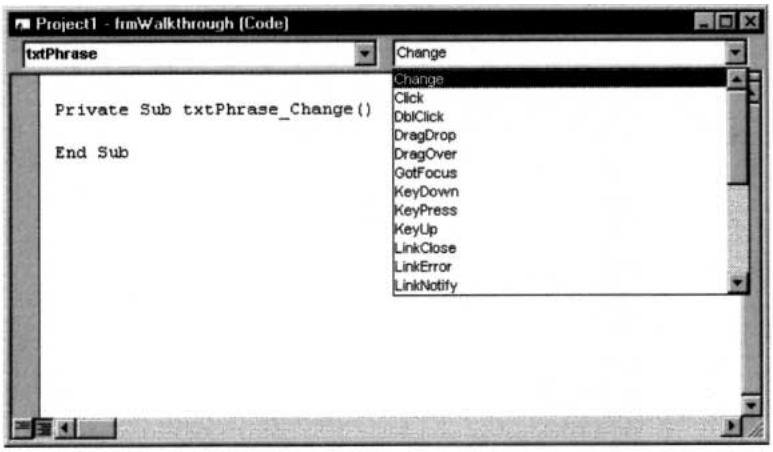


FIGURE 2-15 *Drop-Down Menu of Event Procedures*

4. Scroll down the list of event procedures and click on LostFocus. (LostFocus is the 14th event procedure.) The lines

```
Private Sub txtPhrase_LostFocus()

End Sub
```

appear in the code window with a blinking cursor poised at the beginning of the blank line.

5. Type the line

```
txtPhrase.Font.Size = 12
```

between the existing two lines. (We usually indent lines inside procedures.) (After you type each period, the editor displays a list containing possible choices of items to follow the period. See Figure 2-16. This feature is called "List Properties/Methods." In Figure 2-16, instead of typing the word "Size," you can double-click on "Size" in the displayed list or highlight the word "Size" and press Tab.) The screen appears as in Figure 2-17. We have now created an event procedure that is activated whenever the text box loses the focus.

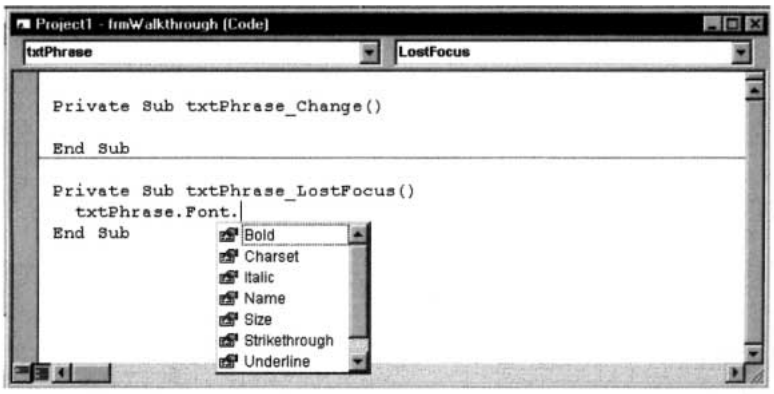


FIGURE 2-16 *A LostFocus Event Procedure*

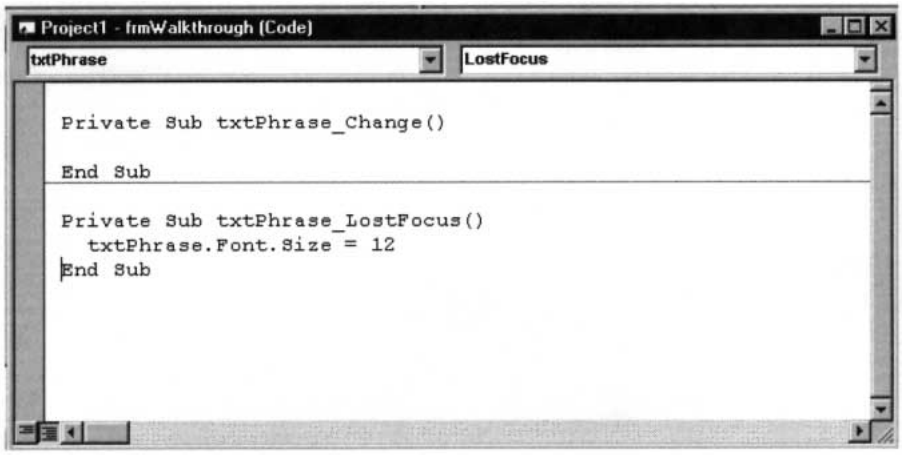


FIGURE 2-17 *A LostFocus Event Procedure*

6. Let's create another event procedure for the text box. Click on the down-arrow button to the right of the Procedure box, scroll up the list of event procedures, and click on GotFocus. Then type the lines

```
txtPhrase.Font.Size = 8txtPhrase.Font.Bold = False
```

between the existing two lines. See Figure 2-18.

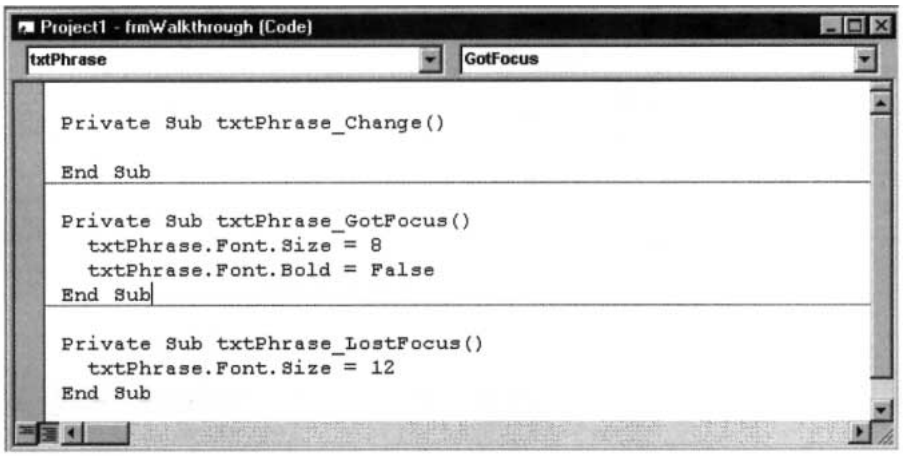


FIGURE 2-18 *A GotFocus Event Procedure*

7. The txtPhrase_Change event procedure in Figure 2-18 was not used and can be deleted. To delete the procedure, highlight it by dragging the mouse across the two lines of code, and then press the Del key.
8. Let's now create an event procedure for the command button. Click on the down-arrow button to the right of the Object box. The drop-down menu contains a list of the objects, along with a mysterious object called (General). See Figure 2-19. [We'll discuss (General) in the next section.]

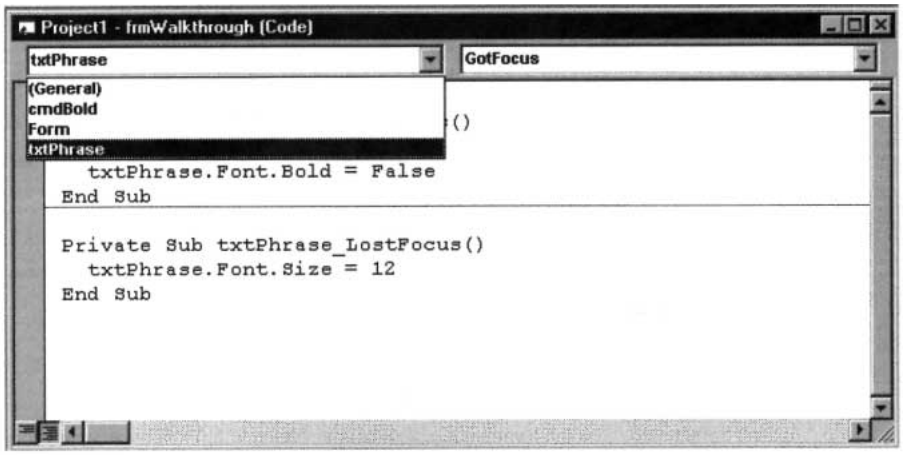


FIGURE 2-19 *List of Objects*

9. Click on cmdBold. The event procedure cmdBold_Click is displayed. Type in the line

```
txtPhrase.Font.Bold = True
```

The screen appears as in Figure 2-20, and the program is complete.

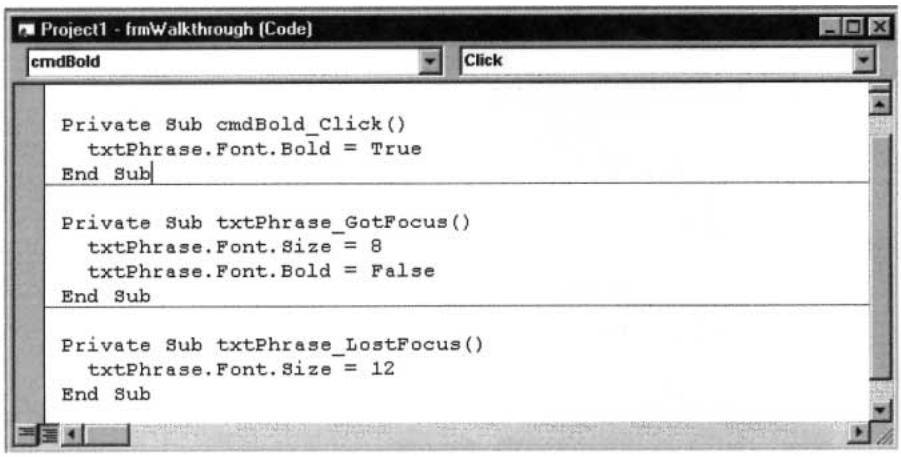


FIGURE 2-20 *The Three Event Procedures*

10. Now run the program by pressing F5.
11. Type something into the text box. In Figure 2-21, the words "Hello Friend" have been typed. (A text box has the focus whenever it is ready to accept typing; that is, whenever it contains a blinking cursor.)

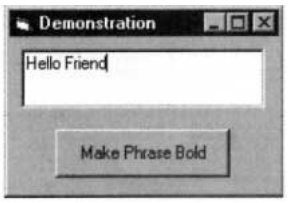


FIGURE 2-21 *Text Box Containing Input*

12. Press the Tab key. The contents of the text box will be enlarged as in Figure 2-22. When Tab was pressed, the text box lost the focus; that is, the event Lost-Focus happened to txtPhrase. Thus, the event procedure txtPhrase_ LostFocus was called, and the code inside the procedure was executed.

FIGURE 2-22 *Text Box After It Has Lost the Focus*

13. Click on the command button. This calls the event procedure cmd Bold_Click, which converts the text to boldface. See Figure 2-23.

FIGURE 2-23 *Text Box After the Command Button Has Been Clicked*

14. Click on the text box or press the Tab key to move the cursor (and, therefore, the focus) to the text box. This calls the event procedure txtPhrase_GotFocus, which restores the text to its original state.
15. You can repeat Steps 11 through 14 as many times as you like. When you are finished, end the program by pressing Alt+F4, clicking the End icon on the Toolbar, or clicking the Close button (X) on the form.

COMMENTS

1. To hide the code window, press the right mouse button and click on Hide. You can also hide it by clicking on the icon at the left side of the title bar and clicking on Close. To view a hidden code window, press Alt/View/Code. To hide a form, close its container. To view a hidden form, press Alt/View/Object.
2. The form is the default object in Visual Basic code. That is, code such as

   ```
   Form1.property = setting
   ```

   can be written as

   ```
   property = setting
   ```

   Also, event procedures associated with Form1 appear as

   ```
   Form_event()
   ```

   rather than

   ```
   Form1_event()
   ```

3. Another useful command is SetFocus. The statement

   ```
   object.SetFocus
   ```

   moves the focus to the object.
4. We have ended our programs by clicking the End icon or pressing Alt+F4. A more elegant technique is to create a command button, call it cmdQuit, with caption Quit and the event procedure:

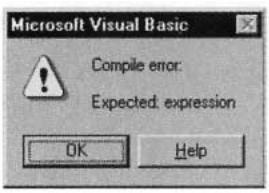
   ```
   Private Sub cmdQuit_Click()
     End
   End Sub
   ```

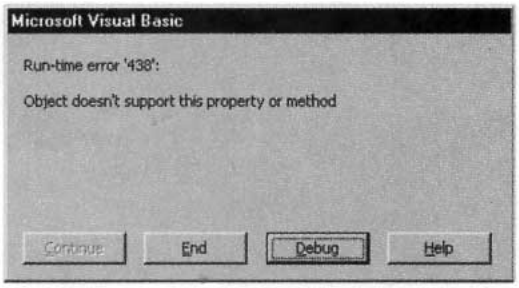
5. Certain words, such as Sub, End, and False, have special meanings in Visual Basic and are referred to as **keywords** or **reserved words**. The Visual Basic editor automatically capitalizes the first letter of a keyword and displays the word in blue.
6. Visual Basic can detect certain types of errors. For instance, consider the line

   ```
   txtPhrase.Font.Bold = False
   ```

   from the walkthrough. Suppose you neglected to type the word False to the right of the equal sign before leaving the line. Visual Basic would tell you something was missing by displaying the left message box at the top of page 66. (Also, the line would turn red.) On the other hand, suppose in the CmdBold_Click procedure you misspell the keyword “Bold” as “bolt.” You might notice something is wrong when the letter “b” is not capitalized. If not, you will certainly know about the problem when the program is run because Visual Basic will display the right message box at the top of page 66 when you click on the command

button. After you click on Debug, the line containing the offending word will be highlighted.





7. At design time, colors are selected from a palette. At run time, the eight most common colors can be assigned with the color constants vbBlack, vbRed, vbGreen, vbYellow, vbBlue, vbMagenta, vbCyan, and vbWhite. For instance, the statement

   ```
   picBox.BackColor = vbYellow
   ```

   gives picBox a yellow background.
8. For statements of the form *object.property* = *setting*, with properties Caption, Text, or Font.Name, the setting must be surrounded by quotes. (For instance, lblTwo.Caption = "Name", txtBox.Text = "Fore", and picBox.Font.Name = "Courier".) When the words True or False appear to the right of the equal sign, they should *not* be surrounded by quotation marks.
9. Code windows have many features of word processors. For instance, the operations cut, copy, paste, find, undo, and redo can be carried out with the sixth through eleventh icons of the Toolbar. These operations, and several others, also can be initiated from the Edit menu.
10. Names of existing event procedures associated with an object are *not* automatically changed when you rename the object. You must change them yourself and also must change any references to the object. Therefore, you should finalize the names of your objects before you put any code into their event procedures.
11. If you find the automatic List Properties/Methods feature distracting, you can turn it off by pressing Tools/Options, selecting the Editor page, and clicking on Auto List Members. If you do so, you can still display a list manually at the appropriate time by pressing Ctrl+J.
12. Earlier versions of Visual Basic used the property FontSize instead of Font.Size. Although Font.Size is preferred, FontSize is allowed for compatibility. Similarly, properties such as FontBold, FontItalic, and FontName have been included for compatibility with earlier versions of Visual Basic.
13. Assignment statements can be written preceded with the keyword Let. For instance, `txtBox.Text = "Hello"` also can be written `Let txtBox.Text = "Hello"`. Therefore, assignment statements are also known as Let statements.

2.3 NUMBERS

Much of the data processed by computers consists of numbers. In "computerese," numbers are often called **numeric constants**. This section discusses the operations that are performed with numbers and the ways numbers are displayed.

Arithmetic Operations

The five arithmetic operations in Visual Basic are addition, subtraction, multiplication, division, and exponentiation. (Because exponentiation is not as familiar as the others, it is reviewed in detail in Comment 10.) Addition, subtraction, and division are denoted in Visual Basic by the standard symbols +, –, and /, respectively. However, the notations for multiplication and exponentiation differ from the customary mathematical notations.

Mathematical Notation	Visual Basic Notation
$a \cdot b$ or $a \times b$	$a * b$
a^r	$a \wedge r$

(The asterisk [*] is the upper character of the 8 key. The caret [^] is the upper character of the 6 key.) ***Note:*** In this book, the proportional font used for text differs from the monospaced font used for programs. In the program font, the asterisk appears as a five-pointed star (*).

One way to show a number on the screen is to display it in a picture box. If *n* is a number, then the instruction

```
picBox.Print n
```

displays the number *n* in the picture box. If the picBox.Print instruction is followed by a combination of numbers and arithmetic operations, it carries out the operations and displays the result. Print is a reserved word and the Print operation is called a **method**. Another important method is Cls. The statement

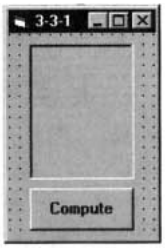
picBox.Cls

erases all text and graphics from the picture box picBox.

EXAMPLE 1

The following program applies each of the five arithmetic operations to the numbers 3 and 2. Notice that 3/2 is displayed in decimal form. Visual Basic never displays numbers as common fractions. ***Note 1:*** The star in the fifth and eighth lines is the computer font version of the asterisk. ***Note 2:*** The word "Run" in the phrasing [Run ...] indicates that F5 should be pressed to execute the program. ***Note 3:*** All programs appearing in examples and case studies are provided on the CD accompanying this book. See the discussion on the next to last page of the book for details.

Below is the form design and a table showing the names of the objects on the form and the settings, if any, for properties of these objects. This form design is also used in the discussion and examples in the remainder of this section.



Object	Property	Setting
frm3_3_1	Caption,	3-3-1
picResults		
cmdCompute	Caption	Compute

```
Private Sub cmdCompute_Click()
  picResults.Cls
  picResults.Print 3 + 2
  picResults.Print 3 - 2
  picResults.Print 3 * 2
  picResults.Print 3 / 2
  picResults.Print 3 ^ 2
  picResults.Print 2 * (3 + 4)
End Sub
```

[Run and then click the command button.]

Scientific Notation

Let us review powers of 10 and scientific notation. Our method of decimal notation is based on a systematic use of exponents.

$10^1 = 10$ $\qquad$ $10^{-1} = 1/10 = .1$
$10^2 = 100$ $\qquad$ $10^{-2} = .01$
$10^3 = 1000$ $\qquad$ $10^{-3} = .001$
.
.
.
$10^n = 1000...0$ $\qquad$ $10^{-n} = .001...01$
(n zeros) $\qquad$ (n digits)

Scientific notation provides a convenient way of writing numbers by using powers of 10 to stand for zeros. Numbers are written in the form $b \cdot 10^r$, where b is a number from 1 up to (but not including) 10, and r is an integer. For example, it is much more convenient to write the diameter of the sun (1,400,000,000 meters) in scientific notation: $1.4 \cdot 10^9$ meters. Similarly, rather than write .0000003 meters for the diameter of a bacterium, it is simpler to write $3 \cdot 10–7$ meters.

Any acceptable number can be entered into the computer in either standard or scientific notation. The form in which Visual Basic displays a number depends on many factors, with size being an important consideration. In Visual Basic, $b \cdot 10^r$ is usually written as bEr. (The letter E is an abbreviation for *exponent*.) The following forms of the numbers just mentioned are equivalent.

1.4 * 10^9	1.4E+09 1	4E+9	1.4E9	1400000000
3 * 10^–7	3E–07	3E–7	.0000003	

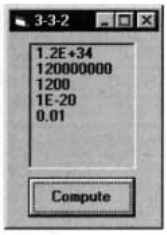
The computer displays r as a two-digit number, preceded by a plus sign if r is positive and a minus sign if r is negative.

EXAMPLE 2

The following program illustrates scientific notation. The computer's choice of whether to display a number in scientific or standard form depends on the magnitude of the number.

```
Private Sub cmdCompute_Click()
  picResults.Cls
  picResults.Print 1.2 * 10 ^ 34
  picResults.Print 1.2 * 10 ^ 8
  picResults.Print 1.2 * 10 ^ 3
  picResults.Print 10 ^ -20
  picResults.Print 10 ^ -2
End Sub
```

[Run and then click the command button.]



VARIABLES

In applied mathematics problems, quantities are referred to by names. For instance, consider the following high school algebra problem. "If a car travels at 50 miles per hour, how far will it travel in 14 hours? Also, how many hours are required to travel 410 miles?" The solution to this problem uses the well-known formula

$$\text{distance} = \text{speed} \times \text{time elapsed}$$

Here's how this problem would be solved with a computer program.

```
Private Sub cmdCompute_Click()
  picResults.Cls speed = 50
  timeElapsed = 14
  distance = speed * timeElapsed
  picResults.Print distance
  distance = 410
  timeElapsed = distance / speed
  picResults.Print timeElapsed
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
700
8.2
```

The third line of the event procedure sets the speed to 50, and the fourth line sets the time elapsed to 14. The fifth line multiplies the value for the speed by the value for the time elapsed and sets the distance to this product. The next line displays the answer to the first question. The three lines before the End Sub statement answer the second question in a similar manner.

The names *speed*, *timeElapsed*, and *distance*, which hold numbers, are referred to as **variables**. Consider the variable *timeElapsed*. In the fourth line, its value was set to 14. In the eighth line, its value was changed as the result of a computation. On the other hand, the variable *speed* had the same value, 50, throughout the program.

In general, a variable is a name that is used to refer to an item of data. The value assigned to the variable may change during the execution of the program. In Visual Basic, variable names can be up to 255 characters long, must begin with a letter, and can consist only of letters, digits, and underscores. (The shortest variable names consist of a single letter.) Visual Basic does not distinguish between uppercase and lowercase letters used in variable names. Some examples of variable names are *total*, *numberOfCars*, *taxRate_1999*, and *n*. As a convention, we write variable names in lowercase letters except for the first letters of additional words (as in *numberOfCars*).

If *var* is a variable and *num* is a constant, then the statement

```
var = num
```

assigns the number *num* to the variable *var*. (Such a statement is called an **assignment statement**.) Actually, the computer sets aside a location in memory with the name *var* and places the number *num* in it. The statement

```
picBox.Print var
```

looks into this memory location for the value of the variable and displays the value in the picture box.

A combination of constants, variables, and arithmetic operations that can be evaluated to yield a number is called a **numeric expression**. Expressions are evaluated by replacing each variable by its value and carrying out the arithmetic. Some examples of expressions are 2 * distance + 7, n + 1, and (a + b) / 3.

EXAMPLE 3

The following program displays the value of an expression.

```
Private Sub cmdCompute_Click()
  picResults.Cls
  a = 5
  b = 4
  picResults.Print a * (2 + b)
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
30
```

If *var* is a variable, then the statement

```
var = expression
```

first evaluates the expression on the right and *then* assigns its value to the variable. For instance, the event procedure in Example 3 can be written as

```
Private Sub cmdCompute_Click()
  picResults.Cls
  a = 5
  b = 4
  c = a * (2 + b)
  picResults.Print c
End Sub
```

The expression a * (2 + b) is evaluated to 30 and then this value is assigned to the variable *c*.

Because the expression on the right side of an assignment statement is evaluated *before* an assignment is made, a statement such as

```
n = n + 1
```

is meaningful. It first evaluates the expression on the right (that is, it adds 1 to the original value of the variable *n*), and then assigns this sum to the variable *n*. The effect is to increase the value of the variable *n* by 1. In terms of memory locations, the statement retrieves the value of *n* from *n*'s memory location, uses it to compute *n* + 1, and then places the sum back into *n*'s memory location.

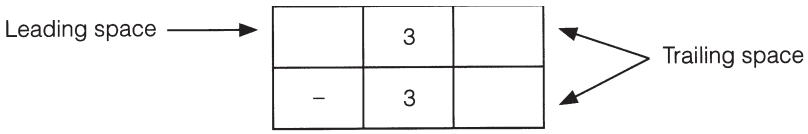
PRINT METHOD

Consider the following event procedure.

```
Private Sub cmdDisplay_Click()
  picResults.Cls
  picResults.Print 3
  picResults.Print -3
End Sub
```

[Run and then click the command button.]

Notice that the negative number –3 begins directly at the left margin, whereas the positive number 3 begins one space to the right. The Print method always displays nonnegative numbers with a leading space. The Print method also displays a trailing space after every number. Although the trailing spaces are not apparent here, we will soon see evidence of their presence.



The Print methods used so far display only one number per line. After displaying a number, the cursor moves to the leftmost position and down a line for the next display. Borrowing from typewriter terminology, we say that the computer performs a carriage return and a line feed after each number is displayed. The carriage return and line feed, however, can be suppressed by placing a semicolon at the end of the number.

EXAMPLE 4

The following program illustrates the use of semicolons in Print methods. The output reveals the presence of the space trailing each number. For instance, the space trailing –3 combines with the leading space of 99 to produce two spaces between the numbers.

```
Private Sub cmdDisplay_Click()
   picResults.Cls
   picResults.Print 3;
   picResults.Print -3;
   picResults.Print 99;
   picResults.Print 100
End Sub
```

[Run, and then click the command button.]

Semicolons can be used to display several numbers with one Print method. If *m*, *n*, and *r* are numbers, a line of the form

```
picBox.Print m; n; r
```

displays the three numbers, one after another, separated only by their leading and trailing spaces. For instance, the Print methods in preceding Example 4 can be replaced by the single line

```
picResults.Print 3; -3; 99; 100
```

COMMENTS

1. Numbers must not contain commas, dollar signs, or percent signs. Also, mixed numbers, such as 8 1/2, are not allowed.
2. Some people think of the equal sign (=) in an assignment statement as an arrow pointing to the left. This stresses the fact that the value on the right is assigned to the variable on the left.
3. Parentheses should be used when necessary to clarify the meaning of an expression. When there are no parentheses, the arithmetic operations are performed in the following order: (1) exponentiations; (2) multiplications and divisions; (3) additions and subtractions. In the event of ties, the leftmost operation is carried out first. Table 2-1 summarizes these rules.

TABLE 2-1
Level of Precedence for Arithmetic Operations

( )	Inner to outer, left to right
^	Left to right in expression
*/	Left to right in expression
+ –	Left to right in expression

4. Restricted keywords cannot be used as names of variables. For instance, the statements print = 99 and end = 99 are not valid. Some other common restricted keywords are Call, If, Let, Select, and Sub. If a keyword is used as a variable name, you will soon be warned that something is wrong. As soon as the cursor is moved from the line, an error message will appear, and the line will turn red. The use of some other keywords (such as Error, Height, Name, Rate, Time, Val, Width, and Year) as variable names does not trigger an immediate warning, but generates an error message when the program is run. Although there is a way to get Visual Basic to accept this last group of keywords as variable names, we will never use keywords as variable names. Most of the items in Appendix C, other than properties, are reserved words. You can tell immediately when you inadvertently use a reserved word as a variable in an assignment statement because Visual Basic automatically capitalizes the first letter of keywords. For instance, if you type "rate = 50" and press the Enter key, the line will change to "Rate = 50".
5. Grammatical errors, such as misspellings or incorrect punctuations, are called **syntax errors**. Certain types of syntax errors are spotted by the smart editor when they are entered, whereas others are not detected until the program is executed. When Visual Basic spots an error, it displays a dialog box. Some incorrect statements and their errors are given below.

Statement	Reason for Error
picBox.Primt 3	Misspelling of keyword
picBox.Print 2 +	No number follows the plus sign
9W = 5	9W is not a valid variable name

6. Errors detected while a program is running are called **run-time errors**. Although some run-time errors are due to improper syntax, others result from the inability of the computer to carry out the intended task. For instance, if the value of *numVar* is 0, then the statement

```
numVarInv = 1/numVar
```

interrupts the program with the run-time error "Division by zero." If the file DATA.TXT is not in the root folder of the C drive, then a statement that refers

to the file by the filespec "C:\DATA.TXT" produces the run-time error "File not found."

The dialog box generated by a run-time error states the type of error and has a row of four command buttons captioned Continue, End, Debug, and Help. If you click on the Debug command button, Visual Basic will highlight in yellow the line of code that caused the error. (***Note:*** After a run-time error occurs, the program is said to be in break mode. See the first page of Appendix D for a discussion of the three program modes.)

7. A third type of error is the so-called **logical error**. Such an error occurs when a program does not perform the way it was intended. For instance, the line

```
ave = firstNum + secondNum / 2
```

is syntactically correct. However, the missing parentheses in the first line are responsible for an incorrect value being generated. Appendix D discusses debugging tools that can be used to detect and correct logical errors.

8. The omission of the asterisk to denote multiplication is a common error. For instance, the expression a(b + c) is not valid. It should read a * (b + c).

9. The largest number that most of the numeric variables considered in this text can represent is 3.402823E+38. Attempting to generate larger values produces the message "Overflow." The numbers generated by the programs in this text usually have a maximum of seven digits.

10. *A Review of Exponents.* The expression 2^3 means $2 \cdot 2 \cdot 2$, the product of three 2's. The number 3 is called the **exponent**, and the number 2 is called the **base**. In general, if r is a positive integer and a is a number, then a is defined as follows:

$$a^r = \underbrace{a \cdot a \ldots a}_{r \text{ factors}}$$

The process of calculating a^r is called *raising a to the rth power*. Some other types of exponents are the following:

$a^{1/2} = \sqrt{a}$		$9^{1\backslash 2} = 3$
$a^{1/n} = \sqrt[n]{a}$	n positive integer	$16^{1/4} = 2$
$a^{m/n} = (\sqrt[n]{a})^m$	m, n positive integers	$8^{2/3} = (\sqrt[3]{8})^2 = 4$
$a^{-r} = 1/a^r$	$a \neq 0$	$10^{-2} = .01$

11. More than one statement can be placed on a single line of a program provided the statements are separated by colons. For instance, the code inside the event procedure in Example 3 can be written as

```
picResults.Cls: a = 5: b = 4: picResults.Print a * (2 + b)
```

In general, though, programs are much easier to follow if just one statement appears on each line. In this text, we almost always use single-statement lines.

12. When you first open a program that has been saved on disk, the Code window may not appear. If so, run and then terminate the program to see the Code window. To see the Form window, click on Object in the View menu or press Shift+F7. To return to the Code window, click on Code in the View window or press F7.

2.4 STRINGS

Two primary types of data can be processed by Visual Basic: numbers and strings. Sentences, phrases, words, letters of the alphabet, names, telephone numbers, addresses, and social security numbers are all examples of strings. Formally, a **string constant** is a sequence of characters that is treated as a single item. Strings can be assigned names with assignment statements, can be displayed with Print methods, and can be combined by an operation called concatenation (denoted by &).

VARIABLES AND STRINGS

A **string variable** is a name used to refer to a string. The allowable names of string variables are identical to those of numeric variables. The value of a string variable is assigned or altered with assignment statements and displayed with Print methods just like the value of a numeric variable.

EXAMPLE 1

The following code shows how assignment statements and Print are used with strings. The string variable *today* is assigned a value by the fourth line and this value is displayed by the fifth line. The quotation marks surrounding each string constant are not part of the constant and are not displayed by the Print method. (The form design for Examples 1 through 5 consists of a command button and picture box.)

```
Private Sub cmdButton_Click()
   picBox.Cls
   picBox.Print "hello"
   today = "9/17/99"
   picBox.Print today
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
hello
9/17/99
```

If *x, y, ..., z* are characters and *strVar1* is a string variable, then the statement

```
strVar1 = "xy...z"
```

assigns the string constant *xy...z* to the variable, and the statement

```
picBox.Print "xy...z"
```

or

```
picBox.Print strVar1
```

displays the string *xy...z* in a picture box. If *strVar2* is another string variable, then the statement

```
strVar2 = strVar1
```

assigns the value of the variable *strVar1* to the variable *strVar2*. (The value of *strVar1* will remain the same.) String constants used in assignment or picBox.Print statements must be surrounded by quotation marks, but string variables are never surrounded by quotation marks.

As with numbers, semicolons can be used with strings in picBox.Print statements to suppress carriage returns and line feeds. However, picBox.Print statements do not display leading or trailing spaces along with strings.

EXAMPLE 2

The following program illustrates the use of the assignment statement and Print method with text.

```
Private Sub cmdShow_Click()
   picOutput.Cls
   phrase = "win or lose that counts."
   picOutput.Print "It's not whether you "; phrase
   picOutput.Print "It's whether I "; phrase
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
It's not whether you win or lose that counts.
It's whether I win or lose that counts.
```

EXAMPLE 3

The following program has strings and numbers occurring together in a picBalance.Print statement.

```
Private Sub cmdCompute_Click()
   picBalance.Cls
   interestRate = 0.0655
   principal = 100
   phrase = "The balance after a year is"
   picBalance.Print phrase; (1 + interestRate) * principal
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
The balance after a year is 106.55
```

CONCATENATION

Two strings can be combined to form a new string consisting of the strings joined together. The joining operation is called **concatenation** and is represented by an ampersand (&). For instance, "good" & "bye" is "goodbye". A combination of strings and ampersands that can be evaluated to form a string is called a **string expression**. The assignment statement and Print method evaluate expressions before assigning them to variables or displaying them.

EXAMPLE 4

The following program illustrates concatenation.

```
Private Sub cmdDisplay_Click()
   picQuote.Cls
   quote1 = "The ballgame isn't over, "
   quote2 = "until it's over."
   quote = quote1 & quote2
   picQuote.Print quote & " Yogi Berra"
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
The ballgame isn't over, until it's over. Yogi Berra
```

Declaring Variable Types

So far, we have not distinguished between variables that hold strings and variables that hold numbers. There are several advantages to specifying the type of values, string or numeric, that can be assigned to a variable. A statement of the form

```
Dim variableName As String
```

specifies that only strings can be assigned to the named variable. A statement of the form

```
Dim variableName As Single
```

specifies that only numbers can be assigned to the named variable. The term Single derives from *single-precision real number*. After you type the space after the word "As," the editor displays a list of all the possible next words. In this text we use only a few of the items from this list.

A Dim statement is said to **declare** a variable. From now on we will declare all variables. However, all the programs will run correctly even if the Dim statements are omitted. Declaring variables at the beginning of each event procedure is regarded as good programming practice because it makes programs easier to read and helps prevent certain types of errors.

EXAMPLE 5

The following rewrite of Example 3 declares all variables.

```
Private Sub cmdCompute_Click()
  Dim interestRate As Single
  Dim principal As Single
  Dim phrase As String
  picBalance.Cls
  interestRate = 0.0655
  principal = 100
  phrase = "The balance after a year is"
  picBalance.Print phrase; (1 + interestRate) * principal
End Sub
```

Several Dim statements can be combined into one. For instance, the first three Dim statements of Example 5 can be replaced by

```
Dim interestRate As Single, principal As Single, phrase As String
```

Visual Basic actually has several different types of numeric variables. So far, we have used only single-precision numeric variables. Single-precision numeric variables can hold numbers of magnitude from as small as 1.4×10^{-45} to as large as 3.4×10^{38}. Another type of numeric variable, called **Integer**, can hold only whole numbers from –32768 to 32767. Integer-type variables are declared with a statement of the form

```
Dim intVar As Integer
```

The Integer data type uses less memory than the Single data type and statements using the Integer type execute faster. (This is only useful in programs with many calculations, such as the programs in later sections that use For...Next loops.) Of course, Integer variables are limited because they cannot hold decimals or large numbers. We will use Integer variables extensively with For...Next loops in Section 6 and occasionally when the data clearly consist of small whole numbers.

Other types of numeric variables are Long, Double, and Currency. We do not use them in this text. If you want to learn about them, consult Appendix C. Whenever we refer to a numeric variable without mentioning a type, we mean Single or Integer.

Using Text Boxes for Input and Output

The contents of a text box is always a string. Therefore, statements such as

```
strVar = txtBox.Text
```

and

```
txtBox.Text = strVar
```

can be used to assign the contents of the text box to the string variable *strVar* and vice versa.

Numbers are stored in text boxes as strings. Therefore, they should be converted to numbers before being assigned to numeric variables. If *str* is a string representation of a number, then

```
Val(str)
```

is that number. Conversely, if *num* is a number, then

```
Str(num)
```

is a string representation of the number. Therefore, statements such as

```
numVar = Val(txtBox.Text)
```

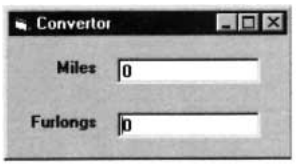
and

```
txtBox.Text = Str(numVar)
```

can be used to assign the contents of the text box to the numeric variable *numVar* and vice versa. ***Note:*** When a non-negative number is converted to a string with Str, its first character (but not its last character) is a blank space.

EXAMPLE 6

The following program converts miles to furlongs and vice versa. ***Note:*** A furlong is 1/8th of a mile.



Object	Property	Setting
frm3_4_6	Caption,	Convertor
lblMile	Caption	Miles
txtMile	Text	0
lblFurlong	Caption	Furlongs
txtFurlong	Text	0

The two text boxes have been named txtMile and txtFurlong. With the Event procedures shown, typing a number into a text box and pressing Tab results in the converted number being displayed in the other text box.

```
Private Sub txtMile_LostFocus()
  txtFurlong.Text = Str(8 * Val(txtMile.Text))
End Sub

Private Sub txtFurlong_LostFocus()
  txtMile.Text = Str(Val(txtFurlong.Text) / 8)
End Sub
```

ANSI Character Set

Each of the 47 different keys in the center typewriter portion of the keyboard can produce two characters, for a total of 94 characters. Adding 1 for the space character produced by the space bar makes 95 characters. These characters have numbers ranging from 32 to 126 associated with

them. These values, called the ANSI (or ASCII) values of the characters, are given in Appendix A. Table 2-2 shows a few of the values.

TABLE 2-2
A Few ANSI Values

32 (space)	48 0	66 B	122 z
33 !	49 1	90 Z	123 {
34 “	57 9	97 a	125 }
35 #	65 A	98 b	126 ~

Most of the best-known fonts, such as Ariel, Courier, Helvetica, and Times New Roman, are essentially governed by the ANSI standard, which assigns characters to the numbers from 32 to 255. Table 2-3 shows a few of the higher ANSI values.

TABLE 2-3
A Few Higher ANSI Values

162 ¢	177 ±	181 µ	190 $^3/_4$
169 ©	178 ^{2}	188 $^1/_4$	247 ÷
176 °	179 ^{3}	189 $^1/_2$	248 Ø

If *n* is a number between 32 and 255, then

```
Chr(n)
```

is the string consisting of the character with ANSI value *n*. If *str* is any string, then

```
Asc(str)
```

is the ANSI value of the first character of *str*. For instance, the statement

```
txtBox.Text = Chr(65)
```

displays the letter A in the text box and the statement

```
picBox.Print Asc(“Apple”)
```

displays the number 65 in the picture box.

Concatenation can be used with Chr to obtain strings using the higher ANSI characters. For instance, with one of the fonts that conforms to the ANSI standard, the statement

```
txtBox.Text = “32” & Chr(176) & ” Fahrenheit”
```

displays 32° Fahrenheit in the text box.

The KeyPress Event Procedure

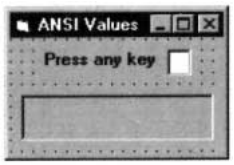
When a text box has the focus and the user presses a key, the KeyPress event procedure identifies the key pressed. When a key is pressed, the event procedure assigns the ANSI value of the key to an Integer variable called *KeyAscii*. The general form of the procedure is

```
Private Sub ControlName_KeyPress(KeyAscii As Integer)
  statements
End Sub
```

The statements usually involve the variable *KeyAscii*. Also, a character does not appear in the text box until End Sub is reached. At that time, the character with ANSI value *KeyAscii* is displayed.

EXAMPLE 7

The following program allows the user to determine the ANSI values of the standard (typewriter) keys of the keyboard. The statement txtCharacter. Text = “” removes any previously typed character from the text box.



Object	Property	Setting
frm3_4_7	Caption,	ANSI Values
lblPress	Caption	Press any key
txtCharacter	Text	(blank)
picOutput		

```
Private Sub txtCharacter_KeyPress(KeyAscii As Integer)
  txtCharacter.Text = ""
  picOutput.Cls
  picOutput.Print Chr(KeyAscii);
  " has ANSI value"; KeyAscii
End Sub
```

[Run, and then press a key. For instance, if A is pressed, the following is displayed in the picture box.]

```
A has ANSI value 65
```

The KeyPress event procedure can alter the character typed into the text box. For instance, if the statement

```
KeyAscii = 65
```

is placed in a KeyPress event procedure, the letter A is displayed when any standard key is pressed. In Section 4, we use a decision structure to prevent the user from typing unwanted keys. For instance, if we want the user to enter a number into a text box, we can intercept and discard any key presses that are not digits. The statement

```
KeyAscii = 0
```

placed in a KeyPress event procedure discards the key pressed. Finally, a program can be made more friendly by letting the Enter key (ANSI value 13) move the focus in the same way that the Tab key moves the focus. This requires having a KeyPress event procedure for each object that is to respond to the Enter key and then setting the focus to the next object when the value of *KeyAscii* is 13.

COMMENTS

1. The string "" contains no characters and is called the **null string** or the **empty string**. It is different than the string containing a single space (" "). String variables that have not been assigned values initially have "" as their default values. (Numeric variables have default value 0.)
2. The statement picBox.Print, with no string or number, simply skips a line in the picture box.
3. Assigning a string value to a numeric variable can result in the error message "Type mismatch."
4. In Visual Basic 6.0, the maximum allowable number of characters in a string is approximately 2 billion.
5. The quotation-mark character (") can be placed into a string constant by using Chr(34). For example, after the statement

   ```
   txtBox.Text = "George " & Chr(34) & "Babe" & Chr(34) & " Ruth"
   ```

   is executed, the text box contains

   ```
   George "Babe" Ruth
   ```

6. Most major programming languages require that all variables be declared before they can be used. Although declaring variables with Dim statements is optional in Visual Basic, you can tell Visual Basic to make declaration mandatory. The steps are as follows:
   (a) From any code window, click on the down-arrow to the right of the Object box and click on (General).
   (b) Type

   ```
   Option Explicit
   ```

   and press Enter.

   Then, if you use a variable without first declaring it in a Dim statement, the message "Variable not defined" will appear as soon as you attempt to run the program. One big advantage of using Option Explicit is that mistypings of variable names will be detected. Otherwise, malfunctions due to typing errors are often difficult to detect.
7. You can have Visual Basic automatically place Option Explicit in every program you write. The steps are as follows:
   (a) Press Alt/T/O and click on the Editor tab to invoke the editor options.
   (b) If the square to the left of "Require Variable Declaration" does not contain a check mark, click on the square and press the OK button.
8. Variables that are not (explicitly) declared with Dim statements are said to be **implicitly declared**. Such variables, which have a data type called Variant, can hold strings, numbers, and several other kinds of information.
9. You can display the type of a variable with the following steps—position the cursor over the word, press the right mouse button, and click on Quick Info.
10. Val can be applied to strings containing nonnumeric characters. If the beginning of the string *str* represents a number, then Val(*str*) is that number; otherwise, it is 0. For instance, Val("123Blastoff") is 123, and Val("ab3") is 0.
11. The KeyPress event also applies to command buttons and picture boxes.
12. Concatenation of strings also can be represented by a plus sign (+). However, restricting the plus sign to operations on numbers eliminates ambiguity and provides self-documenting code.
13. If Val is omitted from the statement

    ```
    numVar = Val(txtBox.Text)
    ```

    or Str is omitted from the statement

    ```
    txtBox.Text = Str(numVar)
    ```

    Visual Basic does not complain, but simply makes the conversion for you. However, errors can arise from omitting Val and Str. For instance, if the contents of txtBox1.Text is 34 and the contents of txtBox2.Text is 56, then the statement

    ```
    numVar = txtBox1.Text + txtBox2.Text
    ```

    assigns the number 3456 rather than 90 to *numVar*. (This is because Visual Basic does not perform the conversion until just before the assignment.) If txtBox1 is empty, then the statement

    ```
    3 * txtBox1.Text
    ```

    will stop the program and produce the error message "Type mismatch." We follow the standards of good programming practice by always using Val and Str to convert values between text boxes and numeric variables. Similar considerations apply to conversions involving label captions.

14. Variable names should describe the role of the variable. Also, some programmers use a prefix, such as sng or str, to identify the type of a variable. For example, they would use names like sngInterestRate and strFirstName.

2.5 INPUT AND OUTPUT

So far we have relied on assignment statements to assign values to variables. Data also can be stored in files and accessed through Input # statements, or data can be supplied by the user in a text box or input box. The Print method, with a little help from commas and the Tab function, can spread out and align the display of data in a picture box or on a printer. Message boxes grab the user's attention and display temporary messages. Comment statements allow the programmer to document all aspects of a program, including a description of the input used and the output to be produced.

Reading Data from Files

In Section 1, we saw how to create data files with Windows' Notepad. (As a rule of thumb, and simply as a matter of style, we enclose each string in quotation marks.) A file can have either one item per line or many items (separated by commas) can be listed on the same line. Usually, related items are grouped together on a line. For instance, if a file consisted of payroll information, each line would contain the name of a person, that person's hourly wage, and the number of hours that person worked during the week, as shown in Figure 2-24.

```
"Mike Jones", 7.35, 35
"John Smith", 6.75, 33
```

FIGURE 2-24 *Contents of STAFF.TXT*

The items of data will be assigned to variables one at a time in the order they appear in the file. That is, "Mike Jones" will be the first value assigned to a variable. After all the items from the first line have been assigned to variables, subsequent requests for values will be read from the next line.

Data stored in a file can be read in order (that is, sequentially) and assigned to variables with the following steps.

1. Choose a number from 1 to 255 to be the **reference number** for the file.
2. Execute the statement

   ```
   Open "filespec" For Input As #n
   ```

   where *n* is the reference number. This procedure is referred to as **Opening a file for input**. It establishes a communications link between the computer and the disk drive for reading data *from* the disk. Data then can be input from the specified file and assigned to variables in the program.
3. Read items of data in order, one at a time, from the file with Input # statements. The statement

   ```
   Input #n, var
   ```

   causes the program to look in the file for the next available item of data and assign it to the variable *var*. In the file, individual items are separated by commas or line breaks. The variable in the Input # statement should be the same type (that is, string versus numeric) as the data to be assigned to it from the file.
4. After the desired items have been read from the file, close the file with the statement

   ```
   Close #n
   ```

EXAMPLE 1

Write a program that uses a file for input and produces the same output as the following code. (The form design for all examples in this section consists of a command button and a picture box.)

```
Private Sub cmdDisplay_Click()
   Dim houseNumber As Single, street As String
   picAddress.Cls
   houseNumber = 1600
   street = "Pennsylvania Ave."
   picAddress.Print "The White House is located at"; houseNumber; street
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
The White House is located at 1600 Pennsylvania Ave.
```

SOLUTION:

Use Windows' Notepad to create the file DATA.TXT containing the following two lines:

1600
"Pennsylvania Ave."

In the following code, the fifth line looks for the first item of data, 1600, and assigns it to the numeric variable *houseNumber*. (Visual Basic records that this piece of data has been used.) The sixth line looks for the next available item of data, "Pennsylvania Ave.", and assigns it to the string variable *street*. ***Note:*** You will have to alter the Open statement in the fourth line to tell it where the file DATA.TXT is located. For instance, if the file is in the root directory (that is, folder) of a diskette in drive A, then the line should read `Open "A:\DATA.TXT" For Input As #1`. If the file is located in the subdirectory (that is, folder) VB6 of the C drive, then the statement should be changed to `Open "C:\VB6\DATA.TXT" For Input As #1`. See Comment 1 for another option.

```
Private Sub cmdReadFile_Click()
   Dim houseNumber As Single, street As String
   picAddress.Cls
   Open "DATA.TXT" For Input As #1
   Input #1, houseNumber
   Input #1, street
   picAddress.Print "The White House is located at"; houseNumber; street
   Close #1
End Sub
```

A single Input # statement can assign values to several different variables. For instance, the two Input # statements in the solution of Example 1 can be replaced by the single statement

```
Input #1, houseNumber, street
```

In general, a statement of the form

```
Input #n, var1, var2, ..., varj
```

has the same effect as the sequence of statements

```
Input #n, var1
Input #n, var2
 .
 .
 .
Input #n, varj
```

EXAMPLE 2

The following program uses the file STAFF.TXT in Figure 2-24 to compute weekly pay. Notice that the variables in the Input # statement are the same types (String, Single, Single) as the constants in each line of the file.

```
Private Sub cmdCompute_Click()
   Dim nom As String, wage As Single, hrs As Single
   picPay.Cls
   Open "STAFF.TXT" For Input As #1
   Input #1, nom, wage, hrs
   picPay.Print nom; hrs * wage
   Input #1, nom, wage, hrs
   picPay.Print nom; hrs * wage
   Close #1
End Sub
```

[Run, and then click the command button. The following will be displayed in the picture box.]

```
Mike Jones 257.25
John Smith 222.75
```

In certain situations, we must read the data in a file more than once. This is accomplished by closing the file and reopening it. After a file is closed and then reopened, subsequent Input # statements begin reading from the first entry of the file.

EXAMPLE 3

The following program takes the average annual amounts of money spent by single-person households for several categories and converts these amounts to percentages. The data are read once to compute the total amount of money spent and then read again to calculate the percentage for each category. ***Note:*** These figures were compiled for the year 1995 by the Bureau of Labor Statistics.

COSTS.TXT consists of the following four lines:

"Transportation", 3887
"Housing", 7643
"Food", 3017
"Other", 7804

```
Private Sub cmdCompute_Click()
   Dim total As Single, category As String, amount As Single
   Open "COSTS.TXT" For Input As #1
   picPercent.Cls
   total = 0
   Input #1, category, amount
   total = total + amount
   Input #1, category, amount
   total = total + amount
   Input #1, category, amount
   total = total + amount
   Input #1, category, amount
   total = total + amount
   Close #1
   Open "COSTS.TXT" For Input As #1
   Input #1, category, amount
   picPercent.Print category; amount / total
   Input #1, category, amount
   picPercent.Print category; amount / total
   Input #1, category, amount
```

```
   picPercent.Print category; amount / total
   Input #1, category, amount
   picPercent.Print category; amount / total
   Close #1
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
Transportation 0.1739072
Housing 0.3419534
Food 0.1349828
Other 0.3491566
```

INPUT FROM AN INPUT BOX

Normally, a text box is used to obtain input described by a label. Sometimes, we want just one piece of input and would rather not have a text box and label stay on the screen forever. The problem can be solved with an **input box**. When a statement of the form

```
stringVar = InputBox(prompt, title)
```

is executed, an input box similar to the one shown in Figure 2-25 pops up on the screen. After the user types a response into the text box at the bottom of the screen and presses Enter (or clicks OK), the response is assigned to the string variable. The *title* argument is optional and gives the caption to appear in the Title bar. The *prompt* argument is a string that tells the user what information to type into the text box.

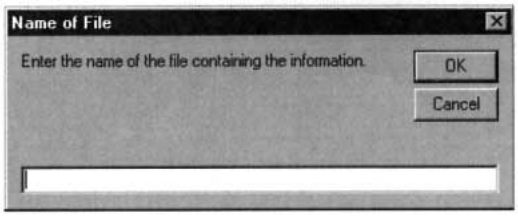


FIGURE 2-25 *Sample Input Box*

When you type the parenthesis following the word InputBox, the editor displays a line containing the general form of the InputBox statement. See Figure 2-26. This feature, which was added in Visual Basic 5.0, is called **Quick Info**. Optional parameters are surrounded by brackets. All the parameters in the general form of the InputBox statement are optional except for *prompt*.

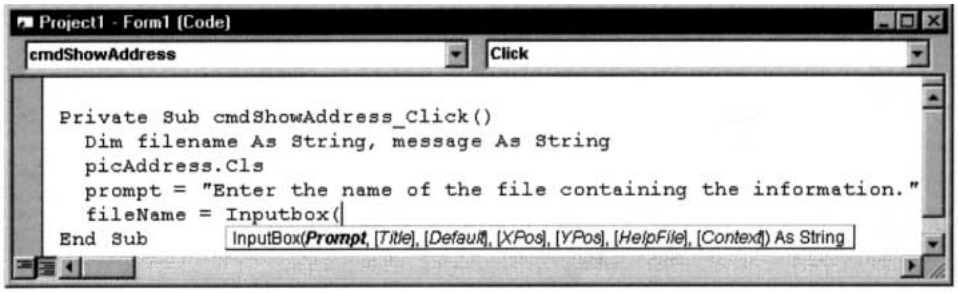


FIGURE 2-26 *Quick Info Feature*

EXAMPLE 4

In the following solution to Example 1, the file name is provided by the user in an input box.

```
Private Sub cmdDisplay_Click()
   Dim fileName As String, prompt As String, title As String
   Dim houseNumber As Single, street As String
   picAddress.Cls
   prompt = "Enter the name of the file containing the information."
   title = "Name of File"
   fileName = InputBox(prompt, title)
   Open fileName For Input As #1
   Input #1, houseNumber
   Input #1, street
   picAddress.Print "The White House is located at"; houseNumber; street
   Close #1
End Sub
```

[Run, and then click the command button. The input box of Figure 2-25 appears on the screen. Type DATA.TXT (possibly preceded with a path) into the input box and click on OK. The input box disappears and the following appears in the picture box.]

```
The White House is located at 1600 Pennsylvania Ave.
```

The response typed into an input box is treated as a single string value, no matter what is typed. (Quotation marks are not needed and, if included, are considered as part of the string.) Numeric data typed into an input box should be converted to a number with Val before it is assigned to a numeric variable or used in a calculation.

FORMATTING OUTPUT WITH PRINT ZONES

Each line in a picture box can be thought of as being subdivided into zones, as shown in Figure 2-27. Each zone contains 14 positions, where the width of a position is the average width of the characters in the font.

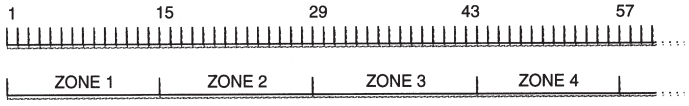


FIGURE 2-27 *Print Zones*

We have seen that when the Print method is followed by several items separated by semicolons, the items are displayed one after another. When commas are used instead of semicolons, the items are displayed in consecutive zones. For instance, if the Font property of picBox is set to Courier, when the motto of the state of Alaska is displayed with the statements

```
picBox.Print "North", "to", "the", "future."
picBox.Print "12345678901234567890123456789012345678901234567890"
```

the resulting picture box is

```
North          to             the            future.
12345678901234567890123456789012345678901234567890
```

where each word is in a separate print zone. This same output can be achieved with the code

```
Dim a As String, b As String, c As String, d As String
   a = "North"
   b = "to"
   c = "the"
   d = "future."
   picBox.Print a, b, c, d
picBox.Print "12345678901234567890123456789012345678901234567890"
```

EXAMPLE 5

The following program uses Print zones to organize expenses for public and private schools into columns of a table. The data represent the average expenses for 1995–96. (The Font setting for picTable is the default font MS Sans Serif.)

```
Private Sub cmdDisplay_Click()
   picTable.Cls
   picTable.Print " ", "Pb 2-yr", "Pr 2-yr", "Pb 4-yr", "Pr 4-yr"
   picTable.Print
   picTable.Print "Tuit & Fees", 1387, 6350, 2860, 12432
   picTable.Print "Bks & Suppl", 577, 567, 591, 601
   picTable.Print "Board", 1752, 1796, 1721, 1845
   picTable.Print "Trans", 894, 902, 929, 863
   picTable.Print "Other Exp", 1142, 1220, 1348, 1169
   picTable.Print " ", "-------", "-------", "-------", "-------"
   picTable.Print "Total", 5752, 10835, 7449, 16910
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
                  Pb 2-yr     Pr 2-yr     Pb 4-yr     Pr 4-yr
Tuit & Fees         1387        6350        2860        12432
Bks & Suppl          577         567         591          601
Board               1752        1796        1721         1845
Trans                894         902         929          863
Other Exp           1142        1220        1348         1169
                   ------      ------      ------       ------
Total               5752        10835       7449         16910
```

TAB FUNCTION

If an item appearing in a Print statement is preceded by

```
Tab(n);
```

where *n* is a positive integer, that item will be displayed (if possible) beginning at the *n*th position of the line. (Exceptions are discussed in Comment 10.)

EXAMPLE 6

The following program uses the Tab function to organize data into columns of a table. The data represent the number of bachelor's degrees conferred (in units of 1000). (*Source:* National Center of Educational Statistics.)

```
   Private Sub cmdDisplay_Click()
   picTable.Cls
   picTable.Print Tab(10); "1970-71"; Tab(20); "1980-81"; Tab(30); "1990-91"
   picTable.Print
   picTable.Print "Male"; Tab(10); 476; Tab(20); 470; Tab(30); 490
```

```
   picTable.Print "Female"; Tab(10); 364; Tab(20); 465; Tab(30); 560
   picTable.Print "Total"; Tab(10); 840; Tab(20); 935; Tab(30); 1050
End Sub
```

[Run and then click the command button. The resulting picture box is shown.]

	1970-71	1980-81	1990-91
Male	476	470	490
Female	364	465	560
Total	840	935	1050

Using a Message Box for Output

Sometimes you want to grab the user's attention with a brief message such as "Correct" or "Nice try, but no cigar." You want this message only to appear on the screen until the user has read it. This mission is easily accomplished with a **message box** such as the one shown in Figure 2-28. When a statement of the form

```
MsgBox prompt, , title
```

is executed, where *prompt* and *title* are strings, a message box with *prompt* displayed and the title bar caption *title* appears, and stays on the screen until the user presses Enter or clicks OK. For instance, the statement MsgBox "Nice try, but no cigar.", , "Consolation" produces Figure 2-28. If you use double quotation marks ("") for *title*, the title bar will be blank.

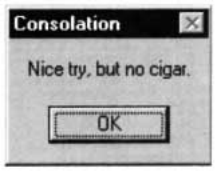


FIGURE 2-28 *Sample Message Box*

Line Continuation Character

Up to 1023 characters can be typed in a line of code. If you use a line with more characters than can fit in the window, Visual Basic scrolls the window toward the right as needed. However, most programmers prefer having lines that are no longer than the width of the code window. This can be achieved with the underscore character (_) preceded by a space. Make sure the underscore doesn't appear inside quotation marks though. For instance, the line

```
msg = "640K ought to be enough for anybody. (Bill Gates, 1981)"
```

can be written as

```
msg = "640K ought to be enough for " & _
"anybody. (Bill Gates, 1981)"
```

Output to the Printer

You print text on a sheet of paper in the printer in much the same way you display text in a picture box. Visual Basic treats the printer as an object named Printer. If *expr* is a string or numeric expression, then the statement

```
Printer.Print expr
```

sends *expr* to the printer in exactly the same way picBox.Print sends output to a picture box. You can use semicolons, commas for print zones, and Tab.

Font properties can be set with statements like

```
Printer.Font.Name = "Script"
Printer.Font.Bold = True
Printer.Font.Size = 12
```

Another useful printer command is

```
Printer.NewPage
```

which starts a new page.

Windows' print manager usually waits until an entire page has been completed before starting to print. To avoid losing information, execute the statement

```
Printer.EndDoc
```

when you are finished printing.

The statement

```
PrintForm
```

prints the content of the form.

INTERNAL DOCUMENTATION

Now that we have the capability to write more complicated programs, we must concern ourselves with program documentation. **Program documentation** is the inclusion of comments that specify the intent of the program, the purpose of the variables, the nature of the data in the files, and the tasks performed by individual portions of the program. To create a comment line, just begin the line with an apostrophe. Such a line is completely ignored when the program is executed. (The keyword Rem can be used instead of an apostrophe. Rem is an abbreviation of Remark.) Program documentation appears whenever the program is displayed or printed. Also, a line of code can be documented by adding an apostrophe, followed by the desired information, to the end of the line. Comments (also known as Rem statements) appear green on the screen.

EXAMPLE 7

Document the program in Example 2.

SOLUTION

In the following program, the first comment describes the entire program, the next three comments give the meanings of the variables, and the final comment describes the items in each line of the file.

```
Private Sub cmdCompute_Click()
   'Compute weekly pay
   Dim nom As String    'Employee name
   Dim wage As Single   'Hourly pay
   Dim hrs As Single    'Number of hours worked during week
   picPay.Cls
   Open "STAFF.TXT" For Input As #1
   'Get person's name, hourly pay, and hours worked
   Input #1, nom, wage, hrs
   picPay.Print nom; hrs * wage Input #1, nom, wage,
   hrs picPay.Print nom; hrs * wage
   Close #1
End Sub
```

Some of the benefits of documentation are as follows:

1. Other people can easily comprehend the program.
2. The program can be understood when read later.
3. Long programs are easier to read because the purposes of individual pieces can be determined at a glance.

COMMENTS

1. Visual Basic provides a convenient device for accessing a file that resides in the same folder as the (saved) program. After a program has been saved in a folder, the value of App.Path is the string containing the name of the folder. Therefore, if a program contains a line such as

   ```
   Open App.Path & "\DATA.TXT" For Input As #1
   ```

   Visual Basic will look for the file DATA.TXT in the folder containing the program.

   The programs from this book, as well as the data files they use, all are contained in the folder Programs on the CD accompanying this book. On the CD, App.Path is used in every Open statement. Therefore, even after you copy the contents of the Programs folder onto a hard drive or diskette, the programs will continue to execute properly without your having to alter any paths.
2. The text box and input box provide a whole new dimension to the capabilities of a program. The user, rather than the programmer, can provide the data to be processed.
3. A string used in a file does not have to be enclosed by quotation marks. The only exceptions are strings containing commas or leading and trailing spaces.
4. If an Input # statement looks for a string and finds a number, it will treat the number as a string. Suppose the first two entries in the file DATA.TXT are the numbers 2 and 3.

   ```
   Private Sub cmdButton_Click()
     Dim a As String, b As String
     picBox.Cls
     Open "DATA.TXT" For Input As #1
     Input #1, a, b
     picBox.Print a + b
     Close #1
   End Sub
   ```

   [Run and then click the command button. The following is displayed in the picture box.]

   ```
   23
   ```

5. If an Input # statement looks for a number and finds a string, the Input # statement will assign the value 0 to the numeric variable. For instance, suppose the first two entries in the file DATA.TXT are "ten" and 10. Then after the statement

   ```
   Input #1, num1, num2
   ```

   is executed, where um1 and *num2* are numeric variables, the values of these variables will be 0 and 10.

6. If all the data in a file have been read by Input # statements and another item is requested by an Input # statement, a box will appear displaying the message "Input past end of file."
7. Numeric data in a text box, input box, or file must be a constant. It *cannot* be a variable or an expression. For instance, num, 1 / 2, and 2 + 3 are not acceptable.
8. To skip a Print zone, just include two consecutive commas.
9. Print zones are usually employed to align information into columns. Since most fonts have proportionally-spaced characters, wide characters occupy more than one fixed-width column and narrow characters occupy less. The best and most predictable results are obtained when a fixed-pitch font (such as Courier) is used with print zones.
10. The Tab function cannot be used to move the cursor to the left. If the position specified in a Tab function is to the left of the current cursor position, the cursor will move to that position on the next line. For instance, the line

    ```
    picBox.Print "hello"; Tab(3); "good-bye"
    ```

    results in the output

    ```
    hello
      good-bye
    ```

11. The statement Close, without any reference number, closes all open files.
12. Windows allows you to alternate between Visual Basic and Notepad without exiting either application. To invoke Notepad with Windows, click on the Start button; successively select Programs, Accessories, and Notepad; and then click on Notepad or press the Enter key. Now that both Visual Basic and Notepad have been invoked, you can go from one application to the other by holding down the Alt key and repeatedly pressing the Tab key until the name of the other application appears. When the Alt key is released, the named application becomes active.

2.6 BUILT-IN FUNCTIONS

Visual Basic has a number of built-in functions that greatly extend its capability. These functions perform such varied tasks as taking the square root of a number, counting the number of characters in a string, and capitalizing letters. Functions associate with one or more values, called the *input*, a single value, called the *output*. The function is said to **return** the output value. The three functions considered in what follows have numeric input and output.

Numeric Functions: Sqr, Int, Round

The function Sqr calculates the square root of a number. The function Int finds the greatest integer less than or equal to a number. Therefore, Int discards the decimal part of positive numbers. The value of Round (*n*, *r*) is the number *n* rounded to *r* decimal places. The parameter *r* can be omitted. If so, *n* is rounded to a whole number. Some examples follow:

```
Sqr(9) is 3.              Int(2.7) is 2.      Round(2.7) is 3.
   Sqr(0) is 0.           Int(3) is 3.        Round(2.317, 2) is 2.32.
Sqr(2) is 1.414214.       Int(-2.7) is -3.    Round(2.317, 1) is 2.3.
```

The terms inside the parentheses can be either numbers (as shown), numeric variables, or numeric expressions. Expressions are first evaluated to produce the input.

EXAMPLE 1

The following program evaluates each of the functions for a specific input given by the value of the variable *n*.

```
Private Sub cmdEvaluate_Click()
   Dim n As Single, root As Single
   'Evaluate functions at a variable
   picResults.Cls
   n = 6.76
   root = Sqr(n)
   picResults.Print root; Int(n); Round(n,1)
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
2.6 6 6.8
```

EXAMPLE 2

The following program evaluates each of the preceding functions at an expression.

```
Private Sub cmdEvaluate_Click()
   Dim a As Single, b As Single
   'Evaluate functions at expressions
   picResults.Cls
   a = 2
   b = 3
   picResults.Print Sqr(5 * b + 1); Int(a ^ b); Round(a / b, 3)
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

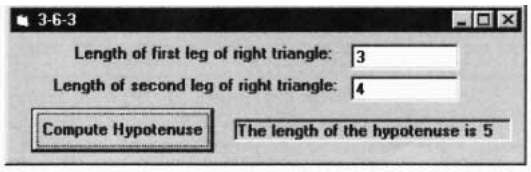
```
4 8 0.667
```

EXAMPLE 3

The following program shows an application of the Sqr function.

```
Private Sub cmdComputeHyp_Click()
   Dim leg1 As Single, leg2 As Single, hyp As Single
   'Find the length of the hypotenuse of a right triangle
   picHyp.Cls
   leg1 = Val(txtFirst.Text)
   leg2 = Val(txtSecond.Text)
   hyp = Sqr(leg1 ^ 2 + leg2 ^ 2)
   picHyp.Print "The length of the hypotenuse is"; hyp
End Sub
```

[Run, type 3 and 4 into the text boxes, and then click the command button.]



EXAMPLE 4

The following program shows how Int is used to carry out long division. When the integer *m* is divided into the integer *n* with long division, the result is a quotient and a remainder. See Figure 2-29.

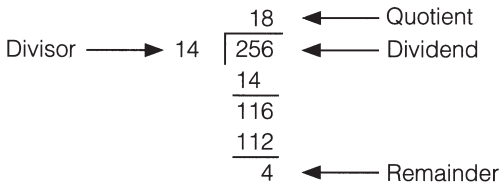
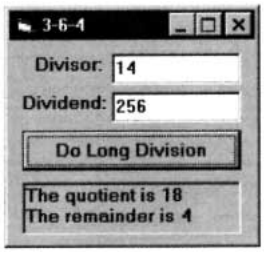


FIGURE 2-29 *Long Division*

```
Private Sub cmdDivide_Click()
   Dim divisor As Single, dividend As Single
   Dim quotient As Single, remainder As Single
   'Long division
   picResult.Cls
   divisor = Val(txtDivisor.Text)
   dividend = Val(txtDividend.Text)
   quotient = Int(dividend / divisor)
   remainder = dividend - quotient * divisor
   picResult.Print "The quotient is"; quotient
   picResult.Print "The remainder is"; remainder
End Sub
```

[Run, type 14 and 256 into the text boxes, and then click the command button.]



STRING FUNCTIONS: LEFT, MID, RIGHT, UCASE, TRIM

The functions Left, Mid, and Right are used to extract characters from the left end, middle, and right end of a string. Suppose *str* is a string and *m* and *n* are positive integers. Then Left(*str*, *n*) is the string consisting of the first *n* characters of *str* and Right(*str*, *n*) is the string consisting of the last *n* characters of *str*. Mid(*str*, *m*, *n*) is the string consisting of *n* characters of *str*, beginning with the *m*th character. UCase(*str*) is the string *str* with all of its lowercase letters capitalized. Trim(*str*) is the string *str* with all leading and trailing spaces removed. Some examples are as follows:

```
Left("fanatic", 3) is "fan".        Right("fanatic", 3) is "tic".
Left("12/15/99", 2) is "12".        Right("12/15/99", 2) is "99".
Mid("fanatic", 5, 1) is "t".        Mid("12/15/99", 4, 2) is "15".
UCase("Disk") is "DISK".            UCase("12three") is "12THREE".
Trim(" 1 2 ") is "1 2".             Trim("-12 ") is "-12".
```

The strings produced by Left, Mid, and Right are referred to as **substrings** of the strings from which they were formed. For instance, "fan" and "t" are substrings of "fanatic". The substring "fan" is said to begin at position 1 of "fanatic" and the substring "t" is said to begin at position 5.

Like the numeric functions discussed before, Left, Mid, Right, UCase, and Trim also can be evaluated for variables and expressions.

EXAMPLE 5

The following program evaluates the functions above for variables and expressions. Note that spaces are counted as characters.

```
Private Sub cmdEvaluate_Click()
   Dim str1 As String, str2 As String
   'Evaluate functions at variables and expressions.
   picResults.Cls
   str1 = "Quick as "
   str2 = "a wink"
   picResults.Print Left(str1, 7)
   picResults.Print Mid(str1 & str2, 7, 6)
   picResults.Print UCase(str1 & str2)
   picResults.Print "The average "; Right(str2, 4); " lasts .1 second."
   picResults.Print Trim(str1); str2
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
Quick a
   as a w
   QUICK AS A WINK
   The average wink lasts .1 second.
Quick asa wink
```

STRING-RELATED NUMERIC FUNCTIONS: LEN, INSTR

The functions Len and InStr operate on strings but produce numbers. The function Len gives the number of characters in a string. The function InStr searches for the first occurrence of one string in another and gives the position at which the string is found. Suppose *str1* and *str2* are strings. The value of Len(*str1*) is the number of characters in *str1*. The value of InStr(*str1*, *str2*) is 0 if *str2* is not a substring of *str1*. Otherwise, its value is the first position of *str2* in *str1*. Some examples of Len and InStr follow:

```
Len("Shenandoah") is 10.          InStr("Shenandoah", "nand") is 4.
Len("Just a moment") is 13.       InStr("Just a moment", " ") is 5.
Len(" ") is 1.                    InStr("Croissant", "ist") is 0.
```

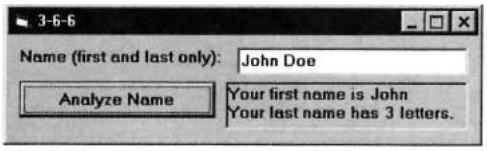
EXAMPLE 6

The following program evaluates functions at variables and expressions. The ninth line locates the position of the space separating the two names. The first name will end one position to the left of this position and the last name will consist of all but the first *n* characters of the full name.

```
Private Sub cmdAnalyze_Click()
   Dim nom As String   'Name
   Dim n As Integer    'Location of space
   Dim first As String  'First name
   Dim last As String  'Last name '
   Evaluate functions at variables and expressions.
   picResults.Cls
   nom = txtFullName.Text
   n = InStr(nom, " ")
   first = Left(nom, n - 1)
```

```
   last = Right(nom, Len(nom) - n)
   picResults.Print "Your first name is "; first
   picResults.Print "Your last name has"; Len(last); "letters."
End Sub
```

[Run, type John Doe into the text box, and then click the command button.]



FORMAT FUNCTIONS

The Format functions are used to display numbers and dates in familiar forms and to right-justify numbers. Here are some examples of how numbers are converted to strings with Format functions.

```
Function                              String Value
FormatNumber(12345.628, 1)            12,345.6
FormatCurrency(12345.628, 2)          $12,345.63
FormatPercent(.185, 2)                18.50%
```

The value of FormatNumber(*n*, *r*) is the string containing the number *n* rounded to *r* decimal places and displayed with commas every three digits to the left of the decimal point. The value of FormatCurrency(*n*, *r*) is the string consisting of a dollar sign followed by the value of FormatNumber(*n*, *r*). FormatCurrency uses the accountant's convention of using surrounding parentheses to denote negative amounts. The value of FormatPercent(*n*, *r*) is the string consisting of the number *n* displayed as a percent and rounded to *r* decimal places.

With all three functions, *r* can be omitted. If so, the number is rounded to 2 decimal places. Strings corresponding to numbers less than 1 in magnitude have a zero to the left of the decimal point. Also, *n* can be a either a number, a numeric expression, or even a string corresponding to a number.

```
Function                              String Value
FormatNumber(1 + Sqr(2), 3)           2.414
FormatCurrency(-1000)                 ($1,000.00)
FormatPercent(".005")                 0.50%
```

If *dateString* represents a date in a form such as "7-4-1999", "7-4-99", or "7/4/99", then the value of FormatDateTime(*dateString*, vbLongDate) is a string giving the date as Sunday, July 04, 1999.

```
Function                                  String Value
FormatDateTime("9-15-99", vbLongDate)     Wednesday, September 15, 1999
FormatDateTime("10-23-00", vbLongDate)    Monday, October 23, 2000
```

The value of Format(*expr*, "@@ . . . @"), where "@@ . . . @" is a string of *n* "at" symbols, is the string consisting of the value of *expr* right-justified in a field of *n* spaces. This function is used with fixed-width fonts, such as Courier or Terminal, to display columns of numbers so that the decimal points and commas are lined up or to display right-justified lists of words. The following examples use a string of 10 "at" symbols.

```
Function                                                  String Value
Format(1234567890, "@@@@@@@@@@")                          1234567890
Format(FormatNumber(1234.5), "@@@@@@@@@@")                1,234.50
Format(FormatNumber(12345.67), "@@@@@@@@@@")              12,345.67
Format(FormatCurrency(13580.17), "@@@@@@@@@@")            $13,580.17
```

EXAMPLE 7

The following program produces essentially the first two columns of the table in Example 5 of Section 2.5. However, Format is used to right-justify the expense categories and to align the numbers.

Object	Property	Setting
frmExpenses	Caption	Public 2-year College Expenses
cmdDisplay	Caption	Display Expenses
picTable	Font.Name	Courier

```
Private Sub cmdDisplay_Click()
   Dim fmt1 As String, fmt2 As String
   Dim col1 As String, col2 As String
   'Average expenses of commuter students (1995-96)
   picTable.Cls
   picTable.Print Tab(19); "Pb 2-yr"
   picTable.Print
   fmt1 = "@@@@@@@@@@@@@@@@@" '17 @ symbols
   fmt2 = "@@@@@@" '6 @ symbols
   col1 = Format("Tuition & Fees", fmt1)
   col2 = FormatNumber(1387, 0)
   col2 = Format(col2, fmt2)
   picTable.Print col1; Tab(19); col2
   col1 = Format("Books & Supplies", fmt1)
   col2 = FormatNumber(577, 0)
   col2 = Format(col2, fmt2)
   picTable.Print col1; Tab(19); col2
   col1 = Format("Board", fmt1)
   col2 = FormatNumber(1752, 0)
   col2 = Format(col2, fmt2)
   picTable.Print col1; Tab(19); col2
   col1 = Format("Transportation", fmt1)
   col2 = FormatNumber(894, 0)
   col2 = Format(col2, fmt2)
   picTable.Print col1; Tab(19); col2
   col1 = Format("Other Expenses", fmt1)
   col2 = FormatNumber(1142, 0)
   col2 = Format(col2, fmt2)
   picTable.Print col1; Tab(19); col2
   picTable.Print Tab(19); "———"
   col1 = Format("Total", fmt1)
   col2 = FormatNumber(5752, 0)
   col2 = Format(col2, fmt2)
   picTable.Print col1; Tab(19);
   col2
End Sub
```

[Run, and then click the command button.]

```
                        Pb 2-Yr
        Tuition & Fees  1,387
     Books & Supplies   577
                Board   1,752
       Transportation   894
       Other Expenses   1,142
                        -----
                Total   5,752
```

Generating Random Numbers: Rnd

Consider a specific collection of numbers. We say that a process selects a number at **random** from this collection if any number in the collection is just as likely to be selected as any other and the number cannot be predicted in advance. Some examples follow:

Collection	Process
1, 2, 3, 4, 5, 6	toss a die
0 or 1,	toss a coin: 0 = tails, 1 = heads
–1, 0, 1, . . . , 36	spin a roulette wheel (interpret –1 as 00)
1, 2, . . . , n	write numbers on slips of paper, pull one from hat
numbers from 0 to 1	flip the spinner in Figure 2-30

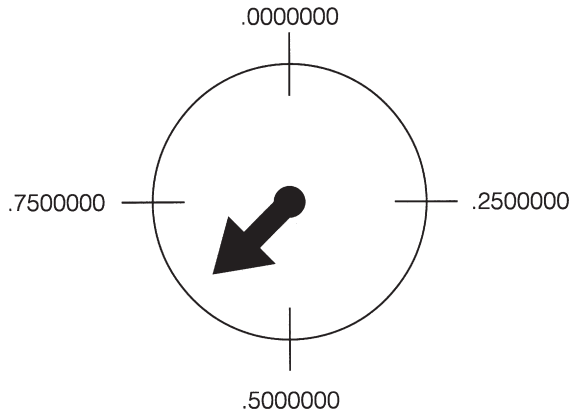


FIGURE 2-30 *Spinner to Randomly Select a Number Between 0 and 1*

The function Rnd, which acts like the spinner in Figure 2-30, returns a random number. The statement

```
picBox.Print Rnd
```

randomly displays a number from 0 up to (but not including) 1. The statement

```
numVar = Rnd
```

randomly assigns a number between 0 and 1 to the variable *numVar*. A different number will be assigned each time Rnd is called in the program, and any number greater than or equal to 0 and less than 1 is just as likely to be generated as any other. Therefore, although Rnd looks like a numeric variable, it does not act at all like a variable.

With appropriate scaling, the Rnd function can generate random numbers from other collections. The statement

```
picBox.Print Int(6 * Rnd) + 1;
```

displays a number from the set 1, 2, 3, 4, 5, 6. Because Rnd always has a value from 0 to 1, excluding 1, 6 * Rnd has a value from 0 to 6 (excluding 6), and Int(6 * Rnd) has one of the values 0, 1, 2, 3, 4, 5. Adding 1 shifts the resulting number into the desired range.

Suppose the preceding statement is repeated many times. The integers generated should exhibit no apparent pattern. They should look very much like a sequence of integers obtained from successively rolling a die. For instance, each of the six integers should appear about one-sixth of the time and be reasonably spread out in the sequence. The longer the sequence, the more likely this is to occur.

Rnd normally generates the same sequence of numbers each time a program is run. However, Visual Basic has another function, Randomize, that changes the sequence of numbers generated by Rnd. This statement will be used in all programs in this text.

EXAMPLE 8

The DC Lottery number is obtained by selecting a Ping-Pong ball from each of three separate bowls. Each ball is numbered with an integer from 0 through 9. Write a computer program to produce a lottery number. (Such a program is said to simulate the selection of Ping-Pong balls.)

SOLUTION:

The value of Int(10 * Rnd) will be an integer from 0 through 9, and each of these integers has the same likelihood of occurring. Repeating the process three times produces the requested digits.

```
Private Sub cmdDisplayANumber_Click()
   'Display a lottery number
   picNumber.Cls
   Randomize
   picNumber.Print Int(10 * Rnd);
   picNumber.Print Int(10 * Rnd);
   picNumber.Print Int(10 * Rnd)
End Sub
```

[Run and then click the command button. One possible output to be displayed in the picture box is as follows.]

```
8 3 9
```

Note: Run the program in Example 8 several times and notice that the output changes each time. Then delete the Randomize statement and run the program several times.

COMMENTS

1. Requesting the square root of a negative number terminates the execution of the program and gives the error message "Invalid procedure call or argument."
2. If *n* is greater than the length of *str*, then the value of Left(*str*, *n*) will be the entire string *str*. A similar result holds for Mid and Right.
3. Visual Basic has a function called LCase that is analogous to UCase. LCase converts all uppercase letters in a string to lowercase letters.
4. Because the values of the functions Left, Mid, Right, UCase, and the Format functions are strings, they are referred to as **string-valued functions**.
5. Mid is an important function. It will be used several times in this book to examine each letter of a string.
6. Trim is useful when reading data from a text box. Sometimes users type spaces at the end of input. Unless the spaces are removed, they can cause havoc elsewhere in the program. Also, Trim is useful in trimming the leading spaces from numbers that have been converted to strings with Str.
7. The InStr function has a useful extension. The value of InStr(*n*, *str1*, *str2*) is the position of the first occurrence of *str2* in *str1* in position *n* or greater. For instance, InStr(5, "Mississippi", "ss") is 6.
8. In Example 4, we found that 4 is the remainder when 256 is divided by 14. Mathematicians say "4 = 256 modulo 14." Visual Basic has an operation, Mod, that performs this calculation directly. If *m* and *n* are positive integers, then *n* Mod *m* is the remainder when *n* is divided by *m*. Visual Basic also has an operation called **integer division**, denoted by \, which gives the quotient portion of a long division problem. For instance, 14/4 is 3.5 whereas 14\4 is the integer 3.
9. Recall that the function Mid has the form Mid(*str*, *m*, *n*) and returns the substring of *str* starting with position *m* and having length *n*. Visual Basic does its best to please for unexpected values of *m* and *n*. If *m* is greater than the length

of the string or *n* is 0, then the empty string is returned. If *m* + *n* is greater than the length of the string, then Mid(*str*, *m*, *n*) is the right part of the string beginning with the mth character. The same is true for Mid(*str*, *m*). For instance, the values of Mid("abcdef", 3, 9) and Mid("abcdef", 3) are both "cdef".

10. With FormatCurrency(*n*, *r*), fractional values are preceded by a leading zero, and negative values are surrounded by parentheses instead of beginning with a minus sign. The function has additional optional parameters. FormatCurrency(*exp*, , vbFalse) suppresses leading zeros for fractional values. FormatCurrency(*exp*, , , vbFalse) uses minus signs for negative numbers. For instance, the value of FormatCurrency(–3/4, 3, vbFalse, vbFalse) is –$.750.
11. With FormatNumber(*n*, *r*), fractional values are preceded by a leading zero. The function has additional optional parameters. FormatNumber(*exp*, , vbFalse) suppresses leading zeros for fractional values. For instance, the value of FormatNumber(3/4, 3) is 0.750 and the value of FormatNumber(3/4, 3, vbFalse) is .750. FormatNumber(*exp*, , , , vbFalse) suppresses commas.
12. When *n* is a number that is halfway between two whole numbers (such as 1.5, 2.5, 3.5, and 4.5), then *n* is rounded by Round(*n*) to the nearest even number. That is, half the time *n* is rounded up and half the time is rounded down. For instance, Round(2.5) is 2 and Round(3.5) is 4. Similar results hold for any number whose decimal part ends in 5. For instance, Round(3.65, 1) is 3.6 and Round(3.75, 1) is 3.8. On the other hand, FormatNumber, FormatCurrency, and FormatPercent always round 5's up. For instance, FormatNumber(2.5) is "3.00".
13. The value of `FormatDateTime(Now, vbLongDate)` is today's date. For any positive number *n*, `FormatDateTime(Now + n, vbLongDate)` is the date *n* days from today and `FormatDateTime(Now - n, vbLongDate)` is the date *n* days ago.
14. The functions FormatNumber, FormatCurrency, FormatPercent, and FormatDateTime were added to Visual Basic in VB6.0. The same results can be obtained with the Format function alone. However, these new functions execute faster than Format and are easier to use. In addition, they can be placed in VBScript programs that are used to make Web pages interactive.
15. Each time the function Rnd appears in a program, it will be reassigned a value. For instance, the task attempted (but not accomplished) by the first set of lines that follows is achieved by the second set of lines. Because each of the Rnd's in the first set of lines will assume a different value, it is highly unlikely that the square of the first one will equal the product of the last two.

```
'Generate the square of a randomly chosen number
Randomize
picBox.Print "The square of"; Rnd; "is"; Rnd * Rnd'
```

```
Generate the square of a randomly chosen number
Randomize
numVar = Rnd
picBox.Print "The square of"; numVar; "is"; numVar * numVar
```

16. Additional information about the keywords, functions, methods, and properties discussed in this section appear in Appendix C. With the Learning, Professional, and Enterprise Editions of Visual Basic you can obtain a detailed (and somewhat advanced) discussion about an item appearing in code by clicking on the item and pressing F1. Other ways of obtaining help with these editions are presented in Appendix B.

SUMMARY

1. The Visual Basic screen consists of a collection of objects for which various properties can be set. Some examples of *objects* are text boxes, labels, command buttons, picture boxes, and the form itself. Objects placed on the form are called *controls*. Some useful properties are Text (set the text displayed by a text box), Caption (set the title of a form, the contents of a label, or the words on a command button), Font.Size (set the size of the characters displayed), Alignment (set the placement of the contents of a label), MultiLine (text box to display text on several lines), Picture (display drawing in picture box), ForeColor (set foreground color), BackColor (set background color), Visible (show or hide object), BorderStyle (alter and possibly remove border), Font.Bold (display boldface text), and Font.Italic (display italic text).
2. An event procedure is called when a specific event occurs to a specified object. Some event procedures are *object*_Click (*object* is clicked), *object*_LostFocus (*object* loses the focus), *object*_GotFocus (*object* receives the focus), and *object*_KeyPress (a key is pressed while *object* has the focus).
3. Visual Basic methods such as Print and Cls are applied to objects and are coded as *object*.Print and *object*.Cls.
4. Two types of *constants* that can be stored and processed by computers are *numbers* and *strings*.
5. The arithmetic *operations* are +, –, *, /, and ^. The only string operation is &, concatenation. An *expression* is a combination of constants, variables, functions, and operations that can be evaluated.
6. A *variable* is a name used to refer to data. Variable names can be up to 255 characters long, must begin with a letter, and may contain letters, digits, and underscores. Dim statements explicitly declare variables and specify the types of the variables. In this book, variables have types Single, Integer, and String.
7. Values are assigned to variables by assignment and Input # statements. The values appearing in assignment statements can be constants, variables, or expressions. Input # statements look to data files for constants. String constants used in assignment statements must be surrounded by quotation marks, whereas quotation marks are optional for string constants input with Input #. InputBox can be used to request that the user type in data.
8. The Print method displays information in a picture box or on the printer. *Semicolons*, *commas*, and *Tab* control the placement of the items on a particular line. A temporary message can be displayed on the screen using the MsgBox statement.
9. You control the printer with the Printer object and write to it with statements of the form Printer.Print *expression*. You set properties with statements of the form Printer.*property* = *setting*. Printer.NewPage starts a new page and PrintForm does a screen dump. A series of commands to the Printer object must end with EndDoc, which actually produces the final printed page.
10. Comment statements are used to explain formulas, state the purposes of variables, and articulate the purposes of various parts of a program.
11. The Format functions provide detailed control of how numbers, dates, and strings are displayed. Numbers can be made to line up uniformly and be displayed with dollar signs, commas, and a specified number of decimal places. Dates can be converted to a long form. Strings can be right-justified.

12. *Functions* can be thought of as accepting numbers or strings as input and returning numbers or strings as output.

Function	Input	Output
Asc	string	number
Chr	number	string
InStr	string, string	number
Int	number	number
LCase	string	string
Left	string, number	string
Len	string	number
Mid	string, number, number	string
Right	string,, number	string
Rnd		number
Round	number, number	number
Sqr	number	number
Str	number	string
Trim	string	string
UCase	string	string
Val	string	number

PROGRAMMING PROJECTS

1. Write a program that allows the user to specify two numbers and then adds, subtracts, or multiplies them when the user clicks on the appropriate command button. The output should give the type of arithmetic performed and the result.
2. Suppose automobile repair customers are billed at the rate of $35 per hour for labor. Also, costs for parts and supplies are subject to a 5 percent sales tax. Write a program to print out a simplified bill. The customer's name, the number of hours of labor, and the cost of parts and supplies should be entered into the program via text boxes. When a command button is clicked, the customer's name (indented) and the three costs should be displayed in a picture box, as shown in the sample run in Figure 2-31.

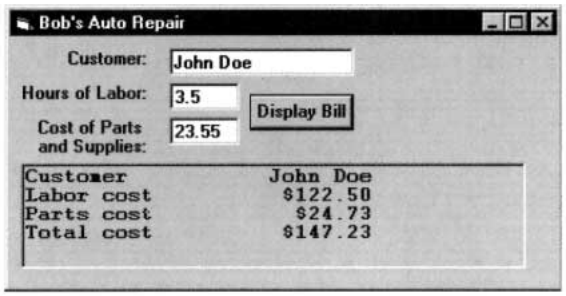


FIGURE 2-31 *Sample Run for Programming Project 2*

3. Write a program to generate the following personalized form letter. The person's name and address should be read from text boxes.

```
Mr. John Jones
123 Main Street
Juneau, Alaska 99803

Dear Mr. Jones,
```

```
    The Jones family has been selected as the
first family on Main Street to have the opportunity
to purchase an Apex solar-powered flashlight. Due to limited
supply, only 1000 of these amazing inventions will be available
in the entire state of Alaska. Don't delay. Order today.
                        Sincerely,
                        Cuthbert J. Twillie
```

4. At the end of each month, a credit card company constructs the table in Figure 2-32 to summarize the status of the accounts. Write a program to produce this table. The first four pieces of information for each account should be read from a data file. The program should compute the finance charges (1.5 percent of the unpaid past due amount) and the current amount due. Format the last column to be aligned right.

Account Number	Past Due Amount	Payments	Purchases	Finance Charges	Current Amt Due
123-AB	123.45	10.00	934.00	1.70	$1,049.15
456-CD	134.56	134.56	300.00	0.00	$300.00

FIGURE 2-32 *Status of Credit Card Accounts*

5. Table 2-11 gives the distribution of the U.S. population (in thousands) by age group and sex. Write a program to produce the table shown in Figure 2-33. For each age group, the column labeled %Males gives the percentage of the people in that age group that are male, similarly for the column labeled %Females. The last column gives the percentage of the total population in each age group. ***Note:*** Store the information in Table 2-11 in a data file. For instance, the first line in the file should be "Under 20", 39168, 37202. Read and add up the data once to obtain the total population, and then read the data again to produce the table.)

TABLE 2-11
U.S. Resident Population in Thousands (1996)

Age Group	Males	Females
Under 20	39,168	37,202
20–64	76,761	78,291
Over 64	13,881	19,980

```
                    U.S. Population (in thousands)

Age group     Males      Females     %Males     %Females     %Total

Under 20      39,168     37,202      51.29%     48.71%       28.79%
20-64         76,761     78,291      49.51%     50.49%       58.45%
Over 64       13,881     19,980      40.99%     59.01%       12.76%
```

FIGURE 2-33 *Output of Programming Project 5*

6. Write a program to convert a U.S. Customary System length in miles, yards, feet, and inches to a Metric System length in kilometers, meters, and centimeters. A sample run is shown in Figure 2-34. After the number of miles, yards, feet, and inches are read from the text boxes, the length should be converted entirely to inches and then divided by 39.37 to obtain the value in meters. The Int function should be used to break the total number of meters into a whole

number of kilometers and meters. The number of centimeters should be displayed to one decimal place. Some of the needed formulas are as follows:

total inches = 63360 * miles + 36 * yards + 12 * feet + inches

total meters = total inches / 39.37

kilometers = Int(meters / 1000)

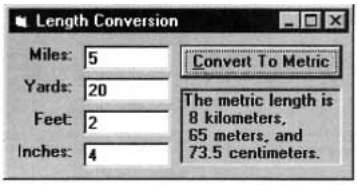


FIGURE 2-34 *Sample Run for Programming Project 6*

SECTION 3

GENERAL PROCEDURES

3.1 SUB PROCEDURES, PART I

Structured program design requires that problems be broken into small problems to be solved one at a time. Visual Basic has two devices, **Sub procedures** and **Function procedures**, that are used to break problems into manageable chunks. To distinguish them from event procedures, Sub and Function procedures are referred to as **general procedures**. General procedures also eliminate repetitive code, can be reused in other programs, and allow a team of programmers to work on a single program.

In this section we show how Sub procedures are defined and used. The programs in this section are designed to demonstrate the use of Sub procedures rather than to accomplish sophisticated programming tasks. Later chapters of the book use them for more substantial programming efforts.

A **Sub procedure** is a part of a program that performs one or more related tasks, has its own name, is written as a separate part of the program. The simplest type of Sub procedure has the form

```
Private Sub ProcedureName()
   statement(s)
End Sub
```

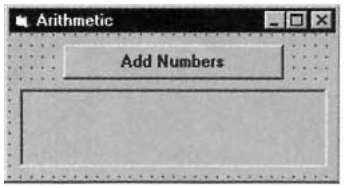
A Sub procedure is invoked with a statement of the form

```
Call ProcedureName
```

The rules for naming general procedures are identical to the rules for naming variables. The name chosen for a Sub procedure should describe the task it performs. Sub procedures can be either typed directly into the code window, or into a template created with the following steps:

1. Press Alt/T/P to select Add Procedure from the Tools menu.
2. Type in the name of the procedure. (Omit parentheses.)
3. Select Sub from the Type box.
4. Select Private from the Scope box. ***Note:*** Actually either Public or Private is OK. A Public procedure is available to all forms, whereas a Private procedure is only available to the form in which it is defined.
5. Press the Enter key or click on OK.

Consider the following program that calculates the sum of two numbers. This program will be revised to incorporate Sub procedures.



Object	Property	Setting
frmArithmetic	Caption	Arithmetic
cmdAdd	Caption	Add Numbers
picResult		

```
Private Sub cmdAdd_Click()
   Dim num1 As Single, num2 As Single
   'Display the sum of two numbers
   picResult.Cls
   picResult.Print "This program displays a sentence "
   picResult.Print "identifying two numbers and their sum."
   picResult.Print num1 = 2 num2 = 3
   picResult.Print "The sum of"; num1; "and"; num2; "is"; num1 + num2
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
This program displays a sentence
identifying two numbers and their sum.

The sum of 2 and 3 is 5
```

The tasks performed by this program can be summarized as follows:

Explain purpose of program.

Display numbers and their sum.

Sub procedures allow us to write and read the program in such a way that we first focus on the tasks and later on how to accomplish each task.

EXAMPLE 1

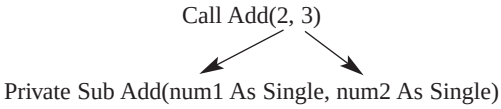
The following program uses a Sub procedure to accomplish the first task of the preceding program. When the statement Call ExplainPurpose is reached, execution jumps to the Sub ExplainPurpose statement. The lines between Sub ExplainPurpose and End Sub are executed, and then execution continues with the line following the Call statement.

```
Private Sub cmdAdd_Click()
   Dim num1 As Single, num2 As Single
   'Display the sum of two numbers
   picResult.Cls
   Call ExplainPurpose
   picResult.Print
   num1 = 2
   num2 = 3
   picResult.Print "The sum of"; num1; "and"; num2; "is"; num1 + num2
End Sub

Private Sub ExplainPurpose()
   'Explain the task performed by the program
   picResult.Print "This program displays a sentence"
   picResult.Print "identifying two numbers and their sum."
End Sub
```

The second task performed by the addition program also can be handled by a Sub procedure. The values of the two numbers, however, must be transmitted to the Sub procedure. This transmission is called **passing**.

EXAMPLE 2

The following revision of the program in Example 1 uses a Sub procedure to accomplish the second task. The statement Call Add(2, 3) causes execution to jump to the Private Sub Add(num1 As Single, num2 As Single) statement, which assigns the number 2 to *num1* and the number 3 to *num2*.



After the lines between Private Sub Add (num1 As Single, num2 As Single) and End Sub are executed, execution continues with the line following Call Add(2, 3), namely, the End Sub statement in the event procedure. ***Note 1:*** When you create the Sub procedure Add, you must type in "(num1 As Single, num2 As Single)" after leaving the Add Procedure dialog box.

```
Private Sub cmdAdd_Click()
   'Display the sum of two numbers
   picResult.Cls
   Call ExplainPurpose
   picResult.Print
   Call Add(2, 3)
End Sub

Private Sub Add(num1 As Single, num2 As Single)
   'Display numbers and their sum
   picResult.Print "The sum of"; num1; "and"; num2; "is"; num1 + num2
End Sub

Private Sub ExplainPurpose()
   'Explain the task performed by the program
   picResult.Print "This program displays a sentence"
   picResult.Print "identifying two numbers and their sum."
End Sub
```

Sub procedures make a program easy to read, modify, and debug. The event procedure gives a description of what the program does and the Sub procedures fill in the details. Another benefit of Sub procedures is that they can be called several times during the execution of the program. This feature is especially useful when there are many statements in the Sub procedure.

EXAMPLE 3

The following extension of the program in Example 2 displays several sums.

```
Private Sub cmdAdd_Click()
   'Display the sums of several pairs of numbers
   picResult.Cls
   Call ExplainPurpose
   picResult.Print
   Call Add(2, 3)
   Call Add(4, 6)
   Call Add(7, 8)
End Sub
```

```
Private Sub Add(num1 As Single, num2 As Single)
   'Display numbers and their sum picResult.Print "The sum of"; num1; "and";
   num2; "is"; num1 + num2
End Sub

Private Sub ExplainPurpose()
   'Explain the task performed by the program
   picResult.Print "This program displays sentences"
   picResult.Print "identifying pairs of numbers and their sums."
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
This program displays sentences
identifying pairs of numbers and their sums.

The sum of 2 and 3 is 5
The sum of 4 and 6 is 10
The sum of 7 and 8 is 15
```

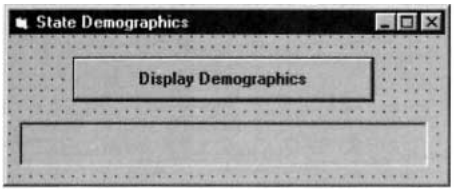
The variables *num1* and *num2* appearing in the Sub procedure Add are called **parameters**. They are merely temporary place holders for the numbers passed to the Sub procedure; their names are not important. The only essentials are their type, quantity, and order. In this Add Sub procedure, the parameters must be numeric variables and there must be two of them. For instance, the Sub procedure could have been written

```
Private Sub Add(this As Single, that As Single)
   'Display numbers and their sum
   picResult.Print "The sum of"; this; "and"; that; "is"; this + that
End Sub
```

A string also can be passed to a Sub procedure. In this case, the receiving parameter in the Sub procedure must be followed by the declaration As String.

EXAMPLE 4

The following program passes a string and two numbers to a Sub procedure. When the Sub procedure is first called, the string parameter *state* is assigned the string constant "Hawaii", and the numeric parameters *pop* and *area* are assigned the numeric constants 1184000 and 6471, respectively. The Sub procedure then uses these parameters to carry out the task of calculating the population density of Hawaii. The second Call statement assigns different values to the parameters.



Object	Property	Setting
frmStates	Caption	State Demographics
cmdDisplay	Caption	Display Demographics
picDensity		

```
Private Sub cmdDisplay_Click()
   'Calculate the population densities of states
   picDensity.Cls
   Call CalculateDensity("Hawaii", 1184000, 6471)
   Call CalculateDensity("Alaska", 607000, 591000)
End Sub

Private Sub CalculateDensity(state As String, pop As Single, area As Single)
   Dim rawDensity As Single, density As Single
```

```
   'The density (number of people per square mile)
   'will be displayed rounded to a whole number
   rawDensity = pop / area
   density = Round(rawDensity) 'round to whole number
   picDensity.Print "The density of "; state; " is"; density;
   picDensity.Print "people per square mile."
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
The density of Hawaii is 183 people per square mile.
The density of Alaska is 1 people per square mile.
```

The parameters in the density program can have any names, as with the parameters in the addition program of Example 3. The only restriction is that the first parameter be a string variable and that the last two parameters be numeric variables. For instance, the Sub procedure could have been written

```
Private Sub CalculateDensity(x As String, y As Single, z As Single)
   Dim rawDensity As Single, density As Single
   'The density (number of people per square mile)
   'will be rounded to a whole number
   rawDensity = y / z
   density = Round(rawDensity)
   picDensity.Print "The density of "; x; " is"; density;
   picDensity.Print "people per square mile."
End Sub
```

When nondescriptive names are used for parameters, the Sub procedure should contain comments giving the meanings of the variables. Possible comments for the preceding program are

```
'x name of the state'
y population of the state'
z area of the state
```

■ VARIABLES AND EXPRESSIONS AS ARGUMENTS

The items appearing in the parentheses of a Call statement are called **arguments**. These should not be confused with parameters, which appear in the heading of a Sub procedure. In Example 3, the arguments of the Call Add statements were constants. These arguments also could have been variables or expressions. For instance, the event procedure could have been written as follows. See Figure 3-1.

```
Private Sub cmdAdd_Click()
   Dim x As Single, y As Single, z As Single
   'Display the sum of two numbers
   picResult.Cls
   Call ExplainPurpose
   picResult.Print
   x = 2
   y = 3
   Call Add(x, y)
   Call Add(x + 2, 2 * y) z = 7
   Call Add(z, z + 1)
End Sub
```

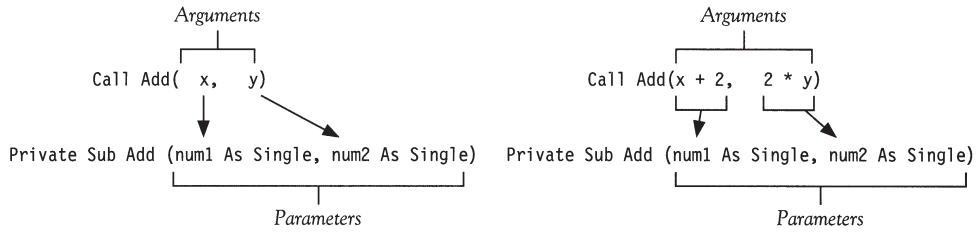
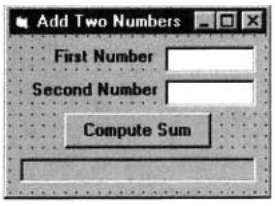


FIGURE 3-1 *Passing Arguments to Parameters*

This feature allows values obtained as input from the user to be passed to a Sub procedure.

EXAMPLE 5

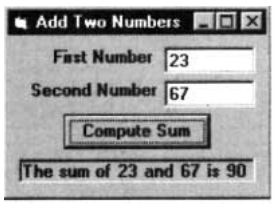
The following variation of the addition program requests the two numbers as input from the user. Notice that the names of the arguments, *x* and *y*, are different from the names of the parameters. The names of the arguments and parameters may be the same or different; what matters is that the order, number, and types of the arguments and parameters match.



Object	Property	Setting
frmAdd	Caption	Add Two Numbers
lblFirstNum	Caption	First Number
txtFirstNum	Text	(blank)
lblSecondNum	Caption	Second Number
txtSecondNum	Text	(blank)
cmdCompute	Caption	Compute Sum
picResult		

```
Private Sub cmdCompute_Click()
   Dim x As Single,, y As Single
   'This program requests two numbers and
   'displays the two numbers and their sum.
   x = Val(txtFirstNum.Text)
   y = Val(txtSecondNum.Text)
   Call Add(x,, y)
End Sub

Private Sub Add(num1 As Single,, num2 As Single)
   'Display numbers and their sum
   picResult.Cls
   picResult.Print "The sum of"; num1; "and"; num2; "is"; num1 + num2
End Sub
```

[Run, type 23 and 67 into the text boxes, and then click the command button.]



EXAMPLE 6

The following variation of Example 4 obtains its input from the file DEMOGRAP.TXT. The second Call statement uses different variable names for the arguments to show that using the same argument names is not necessary. See Figure 3-2.

DEMOGRAP.TXT contains the following two lines:

"Hawaii", 1184000, 6471
"Alaska", 607000, 591000

```
Private Sub cmdDisplay_Click()
   Dim state As String, pop As Single, area As Single
   Dim s As String, p As Single, a As Single
   'Calculate the population densities of states
   picDensity.Cls Open "DEMOGRAP.TXT" For Input As #1
   Input #1, state, pop, area
   Call CalculateDensity(state, pop, area)
   Input #1, s, p, a
   Call CalculateDensity(s, p, a)
   Close #1
End Sub

Private Sub CalculateDensity(state As String, pop As Single, area As Single)
   Dim rawDensity As Single, density As Single
   'The density (number of people per square mile)
   'will be rounded to a whole number
   rawDensity = pop / area density = Round(rawDensity)
   picDensity.Print "The density of "; state; " is "; density;
   picDensity.Print "people per square mile."
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
The density of Hawaii is 183 people per square mile.
The density of Alaska is 1 people per square mile.
```

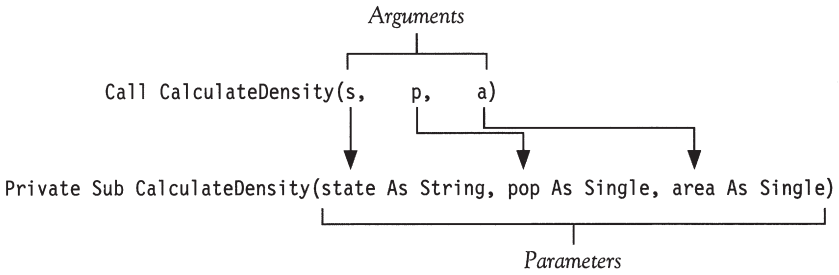


FIGURE 3-2 *Passing Arguments to Parameters in Example 6*

Arguments and parameters also can be used to pass values from Sub procedures back to event procedures or other Sub procedures. This important property is explored in detail in the next section.

COMMENTS

1. In this text, Sub procedure names begin with uppercase letters in order to distinguish them from variable names. Like variable names, however, they can be written with any combination of upper- and lowercase letters. To improve readability, the Visual Basic editor will automatically ensure that the capitalization of a Sub procedure name is consistent throughout a program. For instance, if you type Private Sub PROCEDURENAME and also type Call ProcedureName, the second name will be changed to match the first. ***Note:*** Parameters appearing in a Sub statement are not part of the Sub procedure name.

2. To obtain a list of the general procedures in a program, select (General) from the Code window's Object box and then click on the down-arrow at the right side of the Procedure box.
3. Sub procedures allow programmers to focus on the main flow of the program and defer the details of implementation. Modern programs use them liberally. An event procedure acts as a supervisor, delegating tasks to the Sub procedures. This method of program construction is known as **modular** or **top-down** design.
4. As a rule, a Sub procedure should perform only one task, or several closely related tasks, and should be kept relatively small.
5. After a Sub procedure has been defined, Visual Basic automatically reminds you of the Sub procedure's parameters when you type in Call statements. As soon as you type in the left parenthesis of a Call statement, a banner appears giving the names and types of the parameters. The help feature is called **Parameter Info**. See Figure 3-3.

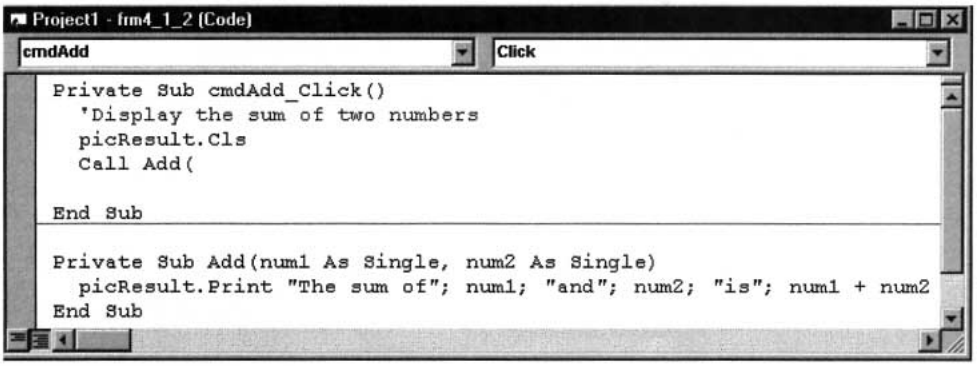


FIGURE 3-3 *The Parameter Info Help Feature*

6. In this text, the first line inside a Sub procedure is often a comment statement describing the task performed by the Sub procedure. If necessary, several comment statements are devoted to this purpose. Conventional programming practice also recommends that all variables used by the Sub procedure be listed in comment statements with their meanings. In this text, we give several examples of this practice but only adhere to it when the variables are especially numerous or lack descriptive names.
7. Although both constants and expressions can be used as arguments in Call statements, only variables can be used as parameters in Sub statements.
8. A Sub procedure can call another Sub procedure. If so, after the End Sub of the called Sub procedure is reached, execution continues with the line in the calling Sub procedure that follows the Call statement.
9. When you write a Sub procedure without parameters, Visual Basic automatically adds a pair of empty parentheses at the end of the Sub procedure name. However, Call statements should not use the empty parentheses.
10. The first lines of event procedures and Sub procedures end with a pair of parentheses. With the event procedures we have discussed, the parentheses are usually empty, whereas with Sub procedures, the parentheses often contain parameters.

3.2 SUB PROCEDURES, PART II

The previous section introduced the concept of a Sub procedure but left some questions unanswered. Why can't the value of a variable be passed from an event procedure to a Sub proce-

dure by just using the variable in the Sub procedure? How do Sub procedures pass values back to an event procedure? The answers to these questions provide a deeper understanding of the workings of Sub procedures and reveal their full capabilities.

Passing Values Back from Sub Procedures

Suppose a variable, call it *arg*, appears as an argument in a Call statement, and its corresponding parameter in the Sub statement is *par*. After the Sub procedure is executed, *arg* will have whatever value *par* had in the Sub procedure. Hence, not only is the value of *arg* passed to *par*, but the value of *par* is passed back to *arg*.

EXAMPLE 1

The following program illustrates the transfer of the value of a parameter to its calling argument.

```
Private Sub cmdDisplay_Click()
   Dim amt As Single
   'Illustrate effect of value of parameter on value of argument
   picResults.Cls
   amt = 2
   picResults.Print amt;
   Call Triple(amt)
   picResults.Print amt
End Sub

Private Sub
   Triple(num As Single)
   'Triple a number picResults.Print num;
   num = 3 * num
   picResults.Print num;
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
2    2    6    6
```

Although this feature may be surprising at first glance, it provides a vehicle for passing values from a Sub procedure back to the place from which the Sub procedure was called. Different names may be used for an argument and its corresponding parameter, but only one memory location is involved. Initially, the cmdDisplay_Click( ) event procedure allocates a memory location to hold the value of *amt* (Figure 3-4(a)). When the Sub procedure is called, the parameter *num* becomes the Sub procedure's name for this memory location (Figure 3-4(b)). When the value of *num* is tripled, the value in the memory location becomes 6 (Figure 3-4(c)). After the completion of the procedure, the parameter name *num* is forgotten; however, its value lives on in *amt* (Figure 3-4(d)). The variable *amt* is said to be **passed by reference**.

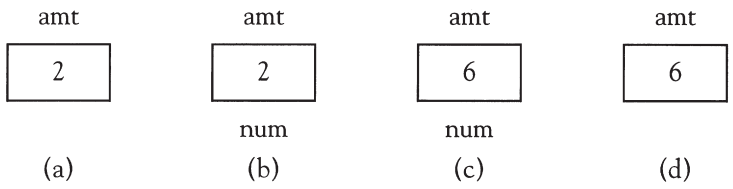
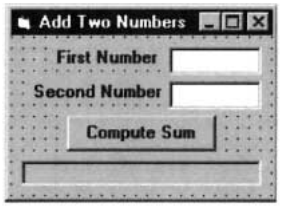


FIGURE 3-4 *Passing a Variable by Reference to a Sub Procedure*

Passing by reference has a wide variety of uses. In the next example, it is used as a vehicle to transport a value from a Sub procedure back to an event procedure.

EXAMPLE 2

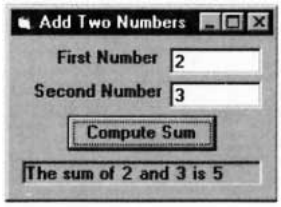
The following variation of the addition program from the previous section uses a Sub procedure to acquire the input. The variables *x* and *y* are not assigned values prior to the execution of the first Call statement. Therefore, before the Call statement is executed, they have the value 0. After the Call statement is executed, however, they have the values 2 and 3. These values then are passed by the second Call statement to the Sub procedure Add.



Object	Property	Setting
frmAdd	Caption	Add Two Numbers
lblFirstNum	Caption	First Number
txtFirstNum	Text	(blank)
lblSecondNum	Caption	Second Number
txtSecondNum	Text	(blank)
cmdCompute	Caption	Compute Sum
picResult		

```
Private Sub cmdCompute_Click()
   Dim x As Single, y As Single
   'Display the sum of the two numbers
   Call GetNumbers(x, y)
   Call Add(x, y)
End Sub

Private Sub Add(num1 As Single, num2 As Single)
   Dim sum As Single
   'Display numbers and their sum
   picResult.Cls
   sum = num1 + num2
   picResult.Print "The sum of"; num1; "and"; num2; "is"; sum
End Sub

Private Sub GetNumbers(num1 As Single, num2 As Single)
   'Record the two numbers in the text boxes
   num1 = Val(txtFirstNum.Text)
   num2 = Val(txtSecondNum.Text)
End Sub
```

[Run, type 2 and 3 into the text boxes, and then click the command button.]



In most situations, a variable with no preassigned value is used as an argument of a Call statement for the sole purpose of carrying back a value from the Sub procedure.

EXAMPLE 3

The following variation of Example 2 allows the cmdCompute_Click event procedure to be written in the input-process-output style.

```
Private Sub cmdCompute_Click()
   Dim x As Single, y As Single, s As Single
   'Display the sum of two numbers
   Call GetNumbers(x, y)
```

```
   Call CalculateSum(x, y, s)
   Call DisplayResult(x, y, s)
End Sub

Private Sub CalculateSum(num1 As Single, num2 As Single, sum As Single)
   'Add the values of num1 and num2
   'and assign the value to sum
   sum = num1 + num2
End Sub

Private Sub DisplayResult(num1 As Single, num2 As Single, sum As Single)
   'Display a sentence giving the two numbers and their sum
   picResult.Cls
   picResult.Print "The sum of"; num1; "and"; num2; "is"; sum
End Sub

Private Sub GetNumbers(num1 As Single, num2 As Single)
   'Record the two numbers in the text boxes
   num1 = Val(txtFirstNum.Text)
   num2 = Val(txtSecondNum.Text)
End Sub
```

Passing by Value

Sometimes you want to pass a variable to a Sub procedure, but you want to ensure that the variable will retain its original value after the Sub procedure terminates—regardless of what was done to the corresponding parameter inside the Sub procedure. Such a variable is said to be **passed by value**. There are two ways to pass a variable by value.

1. In the Call statement, enclose the variable in an extra pair of parentheses.
2. In the Private Sub statement, precede the corresponding parameter with the word ByVal.

For instance, in Example 1, if you change the Call statement to

```
Call Triple((amt))
```

then the output will be

```
2   2   6   2
```

The same output results if you change the Private Sub statement to

```
Private Sub Triple(ByVal num As Single)
```

When a variable is passed by value, two memory locations are involved. At the time the Sub procedure is called, a temporary second memory location for the parameter is set aside for the Sub procedure's use and the value of the argument is copied into that location. After the completion of the Sub procedure, the temporary memory location is released and the value in it is lost.

Local Variables

When the same variable name appears in two different Sub procedures or in a Sub procedure and an event procedure, Visual Basic gives the variables separate identities and treats them as two different variables. A value assigned to a variable in one part of the program will not affect the value of the like-named variable in the other part of the program, unless, of course, the values are passed by a Call statement. Also, each time a Sub procedure is called, all declared variables that are not parameters assume their default values. (Numeric variables have

default value 0, and string variables default to the empty string.) The variables in a Sub procedure are said to be local to the Sub procedure in which they reside.

EXAMPLE 4

The following program illustrates the fact that each time a Sub procedure is called, its variables are set to their default values; that is, numerical variables are set to 0 and string variables are set to the empty string.

```
Private Sub cmdDisplay_Click()
   'Demonstrate that variables in a Sub procedure do
   'not retain their values in subsequent calls
   picResults.Cls
   Call Three
   Call Three
End Sub

Private Sub Three()
   Dim num As Single
   'Display the value of num and assign it the value 3
   picResults.Print num;
   num = 3
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
0 0
```

EXAMPLE 5

The following program illustrates the fact that variables are local to the part of the program in which they reside. The variable *x* in the event procedure and the variable *x* in the Sub procedure are treated as different variables. Visual Basic handles them as if their names were separate, such as *xcmdDisplay_Click* and *xTrivial*. Also, each time the Sub procedure is called, the value of variable *x* inside the Sub procedure is reset to 0.

```
Private Sub cmdDisplay_Click()
   Dim x As Single
   'Demonstrate the local nature of variables
   picResults.Cls
   x = 2
   picResults.Print x;
   Call Trivial
   picResults.Print x;
   Call Trivial
   picResults.Print x;
End Sub

Private Sub Trivial()
   Dim x As Single
   'Do something trivial
   picResults.Print x;
   x = 3
   picResults.Print x;
End Sub
```

[Run and then click the command button. The following is displayed in the picture box.]

```
2 0 3 2 0 3 2
```

FORM-LEVEL VARIABLES

Visual Basic provides a way to make a variable visible to *every* procedure in a form's code without being passed. Such a variable is called a **form-level variable**. Form-level variables appear at the top of the code window and are separated from the rest of the code by a horizontal separator line. Inside the code window, you can move to them either by pressing Ctrl+Home or clicking on General in the Object list box. Form-level variables are said to reside in the (Declarations) section of (General) and are declared with the following steps.

1. Invoke a code window if one is not already active.
2. Click on the down-arrow to the right of the Object list box.
3. Click on (General).
4. Click on (Declarations) in the Procedure list box.
5. Type in a declaration statement, such as Dim strVar As String, and press the Enter key.

When a form-level variable is assigned a value by a procedure, it retains that value when the procedure is exited. In this text, we rarely use form-level variables until Section 6.

EXAMPLE 6

The following program contains the form-level variables *num1* and *num2*. Their Dim statement does not appear inside a procedure.

```
Dim num1 As Single, num2 As Single 'In (Declarations) section of (General)

Private Sub cmdDisplay_Click()
   'Display the sum of two numbers
   num1 = 2
   num2 = 3
   picResults.Cls
   Call AddAndIncrement
   picResults.Print
   picResults.Print "num1 = "; num1
   picResults.Print "num2 = "; num2
End Sub

Private Sub AddAndIncrement()
   'Display numbers and their sum
   picResults.Print "The sum of"; num1; "and"; num2; "is"; num1 + num2
   num1 = num1 + 1
   num2 = num2 + 1
End Sub
```

[Run, and click the command button. The following is displayed in the picture box.]

```
The sum of 2 and 3 is 5

num1 = 3
num2 = 4
```

In the preceding example, we had to click a command button to assign values to the form-level variables. In some situations, we want to assign a value immediately to a form-level variable, without requiring the user to perform some specific action. Visual Basic has a special event procedure called Form_Load that is automatically activated as soon as the program is run, even before the form is created. The Form_Load template is invoked by double-clicking on the form itself.

EXAMPLE 7

The following program demonstrates the use of Form_Load.

```
Dim pi As Single 'In (Declarations) section of (General)

Private Sub Form_Load()
   'Assign a value to pi
   pi = 3.14159
End Sub

Private Sub cmdCompute_Click()
   'Display the area of a circle of radius 5
   picArea.Cls
   picArea.Print "The area of a circle of radius 5 is"; pi * 5 * 5
End Sub
```

[Run, and then click the command button. The following is displayed in the picture box.]

```
The area of a circle of radius 5 is 78.53975
```

COMMENTS

1. In addition to the reasons presented earlier, some other reasons for using Sub procedures follow:
   (a) Programs with Sub procedures are easier to debug. Each Sub procedure can be checked individually before being placed into the program.
   (b) The task performed by a Sub procedure might be needed in another program. The Sub procedure can be reused with no changes. Programmers refer to the collection of their most universal Sub procedures as a **library** of Sub procedures. (The fact that variables appearing in Sub procedures are local to the Sub procedures is quite helpful when reusing Sub procedures in other programs. There is no need to worry if a variable name in the Sub procedure is used for a different purpose in another part of the program.)
   (c) Often, programs are written by a team of programmers. After a problem has been broken into distinct and manageable tasks, each programmer is assigned a single Sub procedure to write.
   (d) Sub procedures make large programs easier to understand. Some programming standards insist that each Sub procedure be at most two pages long.
   (e) Sub procedures permit the following program design, which provides a built-in outline of an event procedure. A reader can focus on the main flow first, and then go into the specifics of accomplishing the secondary tasks.

```
Private Sub Object_Event()
   'An event procedure written entirely as Sub procudures
   Call FirstSubprocedure 'Perform first task
   Call SecondSubprocedure 'Perform second task
   Call ThirdSubprocedure 'Perform third task
End Sub
```

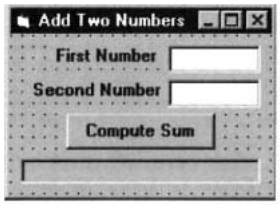
2. Sub procedures can call other Sub procedures. In such cases, the calling Sub procedure plays the role of the event procedure with respect to the called Sub procedure. Complex problems are thereby broken into simpler tasks, which are then broken into still more elementary tasks. This approach to problem solving is called **top-down design**.

3. In Appendix D, the section "Stepping Through a Program Containing a Procedure: Section 3" uses the Visual Basic debugger to trace the flow through a program and observe the interplay between arguments and parameters.
4. You can use the Print method in a Form_Load event procedure. If so, you must set the AutoRedraw property of the picture box to True. Otherwise the contents of the picture box will be erased when the event procedure terminates.

3.3 FUNCTION PROCEDURES

Visual Basic has many built-in functions. In one respect, functions are like miniature programs. They use input, they process the input, and they have output. Some functions we encountered earlier are listed in Table 3.1.

TABLE 3.1
Some Visual Basic Built-In Functions



Function	Example	Input	Output
Int	Int(2.6) is 2	number	number
Chr	Chr(65) is "A"	number	string
Len	Len("perhaps") is 7	string	number
Mid	Mid("perhaps",4,2) is "ha"	string, number, number	string
InStr	InStr("to be"," ") is 3	string,string	number

Although the input can involve several values, the output always consists of a single value. The items inside the parentheses can be constants (as in Table 3.1), variables, or expressions.

In addition to using built-in functions, we can define functions of our own. These new functions, called **Function procedures** or **user-defined functions**, are defined in much the same way as Sub procedures and are used in the same way as built-in functions. Like built-in functions, Function procedures have a single output that can be string or numeric. Function procedures can be used in expressions in exactly the same way as built-in functions. Programs refer to them as if they were constants, variables, or expressions. Function procedures are defined by function blocks of the form

```
Private Function FunctionName(var1 As Type1, var2 As Type2, ...) As dataType
   statement(s)
   FunctionName = expression
End Function
```

The variables in the top line are called **parameters**, and variables inside the function block that are not parameters have local scope. Function names should be suggestive of the role performed and must conform to the rules for naming variables. The type *dataType*, which specifies the type of the output, will be one of String, Integer, Single, and so on. In the preceding general code, the next-to-last line assigns the output, which must be of type *dataType*, to the function name. Two examples of Function procedures are as follows:

```
Private Function FtoC(t As Single) As Single
   'Convert Fahrenheit temperature to Celsius
   FtoC = (5 / 9) * (t - 32)
End Function

Private Function FirstName(nom As String) As String
   Dim firstSpace As Integer
   'Extract the first name from the full name nom
   firstSpace = InStr(nom, " ")
```

```
   FirstName = Left(nom, firstSpace - 1)
End Function
```

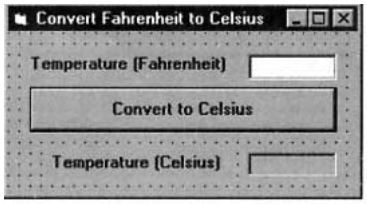
The value of each of the preceding functions is assigned by a statement of the form *FunctionName = expression*. The variables *t* and *nom* appearing in the preceding functions are parameters. They can be replaced with any variable of the same type without affecting the function definition. For instance, the function FtoC could have been defined as

```
Private Function FtoC(temp As Single) As Single
   'Convert Fahrenheit temperature to Celsius
   FtoC = (5 / 9) * (temp - 32)
End Function
```

Like Sub procedures, Function procedures can be created from a code window with Alt/T/P. The only difference is that the circle next to the word Function should be selected. After the name is typed and the OK button is clicked, the lines Private Function *FunctionName*( ) and End Function will be placed automatically (separated by a blank line) in the code window.

EXAMPLE 1

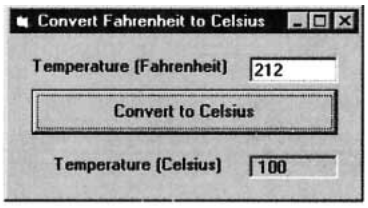
The following program uses the function FtoC.



Object	Property	Setting
frm4_3_1	Caption	Convert Fahrenheit to Celsius
lblTempF	Caption	Temperature (Fahrenheit)
txtTempF	Text	(blank)
cmdConvert	Caption	Convert to Celsius
lblTempC	Caption	Temperature (Celsius)
picTempC		

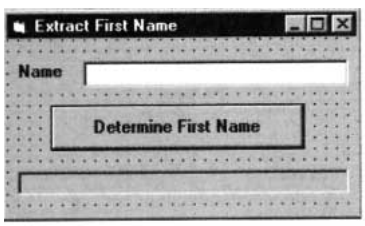
```
Private Sub cmdConvert_Click()
   picTempC.Cls
   picTempC.Print FtoC(Val(txtTempF.Text))
End Sub

Private Function FtoC(t As Single) As Single
   'Convert Fahrenheit temperature to Celsius
   FtoC = (5 / 9) * (t - 32)
End Function
```

[Run, type 212 into the text box, and then click the command button.]



EXAMPLE 2

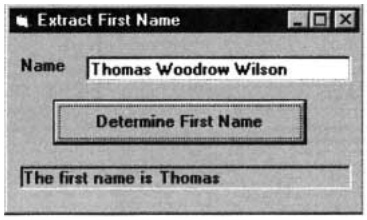
The following program uses the function FirstName.



Object	Property	Setting
frm4_3_2	Caption	Extract First Name
lblName	Caption	Name
txtFullName	Text	(blank)
cmdDetermine	Caption	Determine First Name
picFirstName		

```
Private Sub cmdDetermine_Click()
   Dim nom As String
   'Determine a person's first name
   nom = txtFullName.Text
   picFirstName.Cls
   picFirstName.Print "The first name is "; FirstName(nom)
End Sub

Private Function FirstName(nom As String) As String
   Dim firstSpace As Integer
   'Extract the first name from a full name
   firstSpace = InStr(nom, " ")
   FirstName = Left(nom, firstSpace - 1)
End Function
```

[Run, type Thomas Woodrow Wilson into the text box, and then click the command button.]

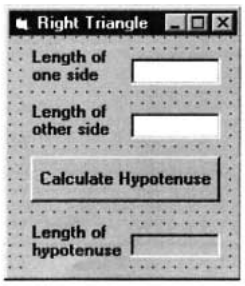


The input to a user-defined function can consist of one or more values. Two examples of functions with several parameters follow. One-letter variable names have been used so the mathematical formulas will look familiar and be readable. Because the names are not descriptive, the meanings of these variables are carefully stated in comment statements.

```
Private Function Hypotenuse(a As Single, b As Single) As Single
   'Calculate the hypotenuse of a right triangle
   'having sides of lengths a and b
   Hypotenuse = Sqr(a ^ 2 + b ^ 2)
End Function

Private Function FV(p As Single,r As Single,c As Single,n As Single)
   As Single Dim i As Single, m As Single
   'Find the future value of a bank savings account
   'p principal, the amount deposited
   'r annual rate of interest
   'c number of times interest is compounded per year
   'n number of years
   'i interest rate per period
   'm total number of times interest is compounded
   i = r / c
   m = c * n
   FV = p * ((1 + i) ^ m)
End Function
```

EXAMPLE 3

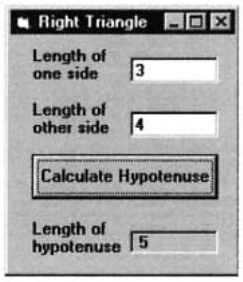
The following program uses the Hypotenuse function.



Object	Property	Setting
frm4_3_3	Caption	Right Triangle
lblSideOne	Caption	Length of one side
txtSideOne	Text	(blank)
lblSideTwo	Caption	Length of other side
txtSideTwo	Text	(blank)
cmdCalculate	Caption	Calculate Hypotenuse
lblHyp	Caption	Length of hypotenuse
picHyp		

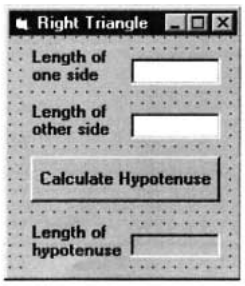
```
Private Sub cmdCalculate_Click()
   Dim a As Single, b As Single
   'Calculate length of the hypotenuse of a right triangle
   a = Val(txtSideOne.Text)
   b = Val(txtSideTwo.Text)
   picHyp.Cls
   picHyp.Print Hypotenuse(a, b)
End Sub

Private Function Hypotenuse(a As Single, b As Single) As Single
   'Calculate the hypotenuse of a right triangle
   'having sides of lengths a and b
   Hypotenuse = Sqr(a ^ 2 + b ^ 2)
End Function
```

[Run, type 3 and 4 into the text boxes, and then click the command button.]



EXAMPLE 4

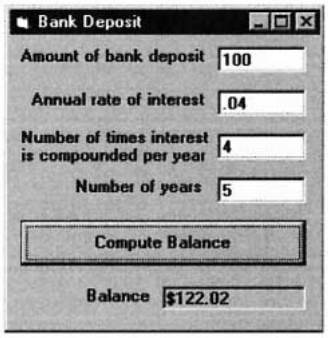
The following program uses the future value function. With the responses shown, the program computes the balance in a savings account when $100 is deposited for 5 years at 4% interest compounded quarterly. Interest is earned 4 times per year at the rate of 1% per interest period. There will be 4 * 5 or interest periods.



Object	Property	Setting
frm4_3_4	Caption	Bank Deposit
lblAmount	Caption	Amount of bank deposit
txtAmount	Text	(blank)
lblRate	Caption	Annual rate of interest
txtRate	Text	(blank)
lblNumComp	Caption	Number of times interest is compounded per year
txtNumComp	Text	(blank)
lblNumYrs	Caption	Number of years
txtNumYrs	Text	(blank)
cmdCompute	Caption	Compute Balance
lblBalance	Caption	Balance
picBalance		

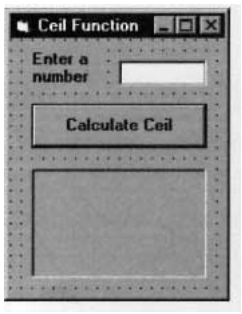
```
 Private Sub cmdCompute_Click()
   Dim p As Single, r As Single, c As Single, n As Single
   'Find the future value of a bank deposit
   Call InputData(p, r, c, n)
   Call DisplayBalance(p, r, c, n)
End Sub

Private Sub DisplayBalance(p As Single,r As Single,c As Single,n As Single)
   Dim balance As Single
   'Display the balance in the picture box
   picBalance.Cls
   balance = FV(p, r, c, n)
   picBalance.Print FormatCurrency(balance)
End Sub

Private Function FV(p As Single,r As Single,c As Single,n As Single) As Single
   Dim i As Single, m As Single
   'Find the future value of a bank savings account
   'p principal, the amount deposited
   'r annual rate of interest
   'c number of times interest is compounded per year
   'n number of years
   'i interest rate per period
   'm total number of times interest is compounded
   i = r / c
   m = c * n
   FV = p * ((1 + i) ^ m)
End Function

Private Sub InputData(p As Single, r As Single, c As Single, n As Single)
   'Get the four values from the text boxes
   p = Val(txtAmount.Text)
   r = Val(txtRate.Text)
   c = Val(txtNumComp.Text)
   n = Val(txtNumYrs.Text)
End Sub
```

[Run, type 100, .04, 4, and 5 into the text boxes, then click the command button.]



EXAMPLE 5

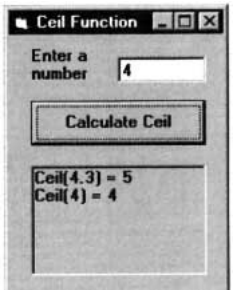
Some computer languages have a useful built-in function called Ceil that is similar to the function Int, except that it rounds noninteger numbers up to the next integer. For instance, Ceil(3.2) is 4 and Ceil(–1.6) is –1. The following program creates Ceil in Visual Basic as a user-defined function.



Object	Property	Setting
frm4_3_5	Caption	Ceil Function
lblNumber	Caption	Enter a number
txtNumber	Text	(blank)
cmdCalculate	Caption	Calculate Ceil
picResults		

```
Private Function Ceil(x As Single) As Single
   'Round nonintegers up
   Ceil = -Int(-x)
End Function

Private Sub cmdCalculate_Click()
   'Demonstrate the Ceil function
   picResults.Print "Ceil("; txtNumber.Text; ") ="; Ceil(Val(txtNumber.Text))
   txtNumber.Text = ""
   txtNumber.SetFocus
End Sub
```

[Run, type 4.3 into the text box, click the command button, type 4 into the text box, and then click the command button again.]



There are many reasons for employing user-defined functions.

1. User-defined functions are consistent with the modular approach to program design. Once we realize a particular function is needed, we can give it a name but save the task of figuring out the computational details until later.
2. Sometimes a single formula must be used several times in a program. Specifying the formula as a function saves repeated typing of the same formula, improves readability, and simplifies debugging.
3. Functions written for one program can be used in other programs. Programmers maintain a collection, or library, of functions that might be needed.

COMMENTS

1. By default, variables passed to a Function procedure are passed by reference; that is, their values are subject to being changed by the Function procedure. Variables also can be passed by value to Function procedures and thereby have their values persist. As with Sub procedures, a variable is passed by value if the variable is enclosed in an extra pair of parentheses when the function is invoked, or if the corresponding parameter in the Private Function statement is

preceded with the word ByVal. Built-in functions have all their arguments passed by value. Some programmers feel that "by value" should have been the default for Function procedures, rather than "by reference."

2. Function procedures can perform the same tasks as Sub procedures. For instance, they can request input and display text; however, they are primarily used to calculate a single value. Normally, Sub procedures are used to carry out other tasks.
3. Function procedures differ from Sub procedures in the way they are accessed. Sub procedures are invoked with Call statements, whereas functions are invoked by placing them where you would otherwise expect to find a constant, variable, or expression. Unlike a Function procedure, a Sub procedure can't be used in an expression.
4. Function procedures can invoke other Function procedures or Sub procedures.
5. Function procedures, like Sub procedures need not have any parameters. Unlike Sub procedures, when a parameterless function is used, the function name may be followed by an empty set of parentheses. The following program uses a "parameterless" function.

```
Private Sub cmdButton_Click()
   'Request and display a saying
   picBox.Cls
   picBox.Print Saying() '() is optional
End Sub

Private Function Saying() As String
   'Retrieve a saying from the user
   Saying = InputBox("What is your favorite saying?")
End Function
```

   [Run, click the command button, and then type *Less is more*. into the message box.]

   The saying *Less is more*. is displayed in the picture box.

6. An alternative method of creating a Function procedure is to move the cursor to a blank line outside of any procedure, type Private Function *FunctionName*, and press the Enter key.

3.4 MODULAR DESIGN

■ Top-Down Design

Large problems usually require large programs. One method programmers use to make a large problem more understandable is to divide it into smaller, less complex subproblems. Repeatedly using a "divide-and-conquer" approach to break up a large problem into smaller subproblems is called **stepwise refinement**. Stepwise refinement is part of a larger methodology of writing programs known as **top-down design**. The term top-down refers to the fact that the more general tasks occur near the top of the design and tasks representing their refinement occur below. Top-down design and structured programming emerged as techniques to enhance programming productivity. Their use leads to programs that are easier to read and maintain. They also produce programs containing fewer initial errors, with these errors being easier to find and correct. When such programs are later modified, there is a much smaller likelihood of introducing new errors.

The goal of top-down design is to break a problem into individual tasks, or modules, that can easily be transcribed into pseudocode, flowcharts, or a program. First, a problem is restated as several simpler problems depicted as modules. Any modules that remain too complex are broken down further. The process of refining modules continues until the smallest

modulnes can be coded directly. Each stage of refinement adds a more complete specification of what tasks must be performed. The main idea in top-down design is to go from the general to the specific. This process of dividing and organizing a problem into tasks can be pictured using a hierarchy chart. When using top-down design, certain criteria should be met:

1. The design should be easily readable and emphasize small module size.
2. Modules proceed from general to specific as you read down the chart.
3. The modules, as much as possible, should be single-minded. That is, they should only perform a single well-defined task.
4. Modules should be as independent of each other as possible, and any relationships among modules should be specified.

This process is illustrated with the following example.

EXAMPLE 1

Write a hierarchy chart for a program that gives certain information about a car loan. The amount of the loan, the duration (in years), and the interest rate should be input. The output should consist of the monthly payment and the amount of interest paid during the first month.

SOLUTION:

In the broadest sense, the program calls for obtaining the input, making calculations, and displaying the output. Figure 3-5 shows these tasks as the first row of a hierarchy chart.

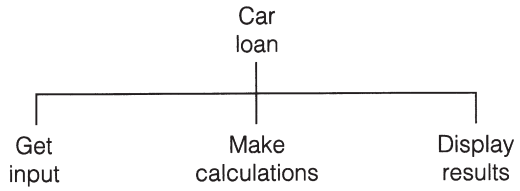


FIGURE 3-5 *Beginning of a Hierarchy Chart for the Car Loan Program*

Each of these tasks can be refined into more specific subtasks. (See Figure 3-6 for the final hierarchy chart.) Most of the subtasks in the second row are straightforward and so do not require further refinement. For instance, the first month's interest is computed by multiplying the amount of the loan by one-twelfth of the annual rate of interest. The most complicated subtask, the computation of the monthly payment, has been broken down further. This task is carried out by applying a standard formula found in finance books; however, the formula requires the number of payments.

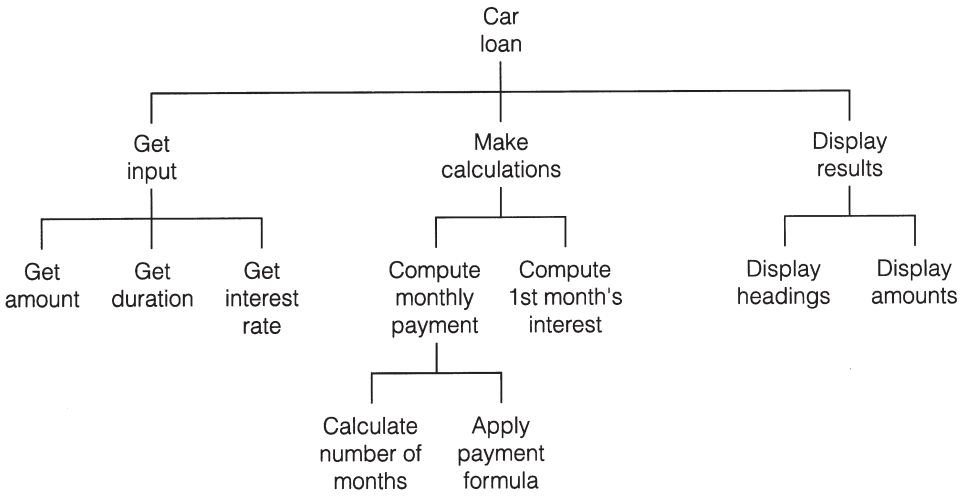


FIGURE 3-6 *Hierarchy Chart for the Car Loan Program*

It is clear from the hierarchy chart that the top modules manipulate the modules beneath them. While the higher-level modules control the flow of the program, the lower-level modules do the actual work. By designing the top modules first, specific processing decisions can be delayed.

Structured Programming

A program is said to be **structured** if it meets modern standards of program design. Although there is no formal definition of the term **structured program**, computer scientists are in uniform agreement that such programs should have modular design and use only the three types of logical structures discussed in Section 1: sequences, decisions, and loops.

Sequences: Statements are executed one after another.

Decisions: One of two blocks of program code is executed based on a test for some condition.

Loops (iteration): One or more statements are executed repeatedly as long as a specified condition is true.

Sections 4 and 5 are devoted to decisions and loops, respectively.

One major shortcoming of the earliest programming languages was their reliance on the GoTo statement. This statement was used to branch (that is, jump) from one line of a program to another. It was common for a program to be composed of a convoluted tangle of branchings that produced confusing code referred to as *spaghetti code*. At the heart of structured programming is the assertion of E. W. Dijkstra that GoTo statements should be eliminated entirely because they lead to complex and confusing programs. Two Italians, C. Bohm and G. Jacopini, were able to prove that GoTo statements are not needed and that any program can be written using only the three types of logic structures discussed before.

Structured programming requires that all programs be written using sequences, decisions, and loops. Nesting of such statements is allowed. All other logical constructs, such as GoTos, are not allowed. The logic of a structured program can be pictured using a flowchart that flows smoothly from top to bottom without unstructured branching (GoTos). The portion of a flowchart shown in Figure 3-7(a) contains the equivalent of a GoTo statement and, therefore, is not structured. A correctly structured version of the flowchart in which the logic flows from the top to the bottom appears in Figure 3-7(b).

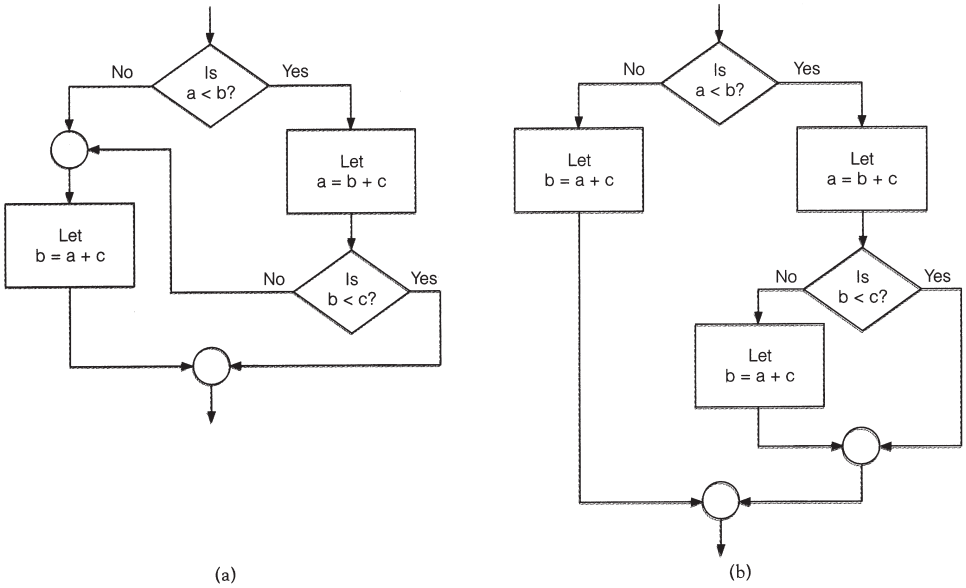


FIGURE 3-7 *Flowcharts Illustrating the Removal of a GoTo Statement*

Advantages of Structured Programming

The goal of structured programming is to create correct programs that are easy to write, understand, and change. Let us now take a closer look at the way modular design, along with a limited number of logical structures, contributes to attaining these goals.

1. *Easy to write.*

   Modular design increases the programmer's productivity by allowing him or her to look at the big picture first and focus on the details later. During the actual coding, the programmer works with a manageable chunk of the program and does not have to think about an entire complex program.

   Several programmers can work on a single large program, each taking responsibility for a specific module.

   Studies have shown structured programs require significantly less time to write than standard programs.

   Often, procedures written for one program can be reused in other programs requiring the same task. Not only is time saved in writing a program, but reliability is enhanced, because reused procedures will already be tested and debugged. A procedure that can be used in many programs is said to be **reusable**.

2. *Easy to debug.*

   Because each procedure is specialized to perform just one task, a procedure can be checked individually to determine its reliability. A dummy program, called a **driver**, is set up to test the procedure. The driver contains the minimum definitions needed to call the procedure to be tested. For instance, if the procedure to be tested is a function, the driver program assigns diverse values to the arguments and then examines the corresponding function value. The arguments should contain both typical and special-case values.

   The program can be tested and debugged as it is being designed with a technique known as **stub programming**. In this technique, the key event procedures and perhaps some of the smaller procedures are coded first. Dummy procedures, or stubs, are written for the remaining procedures. Initially, a stub procedure might consist of a Print method to indicate that the procedure has been called, and thereby confirm that the procedure was called at the right time. Later, a stub might simply display values passed to it in order to confirm not only that the procedure was called, but also that it received the correct values from the calling procedure. A stub also can assign new values to one or more of its parameters to simulate either input or computation. This provides greater control of the conditions being tested. The stub procedure is always simpler than the actual procedure it represents. Although the stub program is only a skeleton of the final program, the program's structure can still be debugged and tested. (The stub program consists of some coded procedures and the stub procedures.)

   Old-fashioned unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such a program is cluttered with details and therefore difficult to follow. Needed tasks are easily left out and crucial details easily neglected. Tricky parts of the program cannot be isolated and examined. Bugs are difficult to locate because they might be present in any part of the program.

3. *Easy to understand.*

   The interconnections of the procedures reveal the modular design of the program.

The meaningful procedure names, along with relevant comments, identify the tasks performed by the modules.

The meaningful variable names help the programmer to recall the purpose of each variable.

4. *Easy to change*.

   Because a structured program is self-documenting, it can easily be deciphered by another programmer.

   Modifying a structured program often amounts to inserting or altering a few procedures rather than revising an entire complex program. The programmer does not even have to look at most of the program. This is in sharp contrast to the situation with unstructured programs that require an understanding of the entire logic of the program before any changes can be made with confidence.

SUMMARY

1. A *general procedure* is a portion of a program that is accessed by event procedures or other general procedures. The two types of general procedures are *Sub procedures* and *Function procedures*.
2. Sub procedures are defined in blocks beginning with Sub statements and ending with End Sub statements. They are accessed by Call statements.
3. Function procedures are defined in blocks beginning with Function statements and ending with End Function statements. A function is activated by a reference in an expression and returns a value.
4. In any procedure, the arguments appearing in the calling statement must match the parameters of the Sub or Function statement in number, type, and order. They need not match in name.
5. A variable declared in the (Declarations) section of (General) is *form-level*. Such a variable is available to every procedure in the form's code and retains its value from one procedure invocation to the next. Form-level variables are often initialized in the Form_Load event procedure.
6. A variable appearing inside a procedure is *local* to the procedure if it is declared in a Dim statement within the procedure or if it is not a form-level variable and does not appear in the parameter list. The values of these variables are reinitialized each time the procedure is called. A variable with the same name appearing in another part of the program is treated as a different variable.
7. Structured programming uses modular design to refine large problems into smaller subproblems. Programs are coded using the three logical structures of sequences, decisions, and loops.

PROGRAMMING PROJECTS

1. The numbers of calories per gram of carbohydrate, fat, and protein are 4, 9, and 4, respectively. Write a program that requests the nutritional content of a 1-ounce serving of food and displays the number of calories in the serving. The input and output should be handled by Sub procedures and the calories computed by a function. A sample run for a typical breakfast cereal is shown in Figure 3-8.

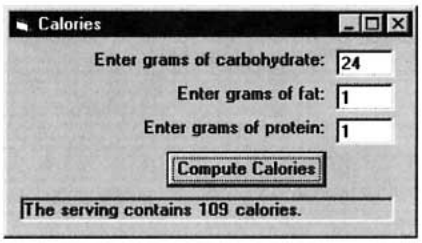


FIGURE 3-8 *Sample Run for Programming Project 1*

2. About seven million notebook computers were sold during 1997. Table 3.2 gives the market share for the four largest vendors. Write a program that displays the number of computers sold by each of the Big Four. The input and output should be handled by Sub procedures and the number of computers calculated by a Function procedure.

TABLE 3.2
1997 Market Shares of the Top Notebook Vendors

Company	Market Share
Toshiba	20%
IBM	11%
Compaq	9%
Dell	6%

Source: PC Magazine, January 20, 1998.

3. Table 3.3 gives the advertising expenditures (in millions of dollars) for the four most advertised soft drink brands during the first nine months of 1995 and 1996. Write a program that displays the percentage change in advertising for each brand. Sub procedures should be used for input and output and the percentage change should be computed with a Function procedure. ***Note:*** The percentage change is 100 * ([1996 expenditure] – [1995 expenditure]) / [1995 expenditure].

TABLE 3.3
Most Advertised Soft Drinks

Brand	1995 Expenditure	1996 Expenditure
Coca-Cola classic	60.7	121.6
Pepsi-Cola	94.8	83.0
Diet Coke	43.7	70.0
Dr. Pepper	46.3	51.8

Source: Beverage World, March 1997.

4. A fast-food vendor sells pizza slices ($1.25), fries ($1.00), and soft drinks ($.75). Write a program to compute a customer's bill. The program should request the quantity of each item ordered in a Sub procedure, calculate the total cost with a Function procedure, and use a Sub procedure to display an itemized bill. A sample output is shown in Figure 3-9.

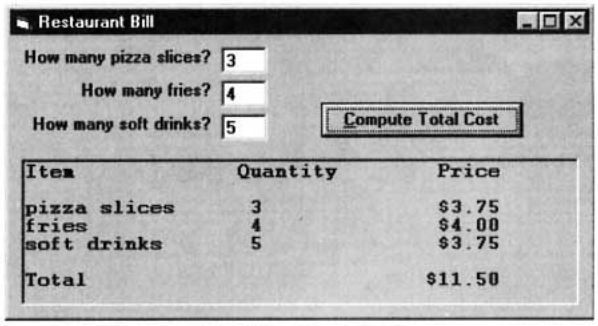


FIGURE 3-9 *Sample Run for Programming Project 4*

5. Write a program to generate a Business Travel Expense attachment for an income tax return. The program should request as input the name of the organization visited, the date and location of the visit, and the expenses for meals and entertainment, airplane fare, lodging, and taxi fares. (Only 50% of the expenses for meals and entertainment are deductible.) A possible form layout and run are shown in Figures 3.10 and 3.11, respectively. Sub procedures should be used for the input and output.

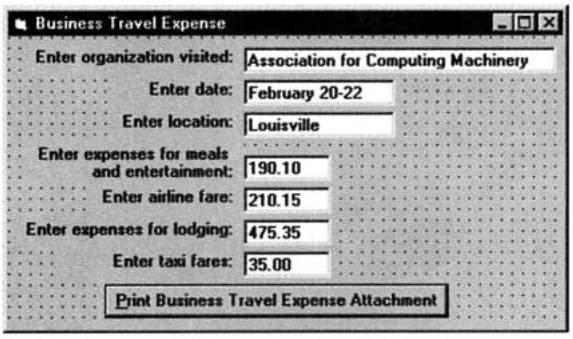


FIGURE 3-10 *Form with Sample Data for Programming Project 5*

```
Business Travel Expense
Trip to attend meeting of
Association for Computing Machinery
February 25-28 in Atlanta

Meals and entertainment  $190.10
Airplane fare            $210.15
Lodging                  $475.35
Taxi fares               $35.00

Total other than Meals and Entertainment: $720.50
50% of Meals and Entertainment: $95.05
```

FIGURE 3-11 *Output on Printer for Sample Run of Programming Project 5*

SECTION

4

DECISIONS

4.1 RELATIONAL AND LOGICAL OPERATORS

A **condition** is an expression involving relational operators (such as < and =) that is either true or false. Conditions also may incorporate logical operators (such as And, Or, and Not).

The relational operator *less than* (<) can be applied to both numbers and strings. The number *a* is said to be less than the number *b* if *a* lies to the left of *b* on the number line. For instance, 2 < 5, –5 < –2, and 0 < 3.5.

The string *a* is said to be less than the string *b* if *a* precedes *b* alphabetically when using the ANSI (or ASCII) table to alphabetize their values. For instance, "cat" < "dog", "cart" < "cat", and "cat" < "catalog". Digits precede uppercase letters, which precede lowercase letters. Two strings are compared working from left to right, character by character, to determine which one should precede the other. Therefore, "9W" < "bat", "Dog" < "cat", and "Sales-99" < "Sales-retail".

Table 4.1 shows the different mathematical relational operators, their representations in Visual Basic, and their meanings.

TABLE 4.1
Relational Operators

Mathematical Notation	Visual Basic Notation	Numeric Meaning	String Meaning
=	=	equal to	identical to
≠	<>	unequal to	different from
<	<	less than	precedes alphabetically
>	>	greater than	follows alphabetically
≤	<=	less than or equal to	precedes alphabetically or is identical to
≥	>=	greater than or equal to	follows alphabetically or is identical to

EXAMPLE 1

Determine whether each of the following conditions is true or false.

(a) 1 <= 1
(b) 1 < 1
(c) "car" "cat"
(d) "Dog" < "dog"

SOLUTION:

(a) True. The notation <= means "less than *or* equal to." That is, the condition is true provided either of the two circumstances holds. The second one (equal to) holds.

(b) False. The notation < means "strictly less than" and no number can be strictly less than itself.

(c) True. The characters of the strings are compared one at a time working from left to right. Because the first two match, the third character decides the order.

(d) True. Because uppercase letters precede lowercase letters in the ANSI table, the first character of "Dog" precedes the first character of "dog."

Conditions also can involve variables, numeric operators, and functions. To determine whether a condition is true or false, first compute the numeric or string values and then decide if the resulting assertion is true or false.

EXAMPLE 2

Suppose the numeric variables *a* and *b* have values 4 and 3, and the string variables *c* and *d* have values "hello" and "bye". Are the following conditions true or false?

(a) (a + b) < 2 * a

(b) (Len(c) – b) = (a / 2)

(c) c < ("good" & d)

SOLUTION:

(a) The value of a + b is 7 and the value of 2 * a is 8. Because 7 < 8, the condition is true.

(b) True, because the value of Len(c) – b is 2, the same as (a / 2).

(c) The condition "hello" < "goodbye" is false, because "h" follows "g" in the ANSI table.

LOGICAL OPERATORS

Programming situations often require more complex conditions than those considered so far. For instance, suppose we would like to state that the value of a numeric variable, *n*, is strictly between 2 and 5. The proper Visual Basic condition is

(2 < n) And (n < 5)

The condition (2 < n) And (n < 5) is a combination of the two conditions 2 < n and n < 5 with the logical operator And.

The three main logical operators are And, Or, and Not. If *cond1* and *cond2* are conditions, then the condition

```
cond1 And cond2
```

is true if both *cond1* and *cond2* are true. Otherwise, it is false. The condition

```
cond1 Or cond2
```

is true if either *cond1* or *cond2* (or both) is true. Otherwise, it is false. The condition

```
Not cond1
```

is true if *cond1* is false, and is false if *cond1* is true.

EXAMPLE 3

Suppose the numeric variable *n* has value 4 and the string variable *answ* has value "Y". Determine whether each of the following conditions is true or false.

(a) (2 < n) And (n < 6)

(b) (2 < n) Or (n = 6)

(c) Not (n < 6)

(d) (answ = “Y”) Or (answ = “y”)

(e) (answ = “Y”) And (answ = “y”)

(f) Not (answ = “y”)

(g) ((2 < n) And (n = 5 + 1)) Or (answ = “No”)

(h) ((n = 2) And (n = 7)) Or (answ = “Y”)

(i) (n = 2) And ((n = 7) Or (answ = “Y”))

SOLUTION:

(a) True, because the conditions (2 < 4) and (4 < 6) are both true.

(b) True, because the condition (2 < 4) is true. The fact that the condition (4 = 6) is false does not affect the conclusion. The only requirement is that at least one of the two conditions be true.

(c) False, because (4 < 6) is true.

(d) True, because the first condition becomes (“Y” = “Y”) when the value of *answ* is substituted for *answ*.

(e) False, because the second condition is false. Actually(isTDD0 Tc2o)55.for *(g) ((2 < n) And (n = 5 + 1)) Or (answ =*

The hierarchy chart in Figure 5-6 shows the stepwise refinement of the problem.

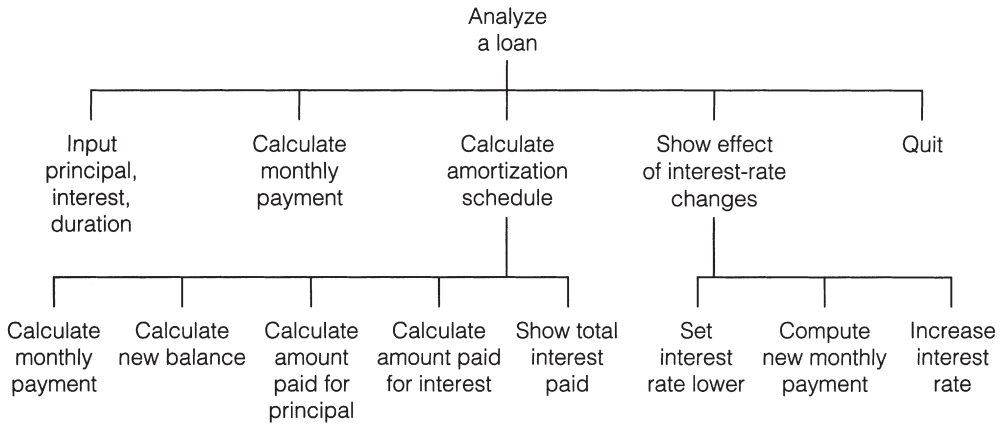


FIGURE 5-6 *Hierarchy Chart for the Analyze a Loan Program*

The User Interface

Figure 5-7 shows a possible form design. Figures 5-8 and 5-9 show possible runs of the program for each task available through the command buttons. The width and height of the picture box were adjusted by trial and error to handle the extensive output generated.

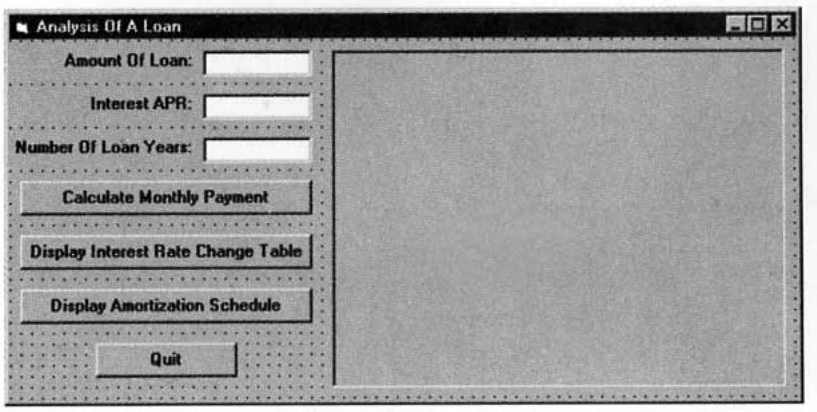


FIGURE 5-7 *Template for Loan Analysis*

TABLE 5.1
Objects and Initial Properties for the Loan Analysis Program

Object	Property	Setting
frmLoan	Caption	Analysis of a Loan
lblAmt	Alignment	1 – Right Justify
Caption	Amount of Loan:	
txtAmt	Text	(blank)
lblApr	Alignment	1 – Right Justify
Caption	Interest APR:	
txtApr	Text	(blank)
lblYrs	Alignment	1 – Right Justify
Caption	Number of Loan Years	
txtYrs	Text	(blank)
cmdPayment	Caption	Calculate Monthly Payment
cmdRateTable	Caption	Display Interest Rate Change Table
cmdAmort	Caption	Display Amortization Schedule
cmdQuit	Caption	Quit

picDisp

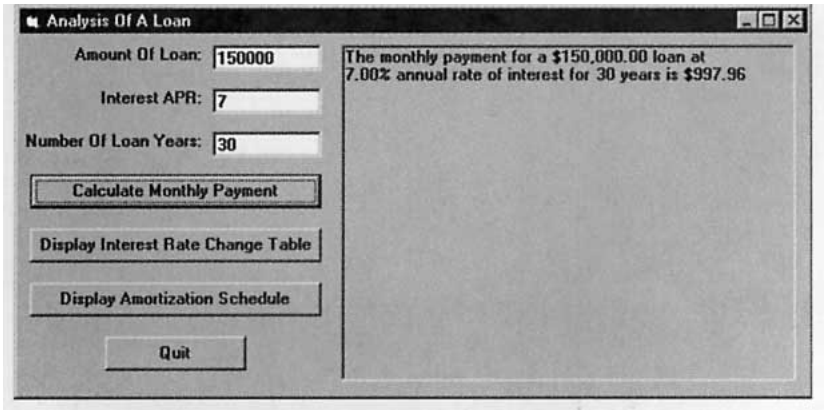


FIGURE 5-8 *Monthly Payment on a 30-Year Loan*

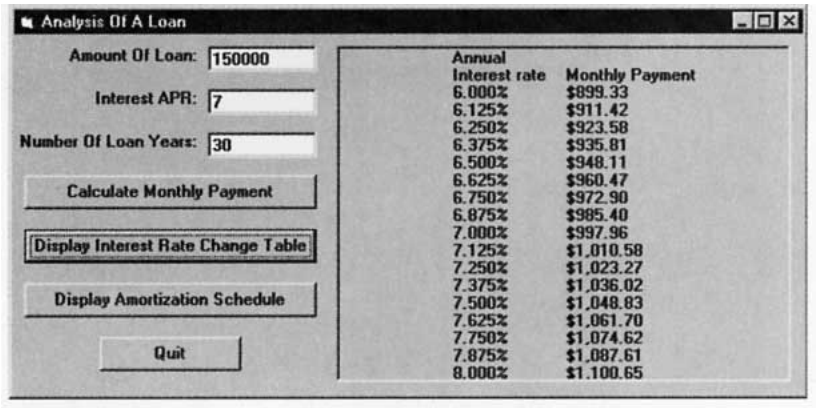


FIGURE 5-9 *Interest Rate Change Table for a 30-Year Loan*

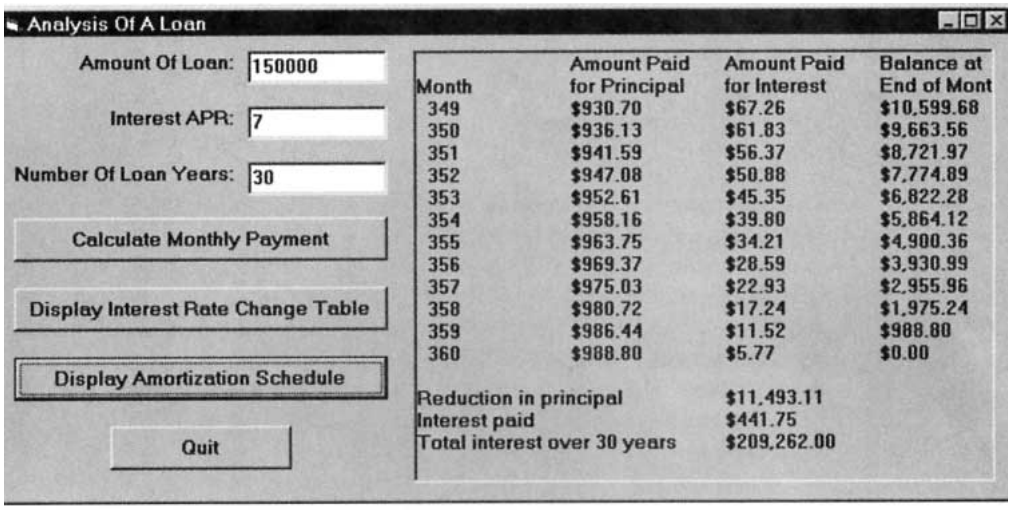


FIGURE 5-10 *Amortization of Year 30 of a Loan*

■ WRITING THE ANALYZE-A-LOAN PROGRAM

Table 5.2 shows each task discussed before and the procedure that carries out the task.

Pseudocode for the Analyze-a-Loan Program

Calculate Monthly Payment command button:
    INPUT LOAN DATA (Sub procedure InputData)
    COMPUTE MONTHLY PAYMENT (Function Payment)
    DISPLAY MONTHLY PAYMENT (Sub procedure ShowPayment)

Display Interest Rate Change Table command button:
    INPUT LOAN DATA (Sub procedure InputData)
    DISPLAY INTEREST RATE CHANGE TABLE
        (Sub procedure ShowInterestChanges)
    Decrease annual rate by .01
    Do
        Display monthly interest rate
        COMPUTE MONTHLY PAYMENT (Function Payment)
        Increase annual rate by .00125
    Loop Until annual rate > original annual rate + .01

Display Amortization Schedule command button:
    INPUT LOAN DATA (Sub procedure InputData)
    DISPLAY AMORTIZATION SCHEDULE (Sub procedure ShowAmortSched)
    Compute monthly interest rate
    COMPUTE MONTHLY PAYMENT (Function Payment)
    Display amortization table
    Display total interest paid

TABLE 5.2
Tasks and Their Procedures

Task	Procedure
1. Input principal, interest, duration.	InputData
2. Calculate monthly payment.	ShowPayment
3. Calculate amortization schedule.	ShowAmortSched
3.1 Calculate monthly payment.	Payment
3.2 Calculate new balance.	Balance
3.3 Calculate amount paid for loan.	ShowAmortSched
3.4 Calculate amount paid for interest.	ShowAmortSched
4. Show effect of interest rate changes.	ShowInterestChanges
4.1 Reduce interest rate.	ShowInterestChanges
4.2 Compute new monthly payment.	Payment
4.3 Increase interest rate.	ShowInterestChanges

```
'Analyze a loan

Private Function Balance(mPayment As Single, prin As Single, mRate As Single) As Single
   Dim newBal As Single 'Compute balance at end of month
   newBal = (1 + mRate) * prin
   If newBal <= mPayment Then
        mPayment = newBal
        Balance = 0
     Else
        Balance = newBal - mPayment
   End If
End Function

Private Sub cmdAmort_Click()
   Dim principal As Single       'Amount of loan
   Dim yearlyRate As Single      'Annual rate of interest
   Dim numMonths As Integer      'Number of months to repay loan
```

```
   Call InputData(principal, yearlyRate, numMonths)
   Call ShowAmortSched(principal, yearlyRate, numMonths)
End Sub

Private Sub cmdPayment_Click()
   Dim principal As Single       'Amount of loan
   Dim yearlyRate As Single      'Annual rate of interest
   Dim numMonths As Integer      'Number of months to repay loan
   Call InputData(principal, yearlyRate, numMonths)
   Call ShowPayment(principal, yearlyRate, numMonths)
End Sub

Private Sub cmdQuit_Click()
   End
End Sub

Private Sub cmdRateTable_Click()
   Dim principal As Single       'Amount of loan
   Dim yearlyRate As Single      'Annual rate of interest
   Dim numMonths As Integer      'Number of months to repay loan
   Call InputData(principal, yearlyRate, numMonths)
   Call ShowInterestChanges(principal, yearlyRate, numMonths)
End Sub

Private Sub InputData(prin As Single, yearlyRate As Single, numMs As Integer)
   Dim percentageRate As Single, numYears As Integer
   'Input the loan amount, yearly rate of interest, and duration
   prin = Val(txtAmt.Text)
   percentageRate = Val(txtApr.Text)
   numYears = Val(txtYrs.Text)
   yearlyRate = percentageRate / 100
   numMs = numYears * 12
End Sub

Private Function Payment(prin As Single, mRate As Single, numMs As Integer) As Single
   Dim payEst As Single
   If numMs = 0 Then
        payEst = prin
     ElseIf mRate = 0 Then
        payEst = prin / numMs
     Else
        payEst = prin * mRate / (1 - (1 + mRate) ^ (-numMs))
   End If
   If payEst <> Round(payEst, 2) Then
        Payment = Round(payEst + .005, 2) 'round up to nearest cent
   End If
End Function

Private Sub ShowAmortSched(prin As Single, yearlyRate As Single, numMs As Integer)
   Dim msg As String, startMonth As Integer, mRate As Single
   Dim monthlyPayment As Single, totalInterest As Single
   Dim yearInterest As Single, oldBalance As Single
   Dim monthNum As Integer, newBalance As Single
   Dim principalPaid As Single, interestPaid As Single
   Dim reducPrin As Single, loanYears As Integer
   'Display amortization schedule
```

```
    msg = "Please enter year (1-"& Str(numMs / 12)
    msg = msg & ") for which amorization is to be shown:"
    startMonth = 12 * Val(InputBox(msg)) - 11 picDisp.Cls
    picDisp.Print "", "Amount Paid ",
    picDisp.Print "Amount Paid", "Balance at"
    picDisp.Print "Month", "for Principal",
    picDisp.Print "for Interest", "End of Month"
    mRate = yearlyRate / 12 'monthly rate
    monthlyPayment = Payment(prin, mRate, numMs)
    totalInterest = 0
    yearInterest = 0
    oldBalance = prin
    For monthNum = 1 To numMs
      newBalance = Balance(monthlyPayment, oldBalance, mRate)
      principalPaid = oldBalance - newBalance
      interestPaid = monthlyPayment - principalPaid
      totalInterest = totalInterest + interestPaid
      If (monthNum >= startMonth) And (monthNum <= startMonth + 11) Then
        picDisp.Print Tab(2); FormatNumber(monthNum, 0),
        picDisp.Print FormatCurrency(principalPaid),
        picDisp.Print FormatCurrency(interestPaid),
        picDisp.Print FormatCurrency(newBalance)
        yearInterest = yearInterest + interestPaid
      End If
      oldBalance = newBalance
    Next monthNum
    reducPrin = 12 * monthlyPayment - yearInterest
    loanYears = numMs / 12
    picDisp.Print
    picDisp.Print "Reduction in principal",
    picDisp.Print FormatCurrency(reducPrin)
    picDisp.Print "Interest paid", ,
    picDisp.Print FormatCurrency(yearInterest)
    picDisp.Print "Total interest over"; loanYears; "years",
    picDisp.Print FormatCurrency(totalInterest)
End Sub

Private Sub ShowInterestChanges(prin As Single, yearlyRate As Single, numMs As Integer)
    Dim newRate As Single, mRate As Single, py As Single
    Dim pymnt As String
    'Display affect of interest changes
    picDisp.Cls
    picDisp.Print , "Annual"
    picDisp.Print , "Interest rate", "Monthly Payment"
    newRate = yearlyRate - .01
    Do
      mRate = newRate / 12 'monthly rate
      py = Payment(prin, mRate, numMs)
      pymnt = FormatCurrency(py)
      picDisp.Print , FormatPercent(newRate, 3) , pymnt
      newRate = newRate + .00125
    Loop Until newRate > yearlyRate + .01
End Sub

Private Sub ShowPayment(prin As Single, yearlyRate As Single, numMs As Integer)
    Dim mRate As Single, prn As String, apr As String
```

```
   Dim yrs As String, pay As Single, pymnt As String
   'Display monthly payment amount
   mRate = yearlyRate / 12 'monthly rate
   prn = FormatCurrency(prin)
   apr = FormatNumber(yearlyRate * 100)
   yrs = FormatNumber(numMs / 12, 0)
   pay = Payment(prin, mRate, numMs)
   pymnt = FormatCurrency(pay)
   picDisp.Cls
   picDisp.Print "The monthly payment for a "& prn & "loan at "
   picDisp.Print apr & "% annual rate of interest for ";
   picDisp.Print yrs & "years is "& pymnt
End Sub
```

COMMENTS

1. Tasks 3.1 and 3.2 are performed by functions. Using functions to compute these quantities simplifies the computations in ShowAmortSched.
2. Because the payment was rounded up to the nearest cent, it is highly likely that the payment needed in the final month to pay off the loan will be less than the normal payment. For this reason, ShowAmortSched checks if the balance of the loan (including interest due) is less than the regular payment, and if so, makes appropriate adjustments.
3. The standard formula for computing the monthly payment cannot be used if either the interest rate is zero percent or the loan duration is zero months. Although both of these situations do not represent reasonable loan parameters, provisions are made in the function Payment so that the program can handle these esoteric situations.

SUMMARY

1. A Do loop repeatedly executes a block of statements either as long as or until a certain condition is true. The condition can be checked either at the top of the loop or at the bottom.
2. The EOF function tells us if we have read to the end of a file.
3. As various items of data are processed by a loop, a *counter* can be used to keep track of the number of items, and an *accumulator* can be used to sum numerical values.
4. A *flag* is a Boolean variable, used to indicate whether or not a certain event has occurred.
5. A For...Next loop repeats a block of statements a fixed number of times. The *control variable* assumes an initial value and increments by one after each pass through the loop until it reaches the terminating value. Alternative increment values can be specified with the Step keyword.

PROGRAMMING PROJECTS

1. Write a program to display a company's payroll report in a picture box. The program should read each employee's name, hourly rate, and hours worked from a

file and produce a report in the form of the sample run shown in Figure 5-11. Employees should be paid time-and-a-half for hours in excess of 40.

```
Payroll Report for Week ending 4/15/99

Employee      Hourly Rate    Hours Worked    Gross Pay
Al Adams      $6.50               38         $247.00
Bob Brown     $5.70               50         $313.50
Carol Coe     $7.00               40         $280.00
FinalTotal    $840.50
```

FIGURE 5-11 *Sample Output from Programming Project 1*

2. Table 5.3 shows the standard prices for items in a department store. Suppose prices will be reduced for the annual George Washington's Birthday Sale. The new price will be computed by reducing the old price by 10 percent, rounding up to the nearest dollar, and subtracting 1 cent. If the new price is greater than the old price, the old price is used as the sale price. Write a program to display in a picture box the output shown in Figure 5-12.

TABLE 5.3
Washington's Birthday Sale

Item	Original Price
GumShoes	39.00
SnugFoot Sandals	21.00
T-Shirt	7.75
Maine Handbag	33.00
Maple Syrup	6.75
Flaked Vest	24.00
Nightshirt	26.00

Item	Sale Price
GumShoes	35.99
SnugFoot Sandals	18.99
T-Shirt	6.99
Maine Handbag	29.99
Maple Syrup	6.75
Flaked Vest	21.99
Nightshirt	23.99

FIGURE 5-12 *Output of Project 2*

3. The Rule of 72 is used to make a quick estimate of the time required for prices to double due to inflation. If the inflation rate is r percent, then the Rule of 72 estimates that prices will double in 72/*r* years. For instance, at an inflation rate of 6 percent, prices double in about 72/6 or 12 years. Write a program to test the accuracy of this rule. The program should display a table showing, for each value of r from 1 to 20, the rounded value of 72/*r* and the actual number of years required for prices to double at an *r* percent inflation rate. (Assume prices increase at the end of each year.) Figure 5-13 shows the first few rows of the output.

Interest Rate (%)	Rule of 72	Actual
1	72	70
2	36	36
3	24	24

FIGURE 5-13 *Rule of 72*

4. Table 5.4 shows the number of bachelor degrees conferred in 1980 and 1994 in certain fields of study. Tables 5.5 and 5.6 show the percentage change and a his-

togram of 1994 levels, respectively. Write a program that allows the user to display any one of these tables as an option and quit as a fourth option.

TABLE 5.4
Bachelor Degrees Conferred in Certain Fields

Field of Study	1980	1994
Business and management	185,361	246,654
Computer and info. science	11,154	24,200
Education	118,169	167,600
Engineering	68,893	62,220
Social sciences	103,519	133,680

Source: U.S. National Center of Educational Statistics

TABLE 5.5
Percentage Change in Bachelor Degrees Conferred

Field of Study	% Change (1980–1994)
Business and management	33.1
Computer and info. science	117.0
Education	41.8
Engineering	–9.7
Social sciences	29.1

TABLE 5.6
Bachelor Degrees Conferred in 1994 in Certain Fields

Business and management	********** 246,654
Computer and info. science	*** 24,200
Education	********** 167,600
Engineering	****** 62,220
Social sciences	********** 133,680

5. *Least-Squares Approximation*. Table 5.7 shows the 1988 price of a gallon of fuel and the consumption of motor fuel for several countries. Figure 5-14 displays the data as points in the *xy* plane. For instance, the point with coordinates (1, 1400) corresponds to the USA. Figure 5-14 also shows the straight line that best fits these data in the least squares sense. (The sum of the squares of the distances of the 11 points from this line is as small as possible.) In general, if (x_i, y_1), (x_2, y_2), . . , (x_n, y_n) are n points in the xy coordinate system, then the least squares approximation to these points is the line $y = mx + b$, where

$$m = \frac{n * (\text{sum of } x_i * y_i) - (\text{sum of } x_i) * *=(\text{sum of } y_i)}{n * (\text{sum of } x_i * x_i) - (\text{sum of } x_i)^2}$$

and

$$\text{b} = ((\text{sum of } y_i) - m * (\text{sum of } x_i))/n$$

Write a program that calculates and displays the equation of the least squares line, and then allows the user to enter a fuel price and uses the equation of the line to predict the corresponding consumption of motor fuel. (Place the numeric data from the table in a data file.) A sample run is shown in Figure 5-15.

TABLE 5.7
A Comparison of 1988 Fuel Prices and Per Capita Motor Fuel Use

Country	Price per gallon in U.S. Dollars	Tons of Oil per 1000 Person	Country	Price per gallon in U.S. Dollars	Tons of Oil per 1000 Person
USA	$1.00	1400	France	$3.10	580
W. Ger.	$2.20	620	Norway	$3.15	600
England	$2.60	550	Japan	$3.60	410
Austria	$2.75	580	Denmark	$3.70	570
Sweden	$2.80	700	Italy	$3.85	430
Holland	$3.00	490			

Source: World Resources Institute

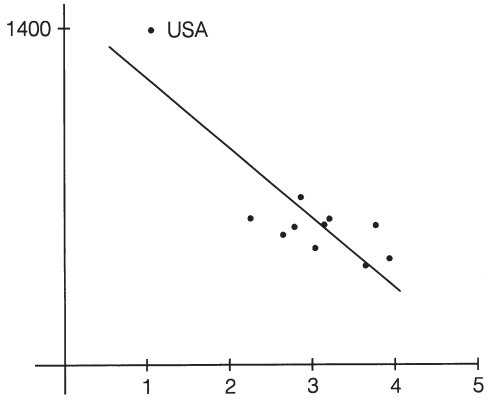


FIGURE 5-14 *Least-Squares Fit to Data from Table 5.7*

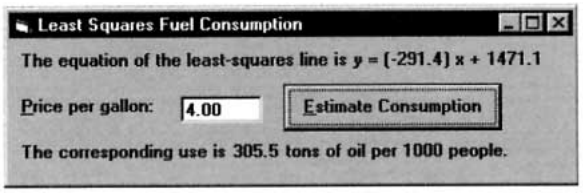


FIGURE 5-15 *Sample Run of Programming Project 5*

6. Write a program to provide information on the height of a ball thrown straight up into the air. The program should request the initial height, *h* feet, and the initial velocity, *v* feet per second, as input. The four options to be provided by command buttons are as follows:

   (a) Determine the maximum height of the ball. ***Note:*** The ball will reach its maximum height after *v*/32 seconds.

   (b) Determine approximately when the ball will hit the ground. ***Hint:*** Calculate the height after every .1 second and observe when the height is no longer a positive number.

   (c) Display a table showing the height of the ball every quarter second for 5 seconds or until it hits the ground.

   (d) Quit.

   The formula for the height of the ball after t seconds, $h + v * t - 16 * t * t$, should be specified in a user-defined function. (Test the program with $v = 148$ and $h = 0$. This velocity is approximately the top speed clocked for a ball thrown by a professional baseball pitcher.)

7. Depreciation to a Salvage Value of 0. For tax purposes an item may be depreciated over a period of several years, *n*. With the straight-line method of depreciation, each year the item depreciates by 1/*n*th of its original value. With the double-declining-balance method of depreciation, each year the item depreciates by 2/*n*ths of its value at the beginning of that year. (In the last year it is depreciated by its value at the beginning of the year.) Write a program that
   (a) Requests a description of the item, the year of purchase, the cost of the item, the number of years to be depreciated (estimated life), and the method of depreciation. The method of depreciation should be chosen by clicking one of two command buttons.
   (b) Displays a depreciation schedule for the item similar to the schedule shown in Figure 5-16.

```
Description: Computer
Year of purchase: 1999
Cost: $2000.00
Estimated life: 5
Method of depreciation: double-declining-balance

          Value at   Amount Deprec   Total Depreciation
Year      Beg of Yr   During Year    to End of Year
1999      2,000.00       800.00      800.00
2000      1,200.00       480.00      1,280.00
2001      720.00         288.00      1,568.00
2002      432.00         172.80      1,740.80
2003      259.20         259.20      2,000.00
```

FIGURE 5-16 *Depreciation Schedule*

8. *The Twelve Days of Christmas.* Each year, Provident National Bank of Philadelphia publishes a Christmas price list. See Table 5.8. Write a program that requests an integer from 1 through 12 and then lists the gifts for that day along with that day's cost. On the *n*th day, the *n* gifts are 1 partridge in a pear tree, 2 turtle doves, . . . *n* of the *n*th item. The program also should give the total cost of all twelve days. As an example, Figure 5-17 shows the output in the picture box when the user enters 3.

TABLE 5.8
Christmas Price Index

Item	Cost	Item	Cost
partridge in a pear tree	27.50	swan-a-swimming	1000.00
turtle dove	25.00	maid-a-milking	4.25
French hen	5.00	lady dancing	289.50
calling bird	70.00	lord-a-leaping	292.50
gold ring	60.00	piper piping	95.75
geese-a-laying	25.00	drummer drumming	95.00

```
The gifts for day 3 are
1 partridge in a pear tree
2 turtle doves
3 French hens
Cost: $92.50

Total cost for the twelve days: $71,613.50
```

FIGURE 5-17 *Sample Output for Programming Project 8*

SECTION

6

ARRAYS

6.1 CREATING AND ACCESSING ARRAYS

A **variable** (or simple variable) is a name to which Visual Basic can assign a single value. An **array variable** is a collection of simple variables of the same type to which Visual Basic can efficiently assign a list of values.

Consider the following situation. Suppose you want to evaluate the exam grades for 30 students. Not only do you want to compute the average score, but you also want to display the names of the students whose scores are above average. You might place the 30 pairs of student names and scores in a data file and run the program outlined.

```
Private Sub cmdButton_Click()
   Dim student1 As String, score1 As Single
   Dim student2 As String, score2 As Single
   Dim student3 As String, score3 As Single
   .
   .
   Dim student30 As String, score30 As Single
   'Analyze exam grades
   Open "SCORES.TXT" For Input As #1
   Input #1, student1, score1
   Input #1, student2, score2
   Input #1, student3, score3
   .
   .
   Input #1, student30, score30
   'Compute the average grade
   .
   .
   'Display names of above average students
   .
   .
End Sub
```

This program is going to be uncomfortably long. What's most frustrating is that the 30 Dim statements and 30 Input # statements are very similar and look as if they should be condensed

into a short loop. A shorthand notation for the many related variables would be welcome. It would be nice if we could just write

```
For i = 1 To 30
   Input #1, studenti, scorei
Next i
```

Of course, this will not work. Visual Basic will treat *studenti* and *scorei* as two variables and keep reassigning new values to them. At the end of the loop they will have the values of the thirtieth student.

Visual Basic provides a data structure called an **array** that lets us do what we tried to accomplish in the loop. The variable names will be similar to those in the Input # statement. They will be

```
student(1), student(2), student(3), ..., student(30)
```

and

```
score(1), score(2), score(3), ..., score(30).
```

We refer to these collections of variables as the array variables *student*( ) and *score*( ). The numbers inside the parentheses of the individual variables are called **subscripts**, and each individual variable is called a **subscripted variable** or **element**. For instance, *student*(3) is the third subscripted variable of the array *student*( ), and *score*( ) is the 20th subscripted variable of the array *score*( ). The elements of an array are assigned successive memory locations. Figure 6-1 shows the memory locations for the array *score*( ).

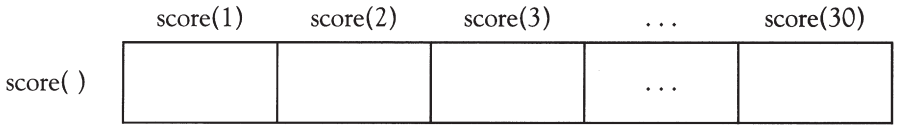


FIGURE 6-1 *The Array Score( )*

Array variables have the same kinds of names as simple variables. If *arrayName* is the name of an array variable and *n* is a whole number, then the statement

```
Dim arrayName(1 To n) As varType
```

placed in the (Declarations) section of (General) reserves space in memory to hold the values of the subscripted variables *arrayName*(1), *arrayName*(2), *arrayName*(3), . . . , *arrayName*(*n*). (Recall from Section 3.1 that the (Declarations) section of (General) is accessed from any code window by selecting these values in the Object and Procedure boxes.) The spread of the subscripts specified by the Dim statement is called the **range** of the array, and the Dim statement is said to **dimension** the array. The subscripted variables will all have the same data type; namely, the type specified by varType. For instance, they could be all String variables or all Integer variables. In particular, the statements

```
Dim student(1 To 30) As String
Dim score(1 To 30) As Integer
```

dimension the arrays needed for the preceding program.

As with any variable created in the (Declarations) section of (General), these array variables are form-level as discussed in Section 3. Recall that form-level variables can be accessed from any procedure in the program and continue to exist and retain their values as long as the program is running.

Values can be assigned to subscripted variables with assignment statements and displayed with Print methods. The statement

```
Dim score(1 To 30) As Integer
```

sets aside a portion of memory for the numeric array *score*( ) and places the default value 0 in each element.

	score(1)	score(2)	score(3)	. . .	score(30)
score( )	0	0	0	. . .	0

The statements

```
score(1) = 87
score(3) = 92
```

assign values to the first and third elements.

	score(1)	score(2)	score(3)	. . .	score(30)
score( )	87	0	92	. . .	0

The statements

```
For i = 1 To 4
   picBox.Print score(i);
Next i
```

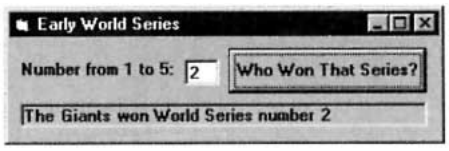
then produce the output 87 0 92 in picBox.

EXAMPLE 1

The following program creates a string array consisting of the names of the first five World Series winners. Figure 6-2 shows the array created by the program.

```
'Create array for five strings
Dim teamName(1 To 5) As String 'in (Declarations) section of (General)

Private Sub cmdWhoWon_Click()
   Dim n As Integer
   'Fill array with World Series Winners
   teamName(1) = "Red Sox"
   teamName(2) = "Giants"
   teamName(3) = "White Sox"
   teamName(4) = "Cubs"
   teamName(5) = "Cubs" '
   Access array of five strings
   n = Val(txtNumber.Text)
   picWinner.Cls
   picWinner.Print "The "; teamName(n); " won World Series number"; n
End Sub
```

[Run, type 2 into the text box, and click the command button.]



	teamName(1)	teamName(2)	teamName(3)	teamName(4)	teamName(5)
teamName( )	Red Sox	Giants	White Sox	Cubs	Cubs

FIGURE 6-2 *The Array TeamName( ) of Example 1*

In Example 1, the array *teamName* was assigned values within the cmdWhoWon_Click event procedure. Every time the command button is clicked, the values are reassigned to the array. This manner of assigning values to an array can be very inefficient, especially in programs with large arrays where the task of the program (in Example 1, looking up a fact) may be repeated numerous times for different user input. When, as in Example 1, the data to be placed in an array are known at the time the program first begins to run, a more efficient location for the statements that fill the array is in Visual Basic's Form_Load event procedure. The Form_Load event procedure is executed by Visual Basic as soon as the program is run, and this execution is guaranteed to occur before the execution of any other event or general procedure in the program. Example 2 uses the Form_Load procedure to improve on Example 1.

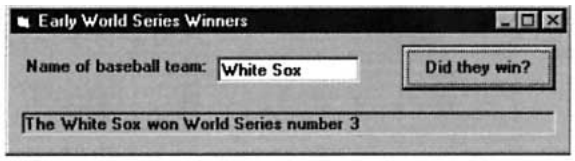
EXAMPLE 2

Modify Example 1 to request the name of a baseball team as input and search the array to determine whether or not the team name appears in the array. Load the array values only once.

```
'Create array for five strings
Dim teamName(1 To 5) As String 'in (Declarations) section of (General)

Private Sub cmdDidTheyWin_Click()
   Dim team As String, foundFlag As Boolean, n As Integer
   'Search for an entry in a list of strings
   team = txtName.Text
   foundFlag = False
   n = 0
   Do
      n = n + 1
      If UCase(teamName(n)) = UCase(team) Then
         foundFlag = True
      End If
   Loop Until (foundFlag = True) Or (n = 5)
   'Above line can be replaced with Loop Until (foundFlag) or (n = 5)
   picWinner.Cls
   If foundFlag = False Then 'Can be replaced by If Not foundFlag Then
      picWinner.Print "The "; team; " did not win any";
      picWinner.Print " of the first five World Series."
   Else
      picWinner.Print "The "; teamName(n); " won World Series number"; n
   End If
End Sub

Private Sub Form_Load()
   'Fill array with World Series winners
   teamName(1) = "Red Sox"
   teamName(2) = "Giants"
   teamName(3) = "White Sox"
    teamName(4) = "Cubs"
   teamName(5) = "Cubs"
```

```
End Sub
```

[Run, type White Sox into the text box, and click the command button.]



We could have written the program in Example 2 with a For...Next loop beginning For *n* = 1 To 5. However, such a loop would unnecessarily search the entire list when the sought-after item is found early. The wasted time could be significant for a large array.

In some applications, arrays are needed only temporarily to help a procedure complete a task. Visual Basic also allows us to create array variables that are local to a specific procedure and that exist temporarily while the procedure is executing. If the statement

```
Dim arrayName(1 To n) As varType
```

is placed inside an event procedure or general procedure, then space for *n* subscripted variables is set aside in memory each time the procedure is invoked and released when the procedure is exited.

In Example 1, values were assigned to the elements of the array with assignment statements. However, data for large arrays are more often stored in a data file and read with Input # statements. Example 3 uses this technique. Also, because the task of the program is likely to be performed only once during a run of the program, a local array is used.

EXAMPLE 3

Table 6.1 gives names and test scores from a mathematics contest given in 1953. Write a program to display the names of the students scoring above the average for these eight students.

TABLE 6.1
The Top Scores on the Fourth Annual Mathematics Contest Sponsored by the Metropolitan NY Section of the MAA

Richard Dolen	135	Paul H. Monsky	150
Geraldine Ferraro	114	Max A. Plager	114
James B. Fraser	92	Robert A. Schade	91
John H. Maltby	91	Barbara M. White	124

Source: The Mathematics Teacher, February 1953.

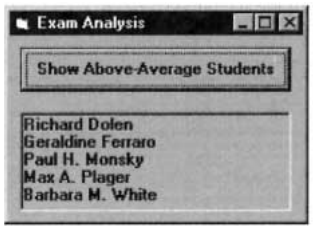
SOLUTION:

The following program creates a string array to hold the names of the contestants and a numeric array to hold the scores. The first element of each array holds data for the first contestant, the second element of each array holds data for the second contestant, and so on. See Figure 6-3. Note that the two arrays can be dimensioned in a single Dim statement by placing a comma between the array declarations.

```
Private Sub cmdShow_Click()
  Dim total As Integer, student As Integer, average As Single
  'Create arrays for names and scores
  Dim nom(1 To 8) As String, score(1 To 8) As Integer
  'Assume the data has been placed in the file "SCORES.TXT"
  '(The first line of the file is "Richard Dolen", 135)
  Open "SCORES.TXT" For Input As #1
  For student = 1 To 8
    Input #1, nom(student), score(student)
  Next student
  Close #1
```

```
  'Analyze exam scores
  total = 0
  For student = 1 To 8
    total = total + score(student)
  Next student
  average = total / 8
  'Display all names with above-average grades
  picTopStudents.Cls
  For student = 1 To 8
    If score(student) > average Then
      picTopStudents.Print nom(student)
    End If
  Next student
End Sub
```

[Run, and click the command button.]



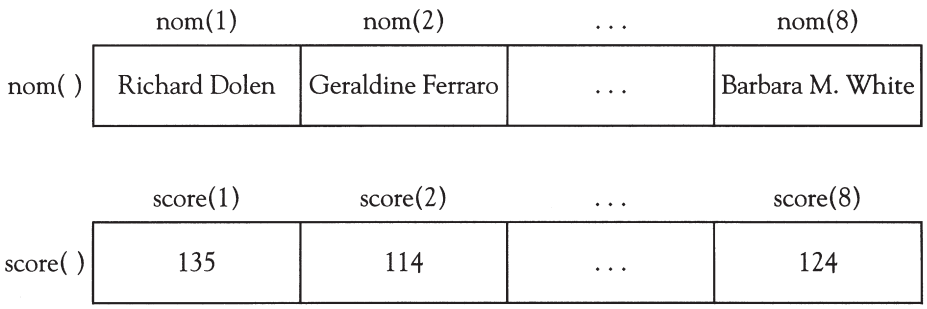


FIGURE 6-3 *Arrays Created by Example 3*

In Example 3, the number of students to be processed had to be known at the time the program was written. In actual practice, the amount of data that a program will be processing is not known in advance. Programs should be flexible and incorporate a method for handling varying amounts of data. Visual Basic makes this possible with the statement

```
ReDim arrayName (1 to n) As varType
```

which can use variables or expressions when indicating the subscript range. However, ReDim statements can only be used inside procedures.

EXAMPLE 4

The following program reworks Example 3 for the case when the amount of data is not known in advance.

```
Private Sub cmdShow_Click()
  Dim numStudents As Integer, nTemp As String, sTemp As Integer
  Dim student As Integer, total As Integer, average As Single
```

```
   'Determine amount of data to be processed
   numStudents = 0
   Open "SCORES.TXT" For Input As #1
   Do While Not EOF(1)
      Input #1, nTemp, sTemp
      numStudents = numStudents + 1
   Loop
   Close #1
   'Create arrays for names and scores
   ReDim nom(1 To numStudents) As String, score(1 To numStudents) As Integer
   Open "SCORES.TXT" For Input As #1
   For student = 1 To numStudents
      Input #1, nom(student), score(student)
   Next student Close #1
   'Analyze exam scores
   total = 0
   For student = 1 To numStudents
      total = total + score(student)
   Next student
   average = total / numStudents
   'Display all names with above-average grades
   picTopStudents.Cls
   For student = 1 To numStudents
      If score(student) > average Then
         picTopStudents.Print nom(student)
      End If
   Next student
End Sub
```

An alternative approach to program flexibility that does not require reading the data file twice is to require that the data file begin with a line that holds the number of records to be processed. If SCORES.TXT is modified by adding a new first line that gives the number of students, then the fourth through eighteenth lines of Example 4 can be replaced with

```
'Create arrays for names and scores
   Open "SCORES.TXT" For
   Input As #1
   Input #1, numStudents
   ReDim nom(1 To numStudents) As String, score(1 To numStudents) As Integer
   For student = 1 To numStudents
      Input #1, nom(student), score(student)
   Next student
Close #1
```

In Example 4, the ReDim statement allowed us to create arrays whose size was not known before the program was run. On the other hand, the arrays that were created were local to the event procedure cmdShow_Click. Many applications require form-level arrays whose size is not known in advance. Unfortunately, Dim statements cannot use variables or expressions to specify the subscript range. The solution offered by Visual Basic is to allow the (Declarations) section of (General) to contain Dim statements of the form

```
Dim arrayName() As varType
```

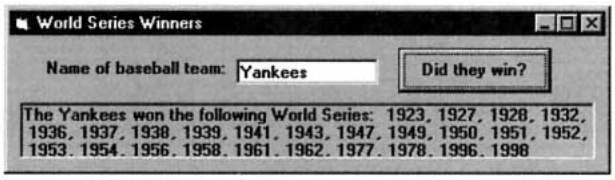
where no range for the subscripts of the array is specified. An array created in this manner will be form-level but cannot be used until a ReDim statement is executed in a procedure to establish the range of subscripts. The "As *varType*" clause can be omitted from the ReDim statement.

EXAMPLE 5

Suppose the data file WINNERS.TXT contains the names of the teams who have won each of the World Series, with the first line of the file giving the number of World Series that have been played. Write a program that displays the numbers, if any, of the World Series that were won by a team specified by the user.

```
'Create form-level array
   Dim teamName() As String
Dim seriesCount As Integer

Private Sub cmdDidTheyWin_Click()
   Dim teamToFind As String, numWon As Integer, series As Integer
   'Search for World Series won by user's team
   teamToFind = UCase(txtName.Text)
   numWon = 0
   picSeriesWon.Cls
   For series = 1 To seriesCount
      If UCase(teamName(series)) = teamToFind Then
         numWon = numWon + 1
         If numWon = 1 Then
            picSeriesWon.Print "The "; teamName(series);
            picSeriesWon.Print " won the following World Series: ";
         Else
            'Separate from previous
            picSeriesWon.Print ",";
            If (numWon = 5) Or (numWon = 16) Then
               'Start a new line at 5th and 16th win
               picSeriesWon.Print
            End If
         End If
         'First world series played in 1903
         picSeriesWon.Print Str(series + 1902);
      End If
   Next series
   If numWon = 0 Then
      picSeriesWon.Print "The "; teamToFind; " did not win any World Series."
   End If
End Sub

Private Sub Form_Load()
   Dim series As Integer
   'Fill array with World Series winners
   Open "WINNERS.TXT" For Input As #1
   Input #1, seriesCount
   ReDim teamName(1 To seriesCount) As String
   For series = 1 To seriesCount
      Input #1, teamName(series)
   Next series
   Close #1
End Sub
```

[Run, type Yankees into the text box, and click the command button.]



The range of an array need not just begin with 1. A statement of the form

```
Dim arrayName(m To n) As varType
```

where *m* is less than or equal to *n*, creates an array with elements *arrayName*(*m*), *arrayName*(*m* + 1), *arrayName*(*m* + 2), . . . , *arrayName*(*n*). The same holds for ReDim.

EXAMPLE 6

The following program segment stores the names of the 13th, 14th, and 15th Chief Justices of the U.S. Supreme Court in the array pictured in Figure 6-4.

```
'Place names of last three Chief Justices in an array
Dim chiefJustice(13 To 15) As String

   Private Sub Form_Load()
  chiefJustice(13) = "Earl Warren"
  chiefJustice(14) = "Warren Burger"
  chiefJustice(15) = "William Rehnquist"
End Sub
```

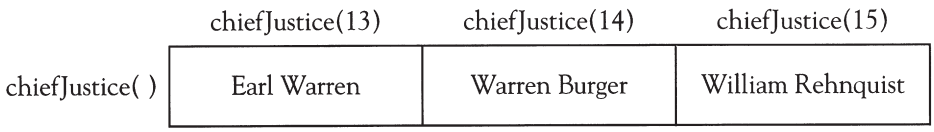


FIGURE 6-4 *The Array Created by Example 6*

An array can be used as either a checklist or frequency table, as in the next example. The function Asc associates each character with its position in the ANSI table.

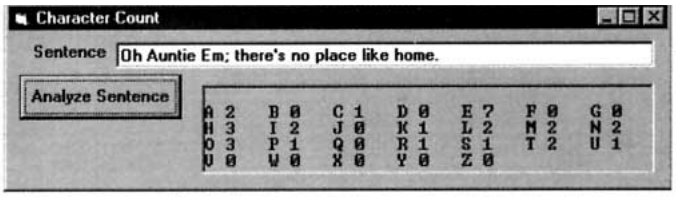
EXAMPLE 7

The following program requests a sentence as input and records the number of occurrences of each letter of the alphabet. The array *charCount*( ) has range Asc("A") To Asc("Z"), that is, 65 To 90. The number of occurrences of each letter is stored in the element whose subscript is the ANSI value of the uppercase letter.

```
Private Sub cmdAnalyze_Click()
  Dim index As Integer, letterNum As Integer, sentence As String
  Dim letter As String, column As Integer
  'Count occurrences of different letters in a sentence
  ReDim charCount(Asc("A") To Asc("Z")) As Integer
  For index = Asc("A") To Asc("Z")
    charCount(index) = 0
  Next index
  'Consider and tally each letter of sentence
  sentence = UCase(txtSentence.Text)
  For letterNum = 1 To Len(sentence)
    letter = Mid(sentence, letterNum, 1)
    If (letter >= "A") And (letter <= "Z") Then
```

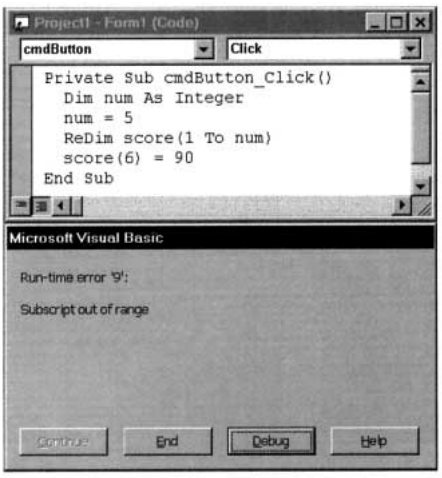
```
        index = Asc(letter)
        charCount(index) = charCount(index) + 1
      End If
    Next letterNum
    'List the tally for each letter of alphabet
    picLetterCount.Font = "Courier"
    picLetterCount.Cls
    column = 1 'Next column at which to display letter & count
      For letterNum = Asc("A") To Asc("Z")
      letter = Chr(letterNum)
      picLetterCount.Print Tab(column); letter;
      picLetterCount.Print Tab(column + 1); charCount(letterNum);
      column = column + 6
      If column > 42 Then 'only room for 7 sets of data in a line
        picLetterCount.Print
        column = 1
      End If
    Next letterNum
End Sub
```

[Run, type in the given sentence, and click the command button.]



COMMENTS

1. Arrays must be dimensioned in a Dim or ReDim statement before they are used. If a statement such as *a*(6) = 3 appears without a previous Dim or ReDim of the array *a*( ), then the error message "Sub or Function not defined" will be displayed when an attempt is made to run the program.
2. Subscripts in ReDim statements can be numeric expressions. Subscripts whose values are not whole numbers are rounded to the nearest whole number. Subscripts outside the range of the array produce an error message as shown below when the last line of the event procedure is reached.



3. The two arrays in Example 3 are referred to as **parallel arrays** because subscripted variables having the same subscript are related.
4. The integers *m* and *n* in the statement Dim *arrayName*(*m* To *n*) As *varType* can be positive, negative, or zero. The only restriction is that *m* cannot be greater than *n*. The same holds true for ReDim statements.
5. Until a value is assigned to an element of an array, the element has its default value. Numeric variables have a default value of 0, and string variables have the default value "", the empty string.
6. The statement Dim *arrayName*(0 To *n*) As *varType* can be replaced by the statement Dim *arrayName*(*n*) As *varType*. The same holds for the ReDim statement.
7. An array whose range has been specified by a ReDim statement is said to be **dynamic**. If *array1*() is a dynamic array, and *array2*() is another array of the same data type (such as String), then the line

   ```
   array1 = array2
   ```

   makes *array1*() an exact duplicate of *array2*(). It will have the same size and contain the same information. This feature was added in Visual Basic 6.0.
8. A dynamic array can be resized with another ReDim statement. However, the resized array loses all its information. If it is resized with the words **ReDim Preserve**, as much information as possible will be retained.

6.2 USING ARRAYS

This section considers three aspects of the use of arrays: processing ordered arrays, reading part of an array, and passing arrays to procedures.

■ ORDERED ARRAYS

An array is said to be **ordered** if its values are in either ascending or descending order. The following arrays illustrate the different types of ordered and unordered arrays. In an ascending ordered array, the value of each element is less than or equal to the value of the next element. That is,

[each element] ≤ [next element].

For string arrays, the ANSI table is used to evaluate the "less than or equal to" condition.

Ordered Ascending Numeric Array

dates( )	1492	1776	1812	1929	1969

Ordered Descending Numeric Array

discov( )	1610	1541	1513	1513	1492

Ordered Ascending String Array

king( )	Edward	Henry	James	John	Kong

Ordered Descending String Array

lake( )	Superior	Ontario	Michigan	Huron	Erie

Unordered Numeric Array

rates( )	8.25	5.00	7.85	8.00	6.50

Unordered String Array

char( )	G	R	E	A	T

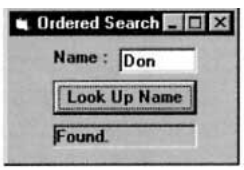
Ordered arrays can be searched more efficiently than unordered arrays. In this section we use their order to shorten the search. The technique used here is applied to searching sequential files in Section 7.

EXAMPLE 1

The following program places an ordered list of names into an array, requests a name as input, and informs the user if the name is in the list. Because the list is ordered, the search of the array ends when an element is reached whose value is greater than or equal to the input name. On average, only half the ordered array will be searched. Figure 6-5 shows the flowchart for this search.

```
'Create array to hold 10 strings
Dim nom(1 To 10) As String

Private Sub cmdSearch_Click()
   Dim n As Integer, name2Find As String
   'Search for a name in an ordered list
   name2Find = UCase(Trim(txtName.Text))
   n = 0 'n is the subscript of the array
   Do
      n = n + 1
   Loop Until (nom(n) >= name2Find) Or (n = 10)
   'Interpret result of search
   picResult.Cls
   If nom(n) = name2Find Then
        picResult.Print "Found."
      Else
        picResult.Print "Not found."
   End If
End Sub

Private Sub Form_Load()
   'Place the names into the array
   'All names must be in uppercase
   nom(1) = "AL"
   nom(2) = "BOB"
   nom(3) = "CARL"
   nom(4) = "DON"
   nom(5) = "ERIC"
   nom(6) = "FRED"
   nom(7) = "GREG"
   nom(8) = "HERB"
   nom(9) = "IRA"
   nom(10) = "JUDY"
End Sub
```

[Run, type Don into the text box, and click the command button.]



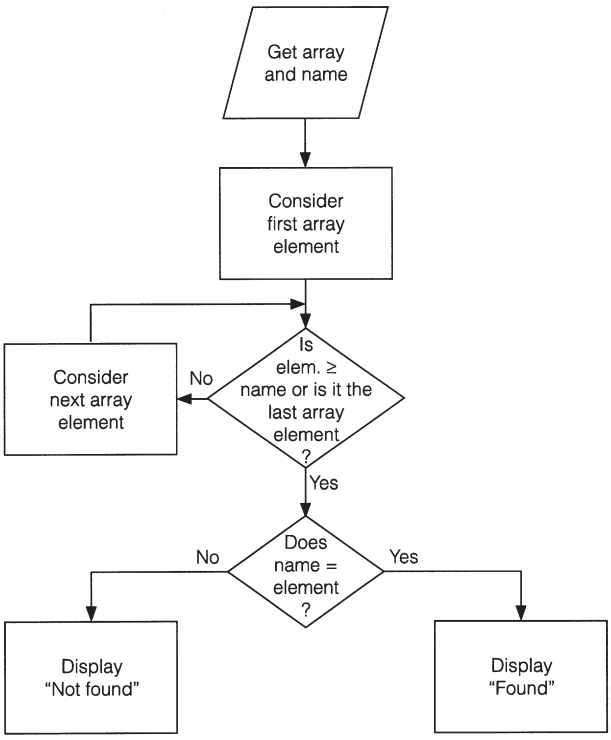


FIGURE 6-5 *Flowchart for a Search of an Ordered Array*

Using Part of an Array

In some programs, we must dimension an array before knowing how many pieces of data are to be placed into it. In these cases, we dimension the array large enough to handle all reasonable contingencies. For instance, if the array is to hold exam grades, and classes sizes vary from 5 to 100 students, we use a statement such as Dim grades(1 To 100) As Integer. In such situations we must employ a **counter variable** to keep track of the number of values actually stored in the array. We create this counter variable using a Dim statement in the (Declarations) section of (General) so that all procedures will have access to it.

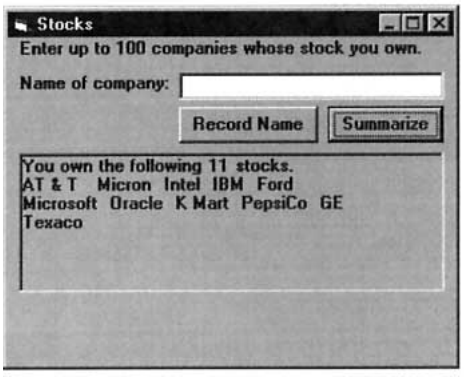
EXAMPLE 2

The following program requests a list of companies and then displays them along with a count.

```
'Demonstrate using only part of an array
   Dim stock(1 To 100) As String
   Dim counter As Integer

Private Sub cmdResult_Click()
   If (counter < 100) Then
      counter = counter + 1
      stock(counter) = txtCompany.Text
      txtCompany.Text = ""
      txtCompany.SetFocus
      Else
         MsgBox "No space to record additional companies.", , ""
         txtCompany.Text = ""
         cmdSummarize.SetFocus
   End If
End Sub
```

```
Private Sub cmdSummarize_Click()
   Dim i As Integer
   'List stock companies that have been recorded
   picStocks.Cls
   picStocks.Print "You own the following"; counter; "stocks."
   For i = 1 To counter
      picStocks.Print stock(i) & " ";
      'Move to new line after every 5 stocks
      If Int(i / 5) = i / 5 Then
         picStocks.Print
      End If
   Next i
End Sub

Private Sub Form_Load()
   'Initialize count of companies
   counter = 0
End Sub
```

[Run, type in eleven companies (press Record Name after each company) and press Summarize.]



Suppose you have two ordered lists of customers (possibly with some customers on both lists) and you want to consolidate them into a single ordered list. The technique for creating the third list, called the **merge algorithm**, is as follows.

1. Compare the two names at the top of the first and second lists.
   (a) If one name alphabetically precedes the other, copy it onto the third list and cross it off its original list.
   (b) If the names are the same, copy the name onto the third list and cross out the name from the first and second lists.
2. Repeat Step 1 with the current top names until you reach the end of either list.
3. Copy the names from the remaining list onto the third list.

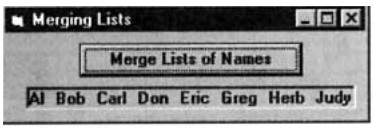
EXAMPLE 3

The following program stores two lists of names in arrays and merges them into a third list. Although at most 10 names will be placed into the third array, duplications will reduce this number. Because the variable *r* identifies the next position to insert a name in the third array, *r* – 1 is the number of names in the array.

```
'Create arrays to hold list of names
   Dim list1(1 To 5) As String, list2(1 To 5) As String
Dim newList(1 To 10) As String
```

```
Private Sub cmdMerge_Click()
   Dim m As Integer, n As Integer, r As Integer
   Dim numNames As Integer, i As Integer
   'Merge two lists of names
   m = 1 'Subscript for first array
   n = 1 'Subscript for second array
   r = 1 'Subscript and counter for third array
   Do While (m <= 5) And (n <= 5)
      Select Case list1(m)
         Case Is < list2(n)
            newList(r) = list1(m)
            m = m + 1
         Case Is < list2(n)
            newList(r) = list2(n)
            n = n + 1
         Case list2(n)
            newList(r) = list1(m)
            m = m + 1
            n = n + 1
      End Select
      r = r + 1
   Loop
   'If one of the lists has items left over, copy them into the third list
   'At most one of the following two loops will be executed
   Do While m <= 5 'Copy rest of first array into third
      newList(r) = list1(m)
      r = r + 1
      m = m + 1
   Loop
   Do While n <= 5 'Copy rest of second array into third
      newList(r) = list2(n)
      r = r + 1
      n = n + 1
   Loop
   numNames = r - 1
   'Show result of merging lists
   picMergedList.Cls
   For i = 1 To numNames
      picMergedList.Print newList(i) & " ";
   Next i
End Sub

Private Sub Form_Load()
   'Fill list1 with names
   list1(1) = "Al"
   list1(2) = "Carl"
   list1(3) = "Don"
   list1(4) = "Greg"
   list1(5) = "Judy"
   'Fill list2 with names
   list2(1) = "Bob"
   list2(2) = "Carl"
   list2(3) = "Eric"
   list2(4) = "Greg"
   list2(5) = "Herb"
End Sub
```

[Run and click the command button.]



Passing Arrays between Procedures

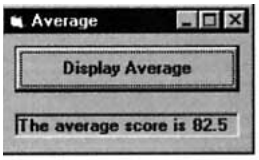
An array that is not dimensioned in the (Declarations) section of (General) but rather is declared in a procedure is local to that procedure and unknown in all other procedures. However, an entire local array can be passed to another procedure. The name of the array, followed by an empty set of parentheses, must appear as an argument in the calling statement, and an array variable name of the same type must appear as a corresponding parameter in the procedure definition of the procedure that is to receive the array.

EXAMPLE 4

The following program illustrates passing an array to both a Sub procedure and a Function procedure.

```
Private Sub cmdDisplayAverage_Click()
   'Pass array to Sub procedure and Function procedure
   Dim score(1 To 10) As Integer
   Call FillArray(score())
   picAverage.Cls
   picAverage.Print "The average score is"; Sum(score()) / 10
End Sub

Private Sub FillArray(s() As Integer)
   'Fill array with scores
   s(1) = 85
   s(2) = 92
   s(3) = 75
   s(4) = 68
   s(5) = 84
   s(6) = 86
   s(7) = 94
   s(8) = 74
   s(9) = 79
   s(10) = 88
End Sub]

Private Function Sum(s() As Integer) As Integer
   Dim total As Integer, index As Integer
   'Add up scores
   total = 0
   For index = 1 To 10
      total = total + s(index)
   Next index
   Sum = total
End Function
```

[Run and click the command button.]



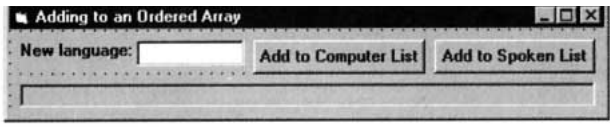
Sometimes it is also necessary to pass a form-level array from one procedure to another. For example, you might have a sorting procedure (discussed in Section 6.3) and three form-level arrays to be sorted. The sorting procedure would be called three times, each time passing a different form-level array. The method for passing a form-level array to another procedure is the same as the method for passing a local array.

EXAMPLE 5

The following program incorporates all three topics discussed in this section. It reads ordered lists of computer languages and spoken languages into form-level arrays, requests a new language as input, and inserts the language into its proper array position (avoiding duplication). The language arrays are dimensioned to hold up to 20 names; the variables *numCompLangs* and *numSpokLangs* record the actual number of languages in each of the ordered arrays. The original contents of the data files are

COMPLANG.TXT:ADA, C, Cobol, Fortran, Pascal, Visual Basic
SPOKLANG.TXT:Cantonese, English, French, Mandarin, Russian, Spanish

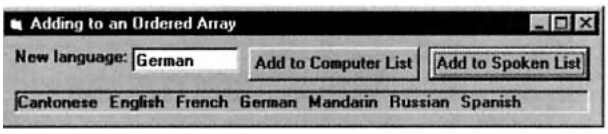
Object	Property	Setting
frmAdding	Caption	Adding to an Ordered Array
lblNew	Caption	New language:
txtLang	Text	(blank)
cmdAddComp	Caption	Add to Computer List
cmdAddSpok	Caption	Add to Spoken List
picAllLang		



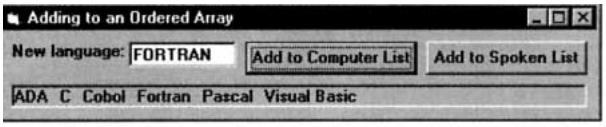

```
Dim compLang(1 To 20) As String
   Dim spokLang(1 To 20) As String
Dim numCompLangs As IntegerDim numSpokLangs As Integer

Private Sub AddALang(lang() As String, langCount As Integer)
   Dim language As String, n As Integer, i As Integer
   'Insert a language into an ordered array of languages
   language = Trim(txtLang.Text)
   n = 0
   Do
      n = n + 1
   Loop Until (UCase(lang(n))>= UCase(language)) Or (n = langCount)
   If UCase(lang(n)) < UCase(language) Then 'Insert new language at end
         lang(langCount + 1) = language
         langCount = langCount + 1
      ElseIf UCase(lang(n)) > UCase(language) Then 'Insert before item n
         For i = langCount To n Step -1
            lang(i + 1) = lang(i)
         Next i
         lang(n) = language
         langCount = langCount + 1
   End If
End Sub

Private Sub cmdAddComp_Click()
   'Insert language into ordered array of computer languages
   Call AddALang(compLang(), numCompLangs)
   Call DisplayArray(compLang(), numCompLangs)
End Sub

Private Sub cmdAddSpok_Click()
   'Insert language into ordered array of spoken languages
   Call AddALang(spokLang(), numSpokLangs)
```

```
   Call DisplayArray(spokLang(), numSpokLangs)
End Sub

Private Sub DisplayArray(lang() As String, howMany As Integer)
   Dim i As Integer
   'Display the languages in the array
   picAllLang.Cls
   For i = 1 To howMany
      picAllLang.Print lang(i) & " ";
   Next i
End Sub

Private Sub Form_Load()
   'Fill computer language array from COMPLANG.TXT
   numCompLangs = 0 Open "COMPLANG.TXT" For Input As #1
   Do While (Not EOF(1)) And (numCompLangs < 20)
      numCompLangs = numCompLangs + 1
      Input #1, compLang(numCompLangs)
   Loop
   Close #1
   'Fill spoken language array from SPOKLANG.TXT
   numSpokLangs = 0
   Open "SPOKLANG.TXT" For Input As #1
   Do While (Not EOF(1)) And (numSpokLangs < 20)
      numSpokLangs = numSpokLangs + 1
      Input #1, spokLang(numSpokLangs)
   Loop
   Close #1
End Sub
```

[Run, type in German, and click Add to Spoken List.]



[Type in FORTRAN and click Add to Computer List.]



COMMENTS

1. In Examples 1 and 5 we searched successive elements of an ordered list beginning with the first element. This is called a **sequential search**. An efficient alternative to the sequential search is the **binary search**, which is considered in the next section.
2. A single element of an array can be passed to a procedure just like any ordinary numeric or string variable.

```
Private Sub cmdButton_Click()
   Dim num(1 To 20) As Integer
```

```
  num(5) = 10
  picOutput.Print Triple(num(5))
End Sub

Private Function Triple(x As Integer) As Integer
  Triple = 3 * x
End Function
```

When the program is run and the command button clicked, 30 will be displayed.

3. Visual Basic provides two functions that simplify working with arrays that have been passed to a procedure. If an array has been dimensioned with the range *m* To *n*, then the values of the functions LBound(*arrayName*) and UBound(*arrayName*) are *m* and *n*, respectively.

```
Private Sub cmdButton_Click()
  Dim chiefJustice(13 To 15) As String
  chiefJustice(13) = "Warren"
  chiefJustice(14) = "Burger"
  chiefJustice(15) = "Rehnquist"
  Call Display(pres())
End Sub

Private Sub Display(a() As String)
  Dim i As Integer
  For i = LBound(a) To UBound(a)
    picOutput.Print a(i) & " ";
  Next i
End Sub
```

When the program is run and the command button clicked, "Warren Burger Rehnquist" will be displayed.

6.3 CONTROL ARRAYS

We have seen many examples of the usefulness of subscripted variables. They are essential for writing concise solutions to many programming problems. Because of the great utility that subscripts provide, Visual Basic also provides a means of constructing arrays of text boxes, labels, command buttons, and so on. Because text boxes, labels, and command buttons are referred to generically in Visual Basic as controls, arrays of these objects are called **control arrays**.

Unlike variable arrays, which can only be created by Dim and ReDim statements once a program is running, at least one element of a control array must be created when the form is designed. The remaining elements can be created either during form design, or, perhaps more typically, with the Load statement when the program is run.

To create the first element of an array of text boxes, create an ordinary text box, then access the Properties window, and select the property called Index. By default this property is blank. Change the Index property to 0 (zero). Your text box is now the first element in a subscripted control array. If the name of a text box is *txtBox* and its Index property is 0, then assigning a value to the text box during run time requires a statement of the form

```
txtBox(0).Text = value
```

Arrays are not of much use if they contain only a single element. To create additional elements of the *txtBox*( ) control array during form design, make sure that the element you just created is active by clicking on it. Next, press Ctrl+C (or open the Edit menu and select Copy). Visual Basic has now recorded all the properties associated with *txtBox*(0) and is

ready to reproduce as many copies as you desire. To create a copy, press Ctrl+V (or open the Edit menu and select Paste). The copy of *txtBox*(0) appears in the upper-left corner of the form. The value of the Index property for this new text box is 1; thus the text box is referred to as *txtBox*(1). Move this text box to the desired position. Press Ctrl+V again and another copy of *txtBox*(0) appears in the upper left corner of the form. Its Index property is 2. Move *txtBox*(2) to an appropriate position. Continue copying *txtBox*(0) in this manner until all desired controls have been created.

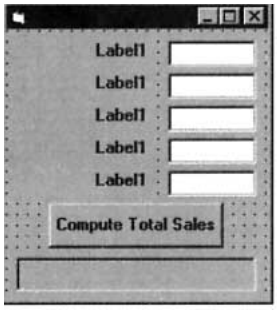
It is important to note that all property settings of *txtBox*(0) are being passed (as default settings) to the other elements of the *txtBox*( ) control array, with the exception of the Index, Top, and Left settings. Thus, as a matter of efficiency, before you begin copying *txtBox*(0), set all properties that you want carried over to all elements of *txtBox*( ). For example, if you desire to have the Text property blank for all *txtBox*( ) elements, set the Text property of *txtBox*(0) to (blank) before starting the copying process.

The preceding discussion gave a process for creating an array of text boxes. This same process applies to creating arrays of labels or any other control. In summary, the following steps create an array of controls while designing a form:

1. Add one instance of the desired control to the form.
2. Set the Index property of this control to 0.
3. Set any other properties of the control that will be common to all elements of the array.
4. Click on the control and then press Ctrl+C to prepare to make a copy of the control.
5. Press Ctrl+V to create a copy of the control. Position this control as desired.
6. Repeat Step 5 until all desired elements of the control array have been created.

EXAMPLE 1

A department store has five departments. The following program requests the amount of sales for each department and displays the total sales. We use a control array of five labels and a control array of five text boxes to handle the input. For the label captions we use "Department 1", "Department 2", and so on. Because these labels are the same except for the number, we wait until run time and use a For...Next loop inside the Form_Load ( ) event procedure to assign the captions to each element of the *lblDepart*( ) control array. At design time, before making copies of *lblDepart*(0), we set the Alignment property to "1 – Right Justify" so that all elements of the array inherit this property. Similarly, the Text property of *txtSales*(0) is set to (blank) before copying.

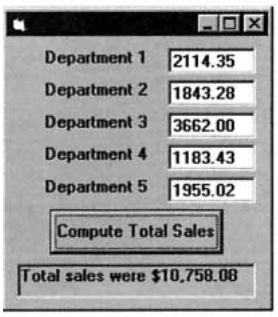


Object	Property	Setting
frm7_3_1	Caption	(blank)
lblDepart()	Index	0 to 4
	Alignment	1 – Right Justify
txtSales( )	Index	0 to 4
	Text	(blank)
cmdCompute	Caption	Compute Total Sales
picTotal		

```
Private Sub Form_Load()
   Dim depNum As Integer
   For depNum = 0 To 4
      lblDepart(depNum).Caption = "Department" & Str(depNum + 1)
   Next depNum
End Sub

Private Sub cmdCompute_Click()
   Dim depNum As Integer, sales As Single
```

```
   sales = 0
   For depNum = 0 To 4
      sales = sales + Val(txtSales(depNum).Text)
   Next depNum
   picTotal.Cls
   picTotal.Print "Total sales were "; FormatCurrency(sales)
End Sub
```

[Run, type the following data into the text boxes, and click the command button.]



Control Array Event Procedures

In Section 3 we discussed several events related to text boxes. One example was the GotFocus event procedure. If *txtBox* is an ordinary text box, then the GotFocus event procedure begins with the statement

```
Private Sub txtBox_GotFocus()
```

If, on the other hand, we make *txtBox* a control array, the GotFocus event procedure begins with the statement

```
Private Sub txtBox_GotFocus(Index As Integer)
```

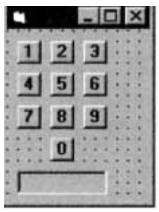
Two points should be noted. First, even though we may have a dozen or more elements in the *txtBox*( ) control array, we will have just one txtBox_GotFocus event procedure to deal with. Second, Visual Basic passes to this one event procedure the value of the Index property for the element of the control array that has just received the focus. We may wish to respond in the same manner whenever any element of *txtBox*( ) has the focus, in which case we simply ignore the value of *Index*. If, on the other hand, we wish to respond in different ways, depending on which element has the focus, then we write the GotFocus event procedure in the form

```
Private Sub txtBox_GotFocus(Index As Integer)
   Select Case Index
      Case 0
         action when txtBox( ) gets the focus
      Case 1
         action when txtBox(1) gets the focus . .
   End Select
End Sub
```

In general, all event procedures for a control array have the additional parameter Index As Integer. This additional parameter can be used, if desired, to base the action taken by the event procedure on which element of the control array underwent the event.

EXAMPLE 2

The following program creates an electronic dialing pad. The form contains a control array of 10 command buttons. Each time a command button is clicked, the Index parameter conveys the digit to be added onto the phone number. This program illustrates using the Index parameter without employing a Select Case statement.



Object	Property	Setting
frm7_3_2	Caption	(blank)
cmdDigit( )	Index	0 to 9
Caption	(same as Index)	
lblPhoneNum	BorderStyle	1 - Fixed Single
Caption	(blank)	

```
Private Sub cmdDigit_Click(Index As Integer)
   lblPhoneNum.Caption = lblPhoneNum.Caption & Right(str(index), 1)
   If Len(lblPhoneNum.Caption) = 3 Then
      lblPhoneNum.Caption = lblPhoneNum.Caption & "-"
   ElseIf Len(lblPhoneNum.Caption) = 8 Then
      MsgBox "Dialing ...", , ""
      lblPhoneNum.Caption = ""
   End If
End Sub
```

Creating Control Arrays at Run Time

We have discussed the process for creating an entire control array while designing a form, that is, at design time. However, copying and positioning control array elements can become tedious if the number of elements is large. Also, the actual number of elements needed in a control array may not be known until a response from the user is processed at run time. In light of these concerns, Visual Basic provides a solution via the Load statement that only requires us to create the first element of a control array during design time. The remaining elements are then created as needed at run time. Before we discuss creating arrays at run time, we must consider a preliminary topic—the Left, Top, Width, and Height properties of controls. These properties specify the location and size of controls.

The standard unit of measurement in Visual Basic is called a **twip**. There are about 1440 twips to the inch. At design time, when a control is active the two panels on the right side of the toolbar give the location and size of the control, respectively. Figure 6-6(a) shows an active text box, named Text1. The first panel says that the left side of the text box is 960 twips from the left side of the form, and the top of the text box is 720 twips down from the title bar of the form. In terms of properties, Text1.Left is 960, and Text1.Top is 720. Similarly, the numbers 1935 and 975 in the second panel give the width and height of the text box in twips. In terms of properties, Text1.Width is 1935 and Text1.Height is 975. Figure 6-6(b) shows the meanings of these four properties.

The location and size properties of a control can be altered at run time with statements such as

```
Text1.Left = 480
```

which moves the text box to the left or

```
Text2.Top = Text1.Top + 1.5 * Text1.Height
```

which places Text2 a comfortable distance below Text1. As a result of the second statement, the distance between the two text boxes will be half the height of Text1.

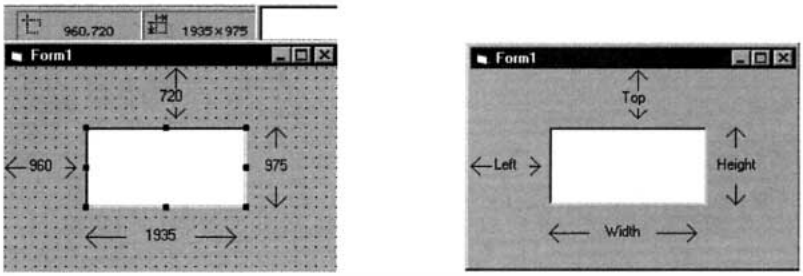


FIGURE 6-6 *The Location and Size of a Control*

If *controlName* is the name of a control whose Index property was assigned a value during form design (thus creating the beginnings of a control array) and *num* is a whole number that has not yet been used as an index for the *controlName*( ) array, then the statement

```
Load controlName(num)
```

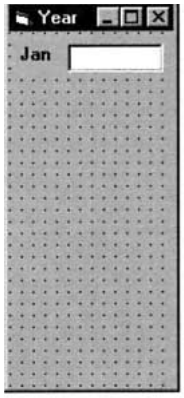
copies all the properties of *controlName*(0), including the Top and Left properties, and creates the element *controlName*(*num*) of the *controlName*( ) array. The only property of *controlName(num*) that may differ from that of *controlName*(0) is the Visible property. The Load statement always sets the Visible property of the created element to False. After creating a new element of a control array, you will want to adjust the Top and Left properties so that the new element has its own unique location on the form, and then set the Visible property of the new element to True.

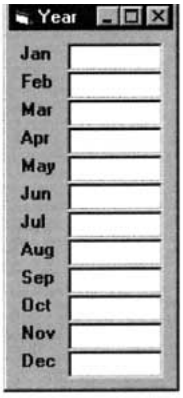
EXAMPLE 3

Write a program to create a control array of 12 labels and a control array of 12 text boxes. Position the labels and text boxes so that they form two columns, with the labels to the left of the text boxes and the text boxes one immediately below the other. Use text boxes whose height is as small as possible. Use labels whose height is just large enough to display a single line. Assign the captions Jan, Feb, and so on, to the labels.

SOLUTION:

When designing the form, we place the first label to the left of the first text box and set the Index property of both controls to 0. The height of the shortest text box is 288 units and the height of a label just tall enough for a single line is 252 units. We use the text box's Height property as the unit of vertical spacing for both the new text box elements and the new label elements. The following is the form at design time and at run time.





Object	Property	Setting
frm 7_3_3	Caption	Year
lblMonth()	Index	0
Caption	Jan	
Height	252	
txtInfo()	Index	0
Text	(blank)	
Height	288	

```
Private Sub Form_Load()
   Dim i As Integer, monthNames As String
   monthNames = "FebMarAprMayJunJulAugSepOctNovDec"
   For i = 1 To 11
      Load lblMonth(i)
      Load txtInfo(i)
      lblMonth(i).Top = lblMonth(i - 1).Top +
      txtInfo(0).Height txtInfo(i).Top = txtInfo(i - 1).Top + txtinfo(0).Height
      lblMonth(i).Caption = Mid(monthNames, 3 * i - 2, 3)
      lblMonth(i).Visible = True
      txtInfo(i).Visible = True
   Next i
End Sub
```

COMMENTS

1. In the discussion and examples of control arrays, the initial index was always 0. For a particular application it may be more natural to have the lowest index of a control array be 1 or even 1995. To achieve this when creating just the first element at design time and the remaining controls at run time, set the Index property of the first element to the desired lowest index value at design time, then Load the other elements using the desired indexes at run time. (The Load statement copies the properties of the element with the lowest index, whatever that lowest index may be.) For example, at design time you might create *txtSales*(1995) and then at run time execute the statements

```
For yearNum = 1996 to 2005
  Load txtSales(yearNum)
Next yearNum
```

   To create an entire control array at design time with indexes starting at a value other than 0, first create the control array using an initial index of 0. Once all elements have been created, use the Properties window to adjust the index of each element of the control array, starting with the element having the highest index.

6.4 SORTING AND SEARCHING

A **sort** is an algorithm for ordering an array. Of the many different techniques for sorting an array we discuss two, the **bubble sort** and the **Shell sort**. Both require the interchange of values stored in a pair of variables. If *var1*, *var2*, and *temp* are all variables of the same data type (such as all String), then the statements

```
temp = var1
    var1 = var2
var2 = temp
```

assign *var1*'s value to *var2*, and *var2*'s value to *var1*.

EXAMPLE 1

Write a program to alphabetize two words supplied in text boxes.

SOLUTION:

```
Private Sub cmdAlphabetize_Click()
   Dim firstWord As String, secondWord As String, temp As String
   'Alphabetize two words
   firstWord = txtFirstWord.Text
   secondWord = txtSecondWord.Text
   If firstWord > secondWord Then
      temp = firstWord
      firstWord = secondWord
      secondWord = temp
   End If
   picResult.Cls
   picResult.Print firstWord; " before "; secondWord
End Sub
```

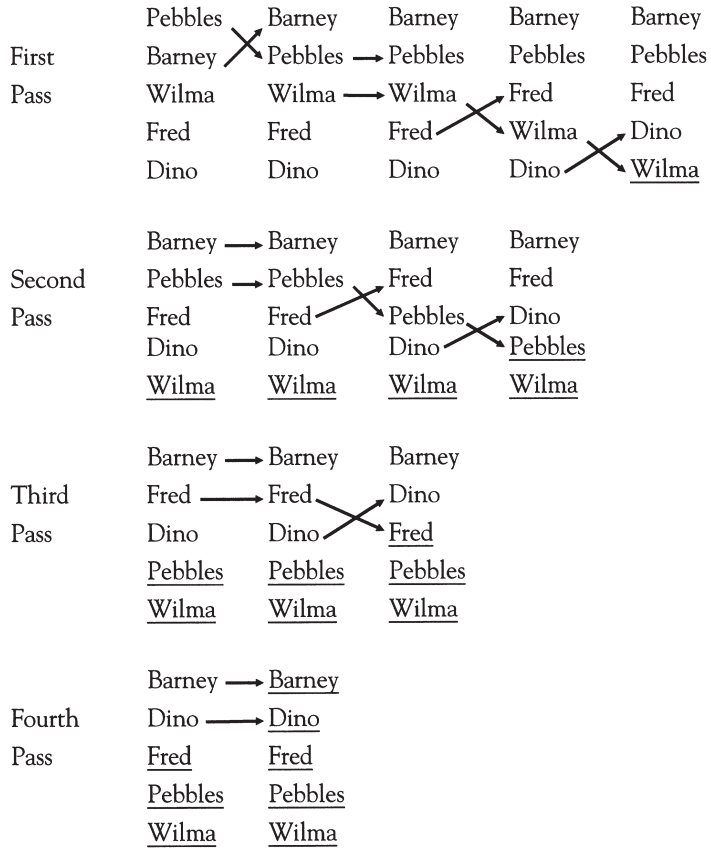
[Run, type the following text shown into the text boxes, and click the command button.]

Bubble Sort

The bubble sort is an algorithm that compares adjacent items and swaps those that are out of order. If this process is repeated enough times, the list will be ordered. Let's carry out this process on the list Pebbles, Barney, Wilma, Fred, Dino. The steps for each pass through the list are as follows:

1. Compare the first and second items. If they are out of order, swap them.
2. Compare the second and third items. If they are out of order, swap them.
3. Repeat this pattern for all remaining pairs. The final comparison and possible swap are between the second-to-last and last elements.

The first time through the list, this process is repeated to the end of the list. This is called the first pass. After the first pass, the last item (Wilma) will be in its proper position. Therefore, the second pass does not have to consider it and so requires one less comparison. At the end of the second pass, the last two items will be in their proper position. (The items that must have reached their proper position have been underlined.) Each successive pass requires one less comparison. After four passes, the last four items will be in their proper positions, and hence, the first will be also.



EXAMPLE 2

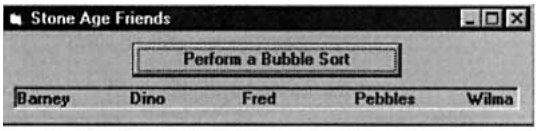
Write a program to alphabetize the names Pebbles, Barney, Wilma, Fred, Dino.

SOLUTION:

Sorting the list requires a pair of nested loops. The inner loop performs a single pass, and the outer loop controls the number of passes.

```
Dim nom(1 To 5) As String

Private Sub cmdSort_Click()
   Dim passNum As Integer, i As Integer, temp As String
   'Bubble sort names
   For passNum = 1 To 4 'Number of passes is 1 less than number of items
      For i = 1 To 5 - passNum 'Each pass needs 1 less comparison
         If nom(i) > nom(i + 1) Then
            temp = nom(i)
            nom(i) = nom(i + 1)
            nom(i + 1) = temp
         End If
      Next i
   Next passNum
   'Display alphabetized list
      picNames.Cls
   For i = 1 To 5
   picNames.Print nom(i),
   Next i
End Sub

Private Sub Form_Load()
   'Fill array with names
   nom(1) = "Pebbles"
   nom(2) = "Barney"
   nom(3) = "Wilma"
   nom(4) = "Fred"
   nom(5) = "Dino"
End Sub
```

[Run, and click the command button.]



EXAMPLE 3

Table 6.2 contains facts about the 10 most populous metropolitan areas with listings in ascending order by city name. Sort the table in descending order by population.

TABLE 6.2
The 10 Most Populous Metropolitan Areas

Metro Area	Population in Millions	Median Income per Household	% Native to State	% Advanced Degree
Boston	4.2	$40,666	73	12
Chicago	8.1	$35,918	73	8
Dallas	3.9	$32,825	64	8
Detroit	4.7	$34,729	76	7

Houston	3.7	$31,488	67	8
Los Angeles	14.5	$36,711	59	8
New York	18.1	$38,445	73	11
Philadelphia	5.9	$35,797	70	8
San Francisco	6.3	$41,459	60	11
Washington	3.9	$47,254	32	17

Note: Column 4 gives the percentage of residents who were born in their current state of residence. Column 5 gives the percentage of residents age 25 or older with a graduate or professional degree.

Source: The 1990 Census

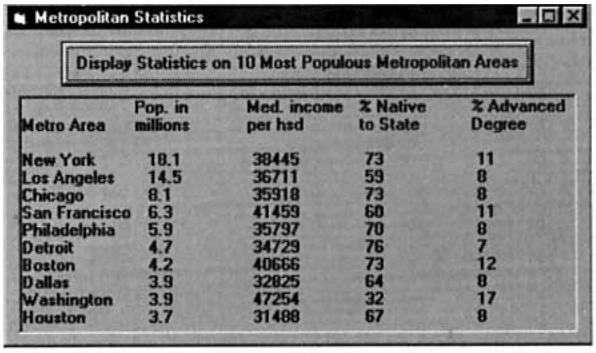
SOLUTION:

Data are read from a file into parallel arrays by the Form_Load event procedure. When cmdDisplayStats is clicked, the collection of parallel arrays is sorted based on the array *pop*( ). Each time two items are interchanged in the array *pop*( ), the corresponding items are interchanged in each of the other arrays. This way, for each city, the items of information remain linked by a common subscript.

```
Dim city(1 To 10) As String, pop(1 To 10) As Single, income(1 To 10) As Single
Dim natives(1 To 10) As Single, advDeg(1 To 10) As Single

Private Sub cmdDisplayStats_Click()
   Call SortData
   Call ShowData
   End Sub
   Private Sub Form_Load()
   Dim i As Integer
   'Assume the data for city name, population, medium income, % native,
   'and % advanced degree have been placed in the file "CITYSTAT.TXT"
   '(First line of file is "Boston", 4.2, 40666, 73, 12)
   Open "CITYSTAT.TXT" For Input As #1
   For i = 1 To 10
      Input #1, city(i), pop(i), income(i), natives(i), advDeg(i)
   Next i
   Close #1
End Sub

Private Sub ShowData()
   Dim i As Integer
   'Display ordered table
   picTable.Cls
   picTable.Print , "Pop. in", "Med. income", "% Native", "% Advanced"
   picTable.Print "Metro Area", "millions", "per hsd", "to State", "Degree"
   picTable.Print For i = 1 To 10
      picTable.Print city(i); Tab(16); pop(i), income(i), natives(i), advDeg(i)
   Next i
End Sub

Private Sub SortData()
   Dim passNum As Integer, index As Integer
   'Bubble sort table in descending order by population
   For passNum = 1 To 9
      For index = 1 To 10 - passNum
         If pop(index) < pop(index + 1) Then Call
            SwapData(index)
         End If
      Next index
   Next passNum
End Sub
```

```
Private Sub SwapData(index As Integer)
   'Swap entries
   Call SwapStr(city(index), city(index + 1))
   Call SwapNum(pop(index), pop(index + 1))
   Call SwapNum(income(index), income(index + 1))
   Call SwapNum(natives(index), natives(index + 1))
   Call SwapNum(advDeg(index), advDeg(index + 1))
End Sub

Private Sub SwapNum(a As Single, b As Single)
   Dim temp As Single
   'Interchange values of a and b
   temp = a
   a = b
   b = temp
End Sub

Private Sub SwapStr(a As String, b As String)
   Dim temp As String
   'Interchange values of a and b
   temp = a
   a = b
   b = temp
End Sub
```

[Run, and click the command button.]



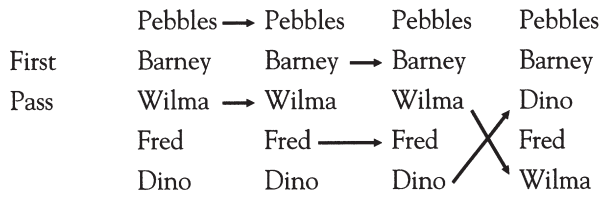
SHELL SORT

The bubble sort is easy to understand and program. However, it is too slow for really long lists. The Shell sort, named for its inventor, Donald L. Shell, is much more efficient in such cases. It compares distant items first and works its way down to nearby items. The interval separating the compared items is called the **gap**. The gap begins at one-half the length of the list and is successively halved until eventually each item is compared with its neighbor as in the bubble sort. The algorithm for a list of n items is as follows:

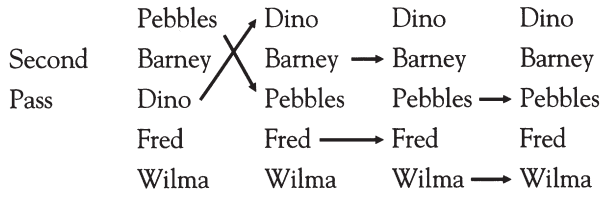
1. Begin with a gap of g = Int(n / 2).
2. Compare items 1 and 1 + g, 2 and 2 + g, . . . , $n - g$ and n. Swap any pairs that are out of order.
3. Repeat Step 2 until no swaps are made for gap g.
4. Halve the value of g.
5. Repeat Steps 2, 3, and 4 until the value of g is 0.

The Shell sort is illustrated in what follows. Crossing arrows indicate that a swap occurred.

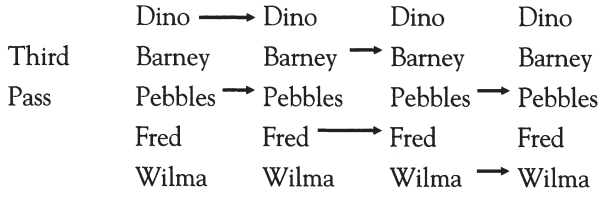
Initial Gap = Int([Number of Items] / 2) = Int(5 / 2) = 2



Because there was a swap, use the same gap for the second pass.

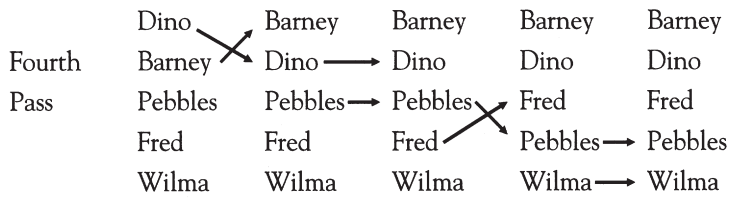


Again, there was a swap, so keep the current gap.

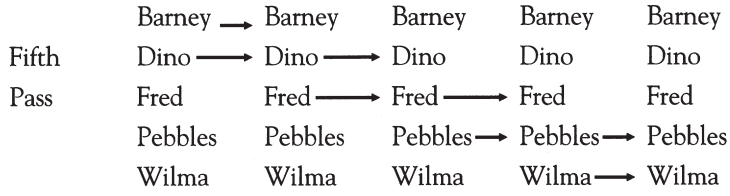


There were no swaps for the current gap of 2, so

Next Gap = Int([Previous Gap] / 2) = Int(2 / 2) = 1



Because there was a swap (actually two swaps), keep the same gap.



Because there were no swaps for the current gap, then

Next Gap = Int([Previous Gap] / 2) = Int(1 / 2) = 0

and the Shell sort is complete.

Notice that the Shell sort required 14 comparisons to sort the list whereas the bubble sort required only 10 comparisons for the same list. This illustrates the fact that for very short lists, the bubble sort is preferable; however, for lists of 30 items or more, the Shell sort will consistently outperform the bubble sort. Table 6.3 shows the average number of comparisons required to sort arrays of varying sizes.

TABLE 6.3
Efficiency of Bubble and Shell Sorts

Array Elements	Bubble Sort Comparisons	Shell Sort Comparisons
5	10	15
10	45	57
15	105	115
20	190	192
25	300	302
30	435	364
50	1225	926
100	4950	2638
500	124,750	22,517
1000	499,500	58,460

EXAMPLE 4

Use the Shell sort to alphabetize the parts of a running shoe (see Figure 6-7).

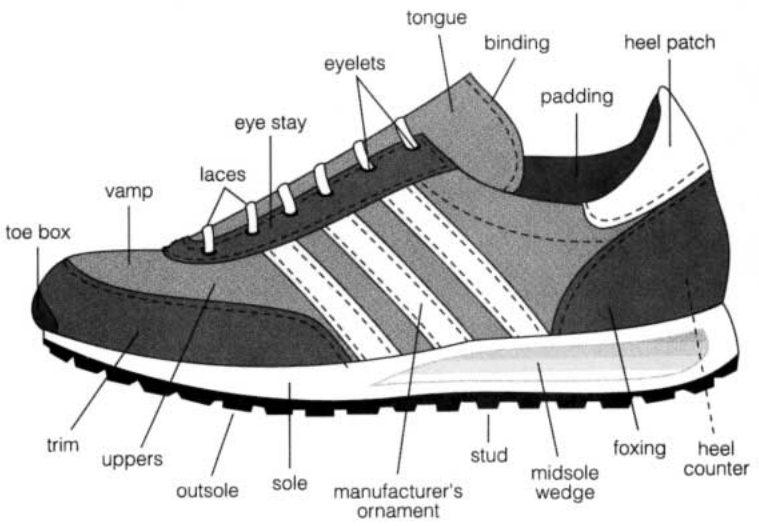


FIGURE 6-7 *Running Shoe*

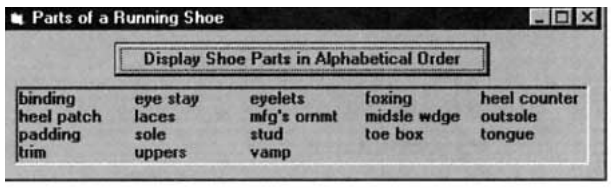
SOLUTION:

In the following program, the data are read into an array that has been dimensioned so as to guarantee more than enough space. In the event procedure Form_Load, the variable *numParts* provides the subscripts for the array and serves as a counter. The final value of *numParts* is available to all procedures because the variable was created in the (Declarations) section of (General). The Sub procedure SortData uses a flag to indicate if a swap has been made during a pass.

```
Dim part(1 To 50) As String
Dim numParts As Integer

Private Sub cmdDisplayParts_Click()
   'Sort and display parts of running shoe
   Call SortData
   Call ShowData
End Sub

Private Sub Form_Load()
   'Read part names numParts = 0 'Number of parts
```

```
   Open "SHOEPART.TXT"
   For Input As #1
   Do While (Not EOF(1)) And (numParts < UBound(part))
      numParts = numParts + 1
      Input #1, part(numParts)
   Loop
   Close #1
End Sub

Private Sub ShowData()
   Dim i As Integer 'Display sorted list of parts
   picParts.Cls
   For i = 1 To numParts
      picParts.Print part(i),
      If i Mod 5 = 0 Then 'only put 5 items per line
         picParts.Print
      End If
   Next i
End Sub

Private Sub SortData()
   Dim gap As Integer, doneFlag As Boolean
   Dim index As Integer, temp As String
   'Shell sort shoe parts
   gap = Int(numParts / 2)
   Do While gap >= 1
      Do
         doneFlag = True
         For index = 1 To numParts - gap
            If part(index) > part(index + gap)
               Then temp = part(index)
               part(index) = part(index + gap)
               part(index + gap) = temp
               doneFlag = False
            End If
         Next index
      Loop Until doneFlag = True 'Can also be written Loop Until doneFlag
      gap = Int(gap / 2) 'Halve the length of the gap
   Loop
End Sub
```

[Run and click the command button.]



SEARCHING

Suppose we had an array of 1000 names in alphabetical order and wanted to locate a specific person in the list. One approach would be to start with the first name and consider each name until a match was found. This process is called a **sequential search**. We would find a person

whose name begins with "A" rather quickly, but 1000 comparisons might be necessary to find a person whose name begins with "Z." For much longer lists, searching could be a time-consuming matter. However, when the list has already been sorted into either ascending or descending order, there is a method, called a **binary search**, that shortens the task considerably.

Let us refer to the sought item as *quarry*. The binary search looks for *quarry* by determining in which half of the list it lies. The other half is then discarded, and the retained half is temporarily regarded as the entire list. The process is repeated until the item is found. A flag can indicate if *quarry* has been found.

The algorithm for a binary search of an ascending list is as follows (Figure 6-8 shows the flowchart for a binary search):

1. At each stage, denote the subscript of the first item in the retained list by *first* and the subscript of the last item by *last*. Initially, the value of *first* is 1, the value of *last* is the number of items in the list, and the value of *flag* is False.
2. Look at the middle item of the current list, the item having the subscript *middle* = Int((*first* + *last*) / 2).
3. If the middle item is *quarry*, then *flag* is set to True and the search is over.
4. If the middle item is greater than *quarry*, then *quarry* should be in the first half of the list. So the subscript of *quarry* must lie between *first* and *middle* [minus] 1. That is, the new value of *last* is *middle* – 1.
5. If the middle item is less than *quarry*, then *quarry* should be in the second half of the list of possible items. So the subscript of *quarry* must lie between *middle* + 1 and *last*. That is, the new value of *first* is *middle* + 1.
6. Repeat Steps 2 through 5 until *quarry* is found or until the halving process uses up the entire list. (When the entire list has been used up, *first* > *last*.) In the second case, *quarry* was not in the original list.

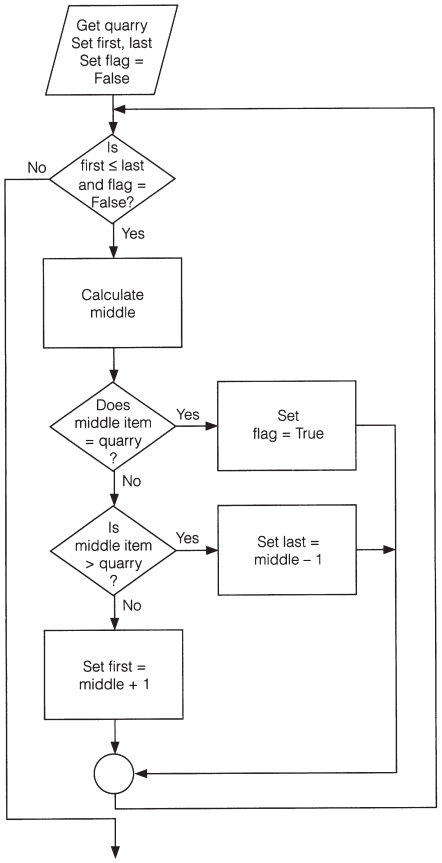


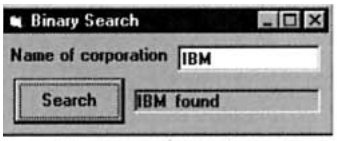
FIGURE 6-8 *Flowchart for a Binary Search*

EXAMPLE 5

In the following program the array *firm*( ) contains the alphabetized names of up to 100 corporations. The program requests the name of a corporation as input and uses a binary search to determine whether or not the corporation is in the array.

```
Dim firm(1 TO 100) As String
Dim numFirms As Integer

Private Sub BinarySearch(corp As String, result As String)
   Dim foundFlag As Boolean
   Dim first As Integer, middle As Integer, last As Integer
   'Array firm() assumed already ordered alphabetically
   'Binary search of firm() for corp
   foundFlag = False
   first = 1
   last = numFirms
   Do While (first <= last) And (Not foundFlag)
      middle = Int((first + last) / 2)
      Select Case UCase(firm(middle))
         Case corp
            foundFlag = True
         Case Is > corp
            last = middle - 1
         Case Is <
            corp first = middle + 1
      End Select
   Loop
   If foundFlag Then
         result = "found"
      Else
         result = "not found"
      End If
End Sub

Private Sub cmdSearch_Click()
   Dim corp As String, result As String
   corp = UCase(Trim(txtCorporation.Text))
   Call BinarySearch(corp, result)
   'Display results of search
   picResult.Cls
   picResult.Print corp; " "; result
End Sub

Private Sub Form_Load()
   'Fill array with data from FIRMS.TXT
   Open "FIRMS.TXT" For Input As #1 'Contains up to 100 companies
   numFirms = 0
   Do While (Not EOF(1)) And (numFirms < UBound(firm))
         numFirms = numFirms + 1
         Input #1, firm(numFirms)
   Loop Close #1
End Sub
```

Run, type IBM into the text box, and click the command button.]



Suppose the array contains 100 corporations and the corporation input in Example 5 is in the second half of the array. On the first pass, *middle* would be assigned Int((1 + 100)/2) = Int(50.5) = 50 and then *first* would be altered to 50 + 1 = 51. On the second pass, *middle* would be assigned Int((51 + 100)/2) = Int(75.5) = 75. If the corporation is not the array element with subscript 75, then either *last* would be assigned 74 or *first* would be assigned 76, depending on whether the corporation appears before or after the 75th element. Each pass through the loop halves the range of subscripts containing the corporation until the corporation is located.

In Example 5, the binary search merely reported whether or not an array contained a certain item. After finding the item, its array subscript was not needed. However, if related data are stored in parallel arrays (as in Table 6.2), the subscript of the found item can be used to retrieve the related information in the other arrays. This process, called a **table lookup**, is used in the following example.

EXAMPLE 6

Use a binary search procedure to locate the data for a city from Example 3 requested by the user.

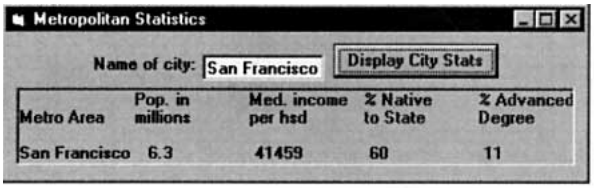
SOLUTION:

The following program does not include a sort of the data file CITYSTAT.TXT because the file is already ordered alphabetically.

```
Dim city(1 To 10) As String, pop(1 To 10) As Single, income(1 To 10) As Single
Dim natives(1 To 10) As Single, advDeg(1 To 10) As Single

Private Sub cmdDisplayStats_Click()
   Dim searchCity As String, result As Integer
   'Search for city in the metropolitan areas table
   Call GetCityName(searchCity)
   Call FindCity(searchCity, result)
   picResult.Cls
   If result > 0 Then
      Call ShowData(result)
   Else
      picResult.Print searchCity & " not in file"
   End If
End Sub

Private Sub FindCity(searchCity As String, result As Integer)
   Dim first As Integer, middle As Integer, last As Integer
   Dim foundFlag As Boolean
   'Binary search table for city name
   first = 1
   last = 10
   foundFlag = False
   Do While (first <= last) And (Not foundFlag)
      middle = Int((first + last) / 2)
      Select Case UCase(city(middle))
         Case searchCity
            foundFlag = True
         Case Is > searchCity
```

```
            last = middle - 1
         Case Is < searchCity
            first = middle + 1
      End Select
   Loop
      If foundFlag Then
         result = middle
      Else
         result = 0
   End If
End Sub

Private Sub Form_Load()
   Dim i As Integer
   'Assume that the data for city name, population, medium income, % native,
   'and % advanced degree have been placed in the file "CITYSTAT.TXT"
   '(First line of file is "Boston", 4.2, 4066, 73, 12)
   Open "CITYSTAT.TXT" For Input As #1
   For i = 1 To 10
      Input #1, city(i), pop(i), income(i), natives(i), advDeg(i)
   Next i
   Close #1
End Sub

Private Sub GetCityName(searchCity As String)
   'Request name of city as input
   searchCity = UCase(Trim(txtCity.Text))
End Sub

Private Sub ShowData(index As Integer)
   'Display city and associated information
   picResult.Print , "Pop. in", "Med. income", "% Native", "% Advanced"
   picResult.Print "Metro Area", "millions", "per hsd", "to State", "Degree"
   picResult.Print picResult.Print city(index), pop(index), income(index),
   picResult.Print natives(index), advDeg(index)
End Sub
```

[Run, type San Francisco into the text box, and click the command button.]



COMMENTS

1. Suppose our bubble sort algorithm is applied to an ordered list. The algorithm will still make $n - 1$ passes through the list. The process could be shortened for some lists by flagging the presence of out-of-order items as in the Shell sort. It may be preferable not to use a flag, because for greatly disordered lists the flag would slow down an already sluggish algorithm.

2. In Example 3, parallel arrays already ordered by one field were sorted by another field. Usually, parallel arrays are sorted by the field to be searched when accessing the file. This field is called the **key field**.
3. Suppose an array of 2000 items is searched sequentially—that is, one item after another—in order to locate a specific item. The number of comparisons would vary from 1 to 2000, with an average of 1000. With a binary search, the number of comparisons would be at most 11 because $2^{11} > 2000$.
4. The built-in function UCase converts all the characters in a string to uppercase. UCase is useful in sorting and searching arrays of strings when the alphabetic case (upper or lower) is unimportant. For instance, Example 5 includes UCase in the Select Case comparisons, and so the binary search will locate "Mobil" in the array even if the user entered "MOBIL".
5. The Visual Basic function Timer can be used to determine the speed of a sort. Precede the sort with the statement t = Timer. After the sort has executed, the statement picOutput.Print Timer – t will display the duration of the sort in seconds.

6.5 TWO-DIMENSIONAL ARRAYS

Each array discussed so far held a single list of items. Such array variables are called **single-subscripted variables**. An array can also hold the contents of a table with several rows and columns. Such arrays are called **two-dimensional arrays** or **double-subscripted variables**. Two tables follow. Table 6.4 gives the road mileage between certain cities. It has four rows and four columns. Table 6.5 shows the leading universities in three disciplines. It has three rows and five columns.

TABLE 6.4
Road Mileage Between Selected U.S. Cities

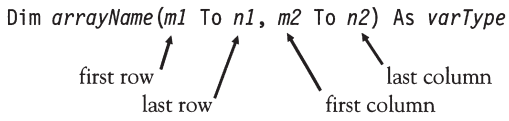
	Chicago	Los Angeles	New York	Philadelphia
Chicago	0	2054	802	738
Los Angeles	2054	0	2786	2706
New York	802	2786	0	100
Philadelphia	738	2706	100	0

TABLE 6.5
University Rankings

	1	2	3	4	5
Business	U of PA	U of IN	U of MI	UC Berk	U of VA
Comp Sci.	MIT	Cng-Mellon	UC Berk	Cornell	U of IL
Engr/Gen.	U of IL	U of OK	U of MD	Cng-Mellon	CO Sch. of Mines

Source: A Rating of Undergraduate Programs in American and International Universities, Dr. Jack Gourman, 1998

Two-dimensional array variables store the contents of tables. They have the same types of names as other array variables. The only difference is that they have two subscripts, each with its own range. The range of the first subscript is determined by the number of rows in the table, and the range of the second subscript is determined by the number of columns. The statement



dimensions an array of type *varType* corresponding to a table with rows labeled from *m1* To *n1* and columns labeled from *m2* To *n2*. The entry in the *j*th row, *k*th column is *arrayName* (*j*,*k*). For instance, the data in Table 6.4 can be stored in an array named rm( ). The statement

```
Dim rm(1 To 4, 1 To 4) As Single
```

will dimension the array. Each element of the array has the form *rm*(*row*, *column*). The entries of the array are

rm(1,1)=0	rm(1,2)=2054	rm(1,3)=802	rm(1,4)=738
rm(2,1)=2054	rm(2,2)=0	rm(2,3)=2786	rm(2,4)=2706
rm(3,1)=802	rm(3,2)=2786	rm(3,3)=0	rm(3,4)=100
rm(4,1)=738	rm(4,2)=2706	rm(4,3)=100	rm(4,4)=0

As with one-dimensional arrays, when a two-dimensional array is created using Dim in the (Declarations) section of (General), the array becomes a form-level subscripted variable, and is therefore accessible in all event procedures and general procedures and retains whatever values are assigned until the program is terminated. Two-dimensional arrays also can be created with Dim that are local to a procedure and cease to exist once the procedure is exited. When the range of the subscripts is given by one or more variables, the proper statement to use is

```
ReDim arrayName(m1 To n1, m2 To n2) As varType
```

The data in Table 7.10 can be stored in a two-dimensional string array named *univ*( ). The statement

```
Dim univ(1 To 3, 1 To 5) As String
```

will dimension the array as form-level. Some of the entries of the array are

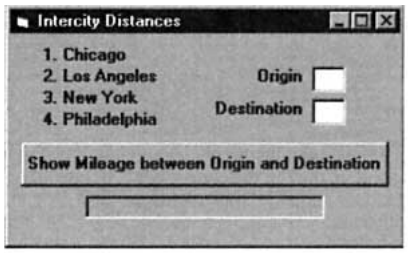
univ(1,1) = “U of PA”
univ(2,3) = “UC Berk”
univ(3,5) = “CO Sch. of Mines”

EXAMPLE 1

Write a program to store and access the data from Table 6.4.

SOLUTION:

Data are read from the data file DISTANCE.TXT into a two-dimensional form-level array using a pair of nested loops. The outer loop controls the rows and the inner loop controls the columns.



Object	Property	Setting
frmDist	Caption	Intercity Distances
lblCh	Caption	1. Chicago
lblLA	Caption	2. Los Angeles
lblNY	Caption	3. New York
lblPh	Caption	4. Philadelphia
lblOrig	Caption	Origin
txtOrig	Text	(blank)
lblDest	Caption	Destination
txtDest	Text	(blank)
cmdShow	Caption	Show Mileage between Origin and Destination
picMiles		

```
Dim rm(1 To 4, 1 To 4) As Single 'In (Declarations) section of (General)
   Private Sub cmdShow_Click()
   Dim row As Integer, col As Integer
   'Determine road mileage between cities
   row = Val(txtOrig.Text)
   col = Val(txtDest.Text)
```

```
   If (row>= 1 And row <= 4) And (col >= 1 And col <= 4)
      Then
         Call ShowMileage(rm(), row, col)
      Else
         MsgBox "Origin and Destination must be numbers from 1 to 4", , "Error"
   End If
   txtOrig.SetFocus
End Sub

Private Sub Form_Load()
   Dim row As Integer, col As Integer
   'Fill two-dimensional array with intercity mileages
   'Assume the data have been placed in the file "DISTANCE.TXT"
   '(First line of the file is 0, 54, 802, 738)
   Open "DISTANCE.TXT" For Input As #1
   For row = 1 To 4
      For col = 1 To 4
         Input #1, rm(row, col)
      Next col
   Next row
   Close #1
End Sub

Private Sub ShowMileage(rm() As Single, row As Integer, col As Integer)
   'Display mileage between cities
   picMiles.Cls
   picMiles.Print "The road mileage is"; rm(row, col)
End Sub
```

[Run, type 3 into the Origin box, type 1 into the Destination box, and click the command button.]

So far, two-dimensional arrays have been used only to store data for convenient lookup. In the next example, an array is used to make a valuable computation.

EXAMPLE 2

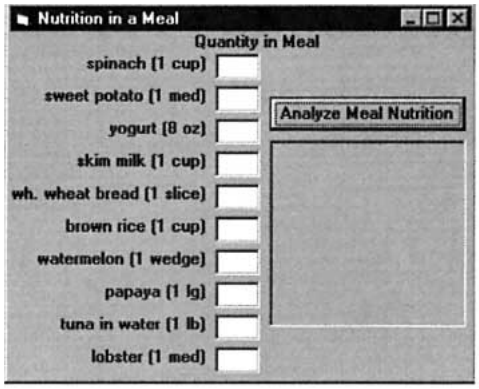
The Center for Science in the Public Interest publishes *The Nutrition Scorebook*, a highly respected rating of foods. The top two foods in each of five categories are shown in Table 6.6 along with some information on their composition. Write a program to compute the nutritional content of a meal. The table should be read into an array and then the program should request the quantities of each food item that is part of the meal. The program should then compute the amounts of each nutritional component consumed by summing each column with each entry weighted by the quantity of the food item.

TABLE 6.6
Composition of 10 Top-Rated Foods

	Calories	Protein (grams)	Fat (grams)	Vit A(IU)	Calcium (mg)
spinach (1 cup)	23	3	0.3	8100	93
sweet potato (1 med.)	160	2	1	9230	46
yogurt (8 oz.)	230	10	3	120	343
skim milk (1 cup)	85	8	0	500	302
whole wheat bread (1 slice)	65	3	1	0	24
brown rice (1 cup)	178	3.8	0.9	0	18
watermelon (1 wedge)	110	2	1	2510	30
papaya (1 lg.)	156	2.4	0.4	7000	80
tuna in water (1 lb)	575	126.8	3.6	0	73
lobster (1 med.)	405	28.8	26.6	984	190

SOLUTION:

Coding is simplified by using a control array of labels to hold the food names and a control array of text boxes to hold the amount input by the user. In the following template, the label captions have been assigned an initial value "(food name)" so that the labels can be seen. The five nutrients of interest and the actual names and nutrient values of the foods to be used in building a meal are read from the data file NUTTABLE.TXT.

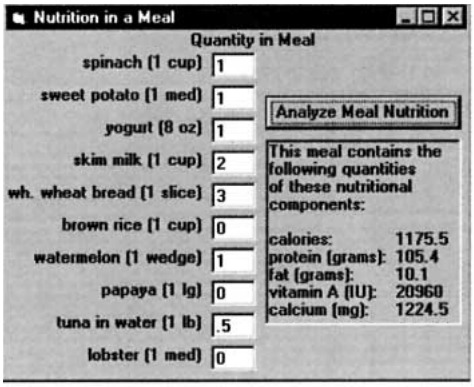


Object	Property	Setting
frmMeal	Caption	Nutrition in a Meal
lblFood()	Caption	(food name)
	Index	0 – 9
lblQnty	Caption	Quantity in Meal
txtQnty()	Text	(blank)
	Index	0 – 9
cmdAnalyze	Caption	Analyze Meal Nutrition
picAnalysis		

```
Dim nutName(1 To 5) As String 'nutrient names
Dim nutTable(1 To 10, 1 To 5) As Single 'nutrient values for each food

Private Sub cmdAnalyze_Click()
   'Determine the nutritional content of a meal
   Dim quantity(1 To 10) As Single 'amount of food in meal
   Call GetAmounts(quantity())
   Call ShowData(quantity())
End Sub

Private Sub Form_Load()
   Dim i As Integer, j As Integer, foodName As String
   'Fill arrays; assign label captions
   Open "NUTTABLE.TXT" For Input As #1
   For i = 1 To 5
      Input #1, nutName(i)
   Next i
   For i = 1 To 10
      Input #1, foodName
      lblFood(i - 1).Caption = foodName
      For j = 1 To 5
         Input #1, nutTable(i, j)
      Next j
   Next i
   Close #1
End Sub

Private Sub GetAmounts(quantity() As Single)
   Dim i As Integer
   'Obtain quantities of foods consumed
      For i = 1 To 10 quantity(i) = Val(txtQnty(i - 1).Text)
   Next i
End Sub

Private Sub ShowData(quantity() As Single)
   Dim col As Integer, row As Integer
```

```
    Dim amount As Single, nutWid As Single
    'Display amount of each component
    picAnalysis.Cls
    picAnalysis.Print "This meal contains the"
    picAnalysis.Print "following quantities"
    picAnalysis.Print "of these nutritional"
    picAnalysis.Print "components:"
    picAnalysis.Print For col = 1 To 5
       amount = 0
          For row = 1 To 10
             amount = amount + quantity(row) * nutTable(row, col)
          Next row
       picAnalysis.Print nutName(col) & ":"; Tab(16); amount
    Next col
End Sub
```

[Run, type the following quantities into each text box, and click the command button.]



COMMENT

1. We can define three- (or higher-) dimensional arrays much as we do two-dimensional arrays. A three-dimensional array uses three subscripts, and the assignment of values requires a triple-nested loop. As an example, a meteorologist might use a three-dimensional array to record temperatures for various dates, times, and elevations. The array might be created by the statement

```
Dim temps(1 To 31, 1 To 24, 0 To 14) As Single
```

6.6 A CASE STUDY: CALCULATING WITH A SPREADSHEET

Spreadsheets are one of the most popular types of software used on personal computers. A spreadsheet is a financial planning tool in which data are analyzed in a table of rows and columns. Some of the items are entered by the user. Other items, often totals and balances, are calculated using the entered data. The outstanding feature of electronic spreadsheets is their ability to recalculate an entire table after changes are made in some of the entered data, thereby allowing the user to determine the financial implications of various alternatives. This is called "What if?" analysis.

The Design of the Program

Figure 6-9 contains an example of a spreadsheet used to analyze a student's financial projections for the four quarters of a year. Column F holds the sum of the entries in columns B through E, rows 6 and 14 hold sums of the entries in rows 3 through 5 and 9 through 13, respectively, and row 16 holds the differences of the entries in rows 6 and 14. Because the total balance is negative, some of the amounts in the spreadsheet must be changed and the totals and balances recalculated.

	A	B	C	D	E	F
1		**Fall**	**Winter**	**Spring**	**Summer**	**Total**
2	**Income**					
3	Job	1000	1300	1000	2000	5300
4	Parents	200	200	200	0	600
5	Scholarship	150	150	150	0	450
6	**Total**	1350	1650	1350	2000	6350
7						
8	**Expenses**					
9	**Tuition**	400	0	400	0	800
10	**Food**	650	650	650	650	2600
11	**Rent**	600	600	600	400	2200
12	**Books**	110	0	120	0	230
13	**Misc**	230	210	300	120	860
14	Total	1990	1460	2070	1170	6690
15						
16	**Balance**	–640	190	–720	830	–340

FIGURE 6-9 *Spreadsheet for Student's Financial Projections*

The 96 locations in the spreadsheet that hold information are called **cells**. Each cell is identified by its row number and column letter. For instance, cell 14, C contains the amount 1460. For programming purposes, each column is identified by a number, starting with 1 for the leftmost column. Thus cell 14, C will be cell 14, 3 in our program.

This case study develops a program to produce a spreadsheet with the five columns of numbers shown in Figure 6-11, three user-specified categories of income, and five user-specified categories of expenses. The following tasks are to be selected by command buttons:

1. Start a new spreadsheet. All current category names and values are erased and the cursor is placed in the text box of the first income category.
2. Quit.

Three additional tasks need to be performed as the result of other events:

1. Create the spreadsheet when the form is loaded.
2. Limit the user to editing category names and quarterly values.
3. Display totals after a change is made in the spreadsheet.

The User Interface

Each cell in the spreadsheet will be an element of a text box control array. A control array of labels is needed for the numeric labels to the left of each row and another control array of labels for the alphabetic labels at the top of each column. Finally, two command buttons are required. The task of controlling which cells the user can edit will be handled by a GotFocus event. The task of updating the totals will be handled by a LostFocus event. Figure 6-10 shows one possible form design with all control elements loaded. For this application of a spreadsheet, the headings that have been assigned to cells in rows 1, 2, 6, 8, 14, and 16 are fixed; the user will not be allowed to edit them. The other entries in column A, the category names, may be edited by the user, but we have provided the set from Figure 6-9 as the default.

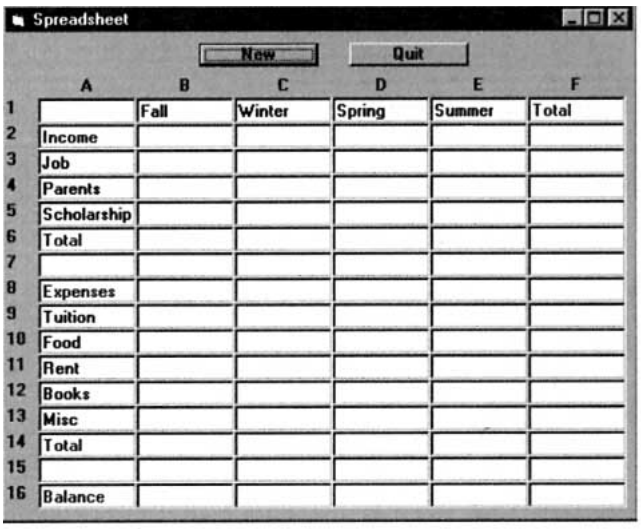


FIGURE 6-10 *Template for Spreadsheet*

Because processing the totals for the spreadsheet involves adding columns and rows of cells, coding is simplified by using a control array of text boxes so that an index in a For...Next loop can step through a set of cells. A two-dimensional array of text boxes seems natural for the spreadsheet. Unfortunately, only a single index is available for control arrays in Visual Basic. However, a single dimensional array of text boxes can be used without much difficulty if we define a function Indx that connects a pair of row (1 to 16) and column (1 to 6) values to a unique index (1 to 96) value. An example of such a rule would be Indx(row,column)=(row–1)*6+column. Successive values of this function are generated by going from left to right across row 1, then left to right across row 2, and so on.

A solution to the spreadsheet problem that uses one control array of text boxes and two control arrays of labels follows. The text box control array *txtCell*( ) provides the 96 text boxes needed for the spreadsheet cells. Because the proposed Indx function advances by one as we move from left to right across a row of cells, the cells must be positioned in this order as they are loaded. The label control array *lblRowLab*( ) provides a label for each of the rows of cells, while label control array *lblColLab*( ) provides a label for each column of cells. Figure 6-11 shows the layout of the form at design time. The properties for the controls are given in Table 6.7. The Height and Width properties given for the text box will assure enough room on the screen for all 96 cells. These dimensions can be obtained by creating a normal size text box, then reducing its width by one set of grid marks and its height by two sets of grid marks.

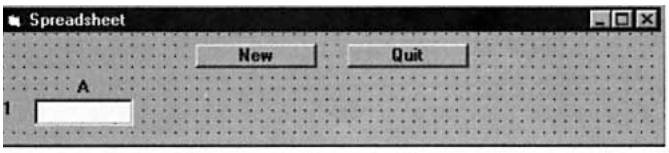


FIGURE 6-11 *Controls at Design Time*

TABLE 6.7
Objects and Their Properties

Object	Property	Setting
frmSpreadsheet	Caption	Spreadsheet
cmdNew	Caption	New
cmdQuit	Caption	Quit
lblRowLab()	Caption	1

Index	1	
lblColLab()	Caption	A
Index	1	
txtCell()	Text	(blank)
Index	1	
Height	1095	
Width	285	

Coding the Program

The top row of Figure 6-12 shows the different events to which the program must respond. Table 6.8 identifies the corresponding event procedures and the general procedures they call. Let's examine each event procedure.

1. **Form_Load** assigns the number of rows (16) and columns (6) in our spreadsheet to the form-level variables *maxRow* and *maxCol*. Form_Load then calls three general procedures to create and initialize the spreadsheet.

   The procedure CreateSpreadsheet loads each element of the *txtCell*( ) control array in order from left to right, top to bottom. Cell 1, which is to be the first cell in the first row, is not loaded because it was created at design time. The Top property of a new cell is set so that the top edge of the new cell overlaps the bottom edge of the previous cell in the column. The Top property of the first cell in a column is not modified, and so the value of the Top property is the same as cell 1. Similarly, the Left property of a new cell is set so that the left edge of the new cell overlaps the right edge of the previous cell in the row. The Left property of the first cell in a row is not modified, and so the value of the Left property is the same as cell 1. CreateSpreadsheet also loads the additional row and column label elements and assigns an appropriate caption. CreateSpreadsheet's final task is to set the Height and Width properties of frmSpreadSheet to accommodate all the objects that have been loaded. The numbers 500 and 200, which appear in these statements, were obtained by trial and error and are necessary to account for the space used by the form caption and borders.

   The procedure SetStructure assigns heading values to various cells of the spreadsheet in accordance with the specific application we were asked to program. The user will not be able to alter the value in these cells, because the rules for which cells are to be totaled, and where these totals are to be placed are "hard wired" into the program and cannot be changed by the user. SetStructure also assigns values to a set of form-level variables so that other procedures in the program can be coded using meaningful names rather than possibly obscure numbers. Besides Form_Load, SetStructure is also called by the cmdNew_Click event procedure.

   The procedure SetDefaults assigns the income and expense category headings shown in Figure 6-9 to the appropriate cells. The user may change these headings at any time, and must supply them if the "New" command is issued.

2. **txtCell_GotFocus** checks to see if the cell that has received the focus may, according to the rules of this application, be edited by the user. The row and column numbers for the cell are computed from the cell's index. If the cell that has received the focus is in a column after *stopCol*, the last editable column, then the column to be edited is changed to *startCol*, the first editable column, and the row to be edited is advanced by one. If the row to be edited does not contain any editable cells, then the row to be edited is advanced to the next row containing editable cells. (When the focus goes past the last row of editable cells, the next editable row is the first editable row, that is, *incStartRow*.) Finally, focus is set to the adjusted row and column, but only if an adjustment has been made. If the test Indx(row,col)<>Index were not made and focus were reset to a cell that already

had the focus, then the GotFocus event procedure would be invoked again as a result of the SetFocus, and then again as a result of the SetFocus performed by this invocation of GotFocus, and so on, resulting in an infinite loop.

3. **txtCell_LostFocus** invokes the general procedure DisplayTotals when the cursor leaves one of the spreadsheet cells. DisplayTotals in turn invokes five general procedures that each compute one set of needed totals and display the results by assigning values to appropriate text boxes. TotalIncome adds up the income for each quarter and saves the results in the array *iTot*( ). Similarly, TotalExpenses adds up the expenses for each quarter and saves the results in the array *eTot*( ). ComputeBalances takes the results stored in *iTot*( ) and *eTot*( ) and subtracts them to determine the balance for each quarter. TotalRows adds the four quarters for each category and assigns the results to the text boxes at the right end of each row. Finally, DetermineGrandTotals adds the values in *iTot*( ) and *eTot*( ) to determine the values for the right end of the "balance" row and each "total" row.
4. **cmdNew_Click** prepares for the entry of a new spreadsheet by setting the Text property of each element of the control array *txtCell*( ) to the null string and then setting focus to the first cell in the spreadsheet.
5. **cmdQuit_Click** ends the program.

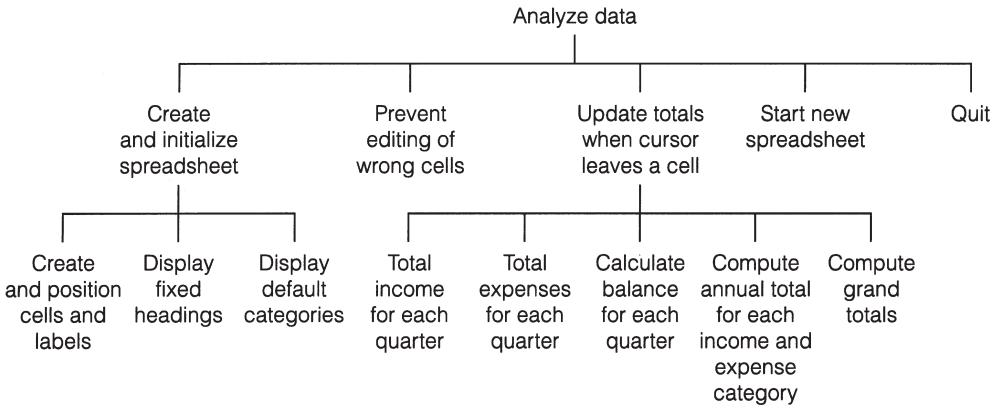


FIGURE 6.12 *Hierarchy Chart for Spreadsheet Program*

TABLE 6.8
Tasks and Their Procedures

1. Create & Initialize spreadsheet Form_Load
   1.1 Create & position cells & labels CreateSpreadsheet
   1.2 Display fixed headings SetStructure
   1.3 Display default categories SetDefaults
2. Prevent editing of wrong cells txtCell_GotFocus
3. Update totals when cursor leaves a cell txtCell_LostFocus (Display Totals)
   3.1 Total income each quarter TotalIncome
   3.2 Total expenses each quarter TotalExpenses
   3.3 Compute balances each quarter ShowBalances
   3.4 Total each income & expense category TotalRows
   3.5 Determine grand totals ShowGrandTotals
4. Start a new spreadsheet cmdNew_Click
5. End program cmdQuit_Click

```
Dim maxCol As Integer                ‘Number of columns in spreadsheet
    Dim maxRow As Integer            ‘Number of rows in spreadsheet
    Dim incStartRow As Integer       ‘Row where income categories begin
    Dim incStopRow As Integer        ‘Row where income categories end
    Dim incTotRow As Integer         ‘Row where income total is displayed
    Dim expStartRow As Integer       ‘Row where expense categories begin
    Dim expStopRow As Integer        ‘Row where expense categories end
    Dim expTotRow As Integer         ‘Row where expense total is displayed
    Dim balRow As Integer            ‘Row where balance is displayed
    Dim startCol As Integer          ‘Column where numeric data begins
    Dim stopCol As Integer           ‘Column where numeric data ends
    Dim totCol As Integer            ‘Column where total for each row is dis-
    played
    ’Control Arrays’txtCell()   Control array for data cells
    ’lblRowLab() Control array  for numeric row labels
’lblColLab() Control array      for alphabetic column labels

Private Sub cmdNew_Click()
   Dim row As Integer, col As Integer
   ‘Clear all data and total text boxes
   For col = 1 To maxCol
      For row = 1 To maxRow
         txtCell(Indx(row, col)).Text = “”
      Next row
   Next col
   Call SetStructure
   ‘Place cursor in first data txtCell
   txtCell(Indx(1, 1)).SetFocus
End Sub

Private Sub cmdQuit_Click()
   End
End Sub

Private Sub CreateSpreadsheet()
   Dim row As Integer, col As Integer, i As Integer
   Dim cellHeight As Single, cellWidth As Single
   Dim cellTop As Single, cellLeft As Single
   cellHeight = txtCell(1).Height
   cellWidth = txtCell(1).Width
   ‘Create cells For row = 1 To maxRow
      For col = 1 To maxCol
         i = Indx(row, col)
         If Not (col = 1 And row = 1) Then
            Load txtCell(i)
         End If
         If row > 1 Then
            cellTop = txtCell(Indx(row - 1, col)).Top
            txtCell(i).Top = cellTop + cellHeight
         End If
         If col > 1 Then
            cellLeft = txtCell(Indx(row, col - 1)).Left
            txtCell(i).Left = cellLeft + cellWidth
         End If
         txtCell(i).Visible = True
      Next col
```

```
    Next row
    'Create Row Labels
    For row = 2 To maxRow
      Load lblRowLab(row)
      lblRowLab(row).Top = lblRowLab(row - 1).Top + cellHeight
      lblRowLab(row).Caption = LTrim(Str(row))
      lblRowLab(row).Visible = True
    Next row
    'Create Column Labels
    For col = 2 To maxCol
      Load lblColLab(col)
      lblColLab(col).Left = lblColLab(col - 1).Left + cellWidth
      lblColLab(col).Caption = Chr(col + 64)
      lblColLab(col).Visible = True
    Next col 'Set form height and width to accommodate all objects
    i = Indx(maxRow, maxCol)
    frmSpreadsheet.Height = txtCell(i).Top + cellHeight + 500
    frmSpreadsheet.Width = txtCell(i).Left + cellWidth + 200
End Sub

Private Sub DisplayTotals()
    ReDim itot(startCol To stopCal) As Single
    ReDim etot(startCol To stopCal) As Single
    'Calculate and show totals for Income each quarter
    Call TotalIncome(itot())
    'Calculate and show totals for Expenses each quarter
    Call TotalExpenses(etot())
    'Calculate and show Balances for each quarter
    Call ShowBalances(itot(), etot())
    'Calculate and show the Total of each Income & Expense category
    Call TotalRows
    'Calculate and show grand totals of quarter totals and balances
    Call ShowGrandTotals(itot(), etot())
End Sub

    Private Sub Form_Load()
    'Establish number of rows and columns. Trial and error show
    'that a maximum of 20 rows and 8 columns will fit the screen.
    'For this particular application, 16 rows and 6 columns are adequate.
    maxRow = 16
    maxCol = 6
    Call CreateSpreadsheet
    Call SetStructure
    Call SetDefaults
End Sub

Private Function Indx(row As Integer, col As Integer) As Integer
    Indx = (row - 1) * maxCol + col
End Function

Private Sub SetDefaults()
    'Set default values specific to this application
    txtCell(Indx(3, 1)).Text = "Job"
    txtCell(Indx(4, 1)).Text = "Parents"
    txtCell(Indx(5, 1)).Text = "Scholarship"
    txtCell(Indx(9, 1)).Text = "Tuition"
```

```
   txtCell(Indx(10, 1)).Text = "Food"
   txtCell(Indx(11, 1)).Text = "Rent"
   txtCell(Indx(12, 1)).Text = "Books"
   txtCell(Indx(13, 1)).Text = "Misc"
End Sub

Private Sub SetStructure()
   txtCell(Indx(1, 2)).Text = "Fall"
   txtCell(Indx(1, 3)).Text = "Winter"
   txtCell(Indx(1, 4)).Text = "Spring"
   txtCell(Indx(1, 5)).Text = "Summer"
   txtCell(Indx(1, 6)).Text = "Total"
   txtCell(Indx(1, 6)).ForeColor = vbGreen
   txtCell(Indx(2, 1)).Text = "Income"
   txtCell(Indx(2, 1)).ForeColor = vbMagenta
   txtCell(Indx(6, 1)).Text = "Total"
   txtCell(Indx(6, 1)).ForeColor = vbGreen
   txtCell(Indx(8, 1)).Text = "Expenses"
   txtCell(Indx(8, 1)).ForeColor = vbMagenta
   txtCell(Indx(14, 1)).Text = "Total"
   txtCell(Indx(14, 1)).ForeColor = vbGreen
   txtCell(Indx(16, 1)).Text = "Balance"
   txtCell(Indx(16, 1)).ForeColor = vbGreen
   incStartRow = 3
   incStopRow = 5
   incTotRow = 6
   expStartRow = 9
   expStopRow = 13
   expTotRow = 14
   balRow = 16
   startCol = 2
   stopCol = 5
   totCol = 6
End Sub

Private Sub ShowBalances(itot() As Single, etot() As Single)
   Dim col As Integer
   For col = startCol To stopCol
      txtCell(Indx(balRow, col)).Text = FormatNumber(itot(col) - etot(col), 0)
   Next col
End Sub

Private Sub ShowGrandTotals(itot() As Single, etot() As Single)
   Dim col As Integer, iTotal As Single, eTotal As Single
   'Compute and display grand totals for income, expenses, and balance
   iTotal = 0
   eTotal = 0
   For col = startCol
   To stopCol
      iTotal = iTotal + itot(col)
      eTotal = eTotal + etot(col)
   Next col
   txtCell(Indx(incTotRow, totCol)) = FormatNumber(iTotal, 0)
   txtCell(Indx(expTotRow, totCol)) = FormatNumber(eTotal, 0)
   txtCell(Indx(balRow, totCol)) = FormatNumber(iTotal - eTotal, 0)
End Sub
```

```
Private Sub TotalExpenses(etot() As Single)
   Dim row As Integer, col As Integer
   'Total expenses for each of four quarters
   For col = startCol To stopCol
   etot(col) = 0
   For row = expStartRow To expStopRow
      etot(col) = etot(col) + Val(txtCell(Indx(row, col)).Text)
   Next row
   txtCell(Indx(expTotRow, col)).Text = FormatNumber(etot(col), 0)
   Next col
End Sub

Private Sub TotalIncome(itot() As Single)
   Dim row As Integer, col As Integer
   'Total income for each of four quarters
   For col = startCol To stopCol
   itot(col) = 0
   For row = incStartRow To incStopRow
      itot(col) = itot(col) + Val(txtCell(Indx(row, col)).Text)
   Next row
   txtCell(Indx(incTotRow, col)).Text = FormatNumber(itot(col), 0)
   Next col
End Sub

Private Sub TotalRows()
   Dim row As Integer, col As Integer, rowTot As Single
   'Total each income category
   For row = incStartRow To incStopRow
      rowTot = 0
      For col = startCol To stopCol
         rowTot = rowTot + Val(txtCell(Indx(row, col)).Text)
      Next col
      txtCell(Indx(row, totCol)).Text = FormatNumber(rowTot, 0)
   Next row
   'Total each expense category
   For row = expStartRow To expStopRow
      rowTot = 0
      For col = startCol To stopCal
         rowTot = rowTot + Val(txtCell(Indx(row, col)).Text)
      Next col
         txtCell(Indx(row, totCol)).Text = FormatNumber(rowTot, 0)
   Next row
End Sub

Private Sub txtCell_GotFocus(Index As Integer)
   Dim row As Integer, col As Integer
   'Force focus into a data txtCell for this application
   row = Int((Index - 1) / maxCol) + 1
   col = ((Index - 1) Mod maxCol) + 1
   If col > stopCol Then
        row = row + 1
        col = startCol
   End If
   If row < incStartRow Then
```

```
        row = incStartRow
      ElseIf (row > incStopRow) And (row < expStartRow) Then
        row = expStartRow
      ElseIf row > expStopRow Then
        row = incStartRow
    End If
    If Indx(row, col) ]<> Index Then
        txtCell(Indx(row, col)).SetFocus
    End If
End Sub

Private Sub txtCell_LostFocus(Indx As Integer)
    Call DisplayTotals
End Sub
```

SUMMARY

1. For programming purposes, tabular data are most efficiently processed if stored in an *array*. The *ranges* of variable arrays are specified by Dim or ReDim statements.
2. An array of labels, text boxes, or command buttons, referred to in general as a *control array*, can be created by assigning a value (usually zero) to the *Index* property of the control at design time. Additional elements of the control array are created either at design time by using Ctrl+C and Ctrl+V to copy the first element in the array or at run time by using the Load statement. New elements created in either way inherit all the properties of the first element except the Index, Visible (if created with Load), Top (when copied at design time), and Left (when copied at design time) properties.
3. Two of the best-known methods for ordering (or *sorting*) arrays are the *bubble sort* and the *Shell sort*.
4. Any array can be searched *sequentially* to find the subscript associated with a sought-after value. Ordered arrays can be searched most efficiently by a *binary search*.
5. A table can be effectively stored in a *two-dimensional array*.

PROGRAMMING PROJECTS

1. Table 6.9 contains some lengths in terms of feet. Write a program that displays the nine different units of measure, requests the unit to convert from, the unit to convert to, and the quantity to be converted, and then displays the converted quantity. A typical outcome is shown in Figure 6-13.

TABLE 6.9
Equivalent Lengths

1 inch = .0833 foot	1 rod = 16.5 feet
1 yard = 3 feet	1 furlong = 660 feet
1 meter = 3.2815 feet	1 kilometer = 3281.5 feet
1 fathom = 6 feet	1 mile = 5280 feet

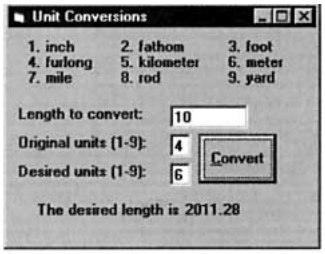


FIGURE 6-13 *Possible Outcome of Project 1*

2. Statisticians use the concepts of **mean** and **standard deviation** to describe a collection of data. The mean is the average value of the items, and the standard deviation measures the spread or dispersal of the numbers about the mean. Formally, if $x_1, x_2, x_3, \ldots, x_n$ is a collection of data, then

$$\text{mean} = m = \frac{x_1 + x_2 + x_3 + \ldots + x_n}{n}$$

$$\text{standard deviation} = s = \sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2 + \ldots + (x_n - m)^2}{n - 1}}$$

Write a computer program to

(a) Place the exam scores 59, 60, 65, 75, 56, 90, 66, 62, 98, 72, 95, 71, 63, 77, 65, 77, 65, 50, 85, and 62 into an array.

(b) Calculate the mean and standard deviation of the exam scores.

(c) Assign letter grades to each exam score, ES, as follows:

$ES \geq m + 1.5s$	A
$m + .5s \leq ES < m + 1.5s$	B
$m - .5s \leq ES < m + .5s$	C
$m - 1.5s \leq ES < m - .5s$	D
$ES < m - 1.5s$	F

For instance, if m were 70 and s were 12, then grades of 88 or above would receive A's, grades between 76 and 87 would receive B's, and so on. A process of this type is referred to as *curving grades*.

(d) Display a list of the exam scores along with their corresponding grades as shown in Figure 6-14.

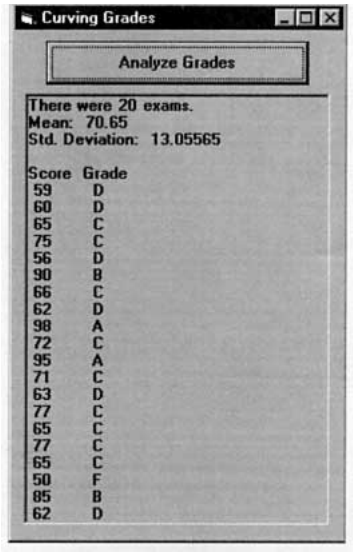


FIGURE 6-14 *Output of Project 2*

3. *Rudimentary Translator*. Table 6.10 gives English words and their French and German equivalents. Store these words in a data file and read them into three parallel arrays, one for each language. Write a program that sorts all three arrays according to the array of English words. The program should then request an English sentence as input from the keyboard and translate it into French and German. For example, if the English sentence given is MY PENCIL IS ON THE TABLE, then the French translation will be MON CRAYON EST SUR LA TABLE, and the German translation will be MEIN BLEISTIFT IST AUF DEM TISCH.

TABLE 6.10
English Words and Their French and German Equivalents

YES	OUI	JA	LARGE	GROS	GROSS
TABLE	TABLE	TISCH	NO	NON	NEIN
THE	LA	DEM	HAT	CHAPEAU	HUT
IS	EST	IST	PENCIL	CRAYON	BLEISTIFT
YELLOW	JAUNE	GELB	RED	ROUGE	ROT
FRIEND	AMI	FREUND	ON	SUR	AUF
SICK	MALADE	KRANK	AUTO	AUTO	AUTO
MY	MON	MEIN	OFTEN	SOUVENT, OFT	

4. Write a program that allows a list of no more than 50 soft drinks and their percent changes in market share for a particular year to be input and displays the information in two lists titled *gainers* and *losers*. Each list should be sorted by the *amount* of the percent change. Try your program on the data for the 8 soft drinks in Table 6.11. ***Note:*** You will need to store the data initially in an array to determine the number of gainers and losers.

TABLE 6.11
Changes in Market Share from 1996 to 1997 of Leading Soft-Drink Brands

Brand	% Change in Market Share	Brand	% Change in Market Share
Coke Classic	–.2	Sprite	.4
Pepsi-Cola	–.4	Dr. Pepper	.1
Diet Coke	–.2	Diet Pepsi	–.2
Mt. Dew	.5	7-Up	0

Source: Beverage Digest, 1998

5. Each team in a six-team soccer league played each other team once. Table 6.12 shows the winners. Write a program to

   (a) Place the team names in a one-dimensional array.

   (b) Place the data from Table 6.12 in a two-dimensional array.

   (c) Place the number of games won by each team in a one-dimensional array.

   (d) Display a listing of the teams giving each team's name and number of games won. The list should be in decreasing order by the number of wins.

TABLE 6.12
Soccer League Winners

	Jazz	Jets	Owls	Rams	Cubs	Zips
Jazz	—	Jazz	Jazz	Rams	Cubs	Jazz
Jets	Jazz	—	Jets	Jets	Cubs	Zips
Owls	Jazz	Jets	—	Rams	Owls	Owls
Rams	Rams	Jets	Rams	—	Rams	Rams
Cubs	Cubs	Cubs	Owls	Rams	—	Cubs
Zips	Jazz	Zips	Owls	Rams	Cubs	—

6. A poker hand can be stored in a two-dimensional array. The statement

```
Dim hand(1 TO 4, 1 TO 13) As Integer
```

declares a 52-element array, where the first dimension ranges over the four suits and the second dimension ranges over the thirteen denominations. A poker hand is specified by placing ones in the elements corresponding to the cards in the hand. See Figure 6-15.

Write a program that requests the five cards as input from the user, creates the related array, and passes the array to procedures to determine the type of the hand: flush (all cards have the same suit), straight (cards have consecutive denominations—ace can come either before 2 or after King), straight flush, four-of-a-kind, full house (3 cards of one denomination, 2 cards of another denomination), three-of-a-kind, two pairs, one pair, or none of the above.

	A	2	3	4	5	6	7	8	9	10	J	Q	K
Club	0	0	0	0	0	0	0	0	1	0	0	0	0
Diamond	1	0	0	0	0	0	0	0	0	0	0	0	0
Heart	1	0	0	0	0	0	0	0	0	0	0	1	0
Spade	0	0	0	0	1	0	0	0	0	0	0	0	0

FIGURE 6-15 *Array for the Poker Hand A♥ A♦ 5♠ 9♣ Q♥*

7. *Airline Reservations*. Write a reservation system for an airline flight. Assume the airplane has 10 rows with 4 seats in each row. Use a two-dimensional array of strings to maintain a seating chart. In addition, create an array to be used as a waiting list in case the plane is full. The waiting list should be "first come, first served," that is, people who are added early to the list get priority over those added later. Allow the user the following three options:

   (1) Add a passenger to the flight or waiting list.
      (a) Request the passenger's name.
      (b) Display a chart of the seats in the airplane in tabular form.
      (c) If seats are available, let the passenger choose a seat. Add the passenger to the seating chart.
      (d) If no seats are available, place the passenger on the waiting list.

   (2) Remove a passenger from the flight.
      (a) Request the passenger's name.
      (b) Search the seating chart for the passenger's name and delete it.
      (c) If the waiting list is empty, update the array so the seat is available.
      (d) If the waiting list is not empty, remove the first person from the list, and give him or her the newly vacated seat.

   (3) Quit.

8. The Game of Life was invented by John H. Conway to model some genetic laws for birth, death, and survival. Consider a checkerboard consisting of an *n*-by-*n* array of squares. Each square can contain one individual (denoted by 1) or be empty (denoted by –). Figure 6-16(a) shows a 6-by-6 board with four of the squares occupied. The future of each individual depends on the number of his neighbors. After each period of time, called a *generation*, certain individuals will survive, others will die due to either loneliness or overcrowding, and new individuals will be born. Each nonborder square has eight neighboring squares. After each generation, the status of the squares change as follows:

   (a) An individual *survives* if there are two or three individuals in neighboring squares.

(b) An individual *dies* if he has more than three individuals or less than two in neighboring squares.

(c) A new individual is *born* into each empty square with exactly three individuals as neighbors.

Figure 6-16(b) shows the status after one generation. Write a program to do the following:

(a) Dimension an *n*-by-*n* array, where *n* is input by the user, to hold the status of each square in the current generation. To specify the initial configuration, have the user input each row as a string of length *n*, and break the row into 1's or dashes with Mid.

(b) Dimension an *n*-by-*n* array to hold the status of each square in the next generation. Compute the status for each square and produce the display in Figure 6-16(b). ***Note:*** The generation changes all at once. Only current cells are used to determine which cells will contain individuals in the next generation.

(c) Assign the next-generation values to the current generation and repeat as often as desired.

(d) Display the number of individuals in each generation.

Hint: The hardest part of the program is determining the number of neighbors a cell has. In general, you must check a 3-by-3 square around the cell in question. Exceptions must be made when the cell is on the edge of the array. Don't forget that a cell is not a neighbor of itself.

(Test the program with the initial configuration shown in Figure 6-17. It is known as the figure-eight configuration and repeats after eight generations.)

```
- - - - - -   - - - - - -
- - - - - -   - - 1 1 - -
- 1 1 1 1 -   - - 1 1 - -
- - - - - -   - - 1 1 - -
- - - - - -   - - - - - -
- - - - - -   - - - - - -
```

FIGURE 6-16 *Two Generations*

```
- - - - - - - - - - -
- - - - - - - - - - -
- - 1 1 1 - - - - - -
- - 1 1 1 - - - - - -
- - 1 1 1 - - - - - -
- - - - - 1 1 1 - - -
- - - - - 1 1 1 - - -
- - - - - 1 1 1 - - -
- - - - - - - - - - -
- - - - - - - - - - -
```

FIGURE 6-17 *The Figure Eight*

9. Simulate the game Concentration. The Form_Load routine should create an array of 20 command buttons placed vertically on a form. A list of 10 words should be randomly assigned as potential captions for the buttons, with each word assigned to two command buttons. Initially, none of the buttons should show their words. However, when a button is clicked on, its word is revealed as its caption. After two words have been revealed, either both of the command buttons should become invisible (if their words match) or their captions should again become blank (if the two words do not match). When all matches have been found, a message box should display the number of tries (pairs of words exposed) and an appropriate remark about the user's concentration ability. Possible remarks might be, "You must have ESP" (less than 14 tries), "Amazing concentration" (14 to 20 tries), "Can't hide anything from you" (21 to 28 tries), "Perhaps a nap would recharge your concentration" (29 to 37 tries), and "Better find a designated driver" (more than 37 tries).

SECTION 7

SEQUENTIAL FILES

7.1 SEQUENTIAL FILES

Throughout this text we have processed data from files created with Windows' Notepad and saved on a disk. Such files are stored on disk as a sequence of characters. (Two special characters, called the "carriage return" and "line feed" characters, are inserted at the end of each line to indicate where new lines should be started.) Such files are called **sequential files** or **text files**. In this section, we create sequential files from Visual Basic programs and develop techniques for using sequential files.

■ CREATING A SEQUENTIAL FILE

There are many ways to organize data in a sequential file. The technique presented here is easy to implement. The following steps create a new sequential file and write data to it.

1. Choose a file name. A file name can contain up to 255 characters consisting of letters, digits, and a few other assorted characters (including spaces and periods). In this book we use 8.3 format names; that is, each name has a base name of at most 8 characters, and optionally a period followed by a three-letter extension. (Such names are recognized by all utility programs.)
2. Choose a number from 1 through 511 to be the **reference number** of the file. While the file is in use, it will be identified by this number.
3. Execute the statement

   ```
   Open "filespec" For Output As #n
   ```

   where *n* is the reference number. This process is referred to as **opening a file for output**. It establishes a communications link between the computer and the disk drive for storing data *onto* the disk. It allows data to be output from the computer and recorded in the specified file.
4. Place data into the file with the Write # statement. If *a* is a string, then the statement

   ```
   Write #n, a
   ```

   writes the string *a* surrounded by quotation marks into the file. If *c* is a number, then the statement

   ```
   Write #n, c
   ```

writes the number *c*, without any leading or trailing spaces, into file number *n*. The statement

```
Write #n, a, c
```

writes *a* and *c* as before, but with a comma separating them. Similarly, if the statement Write #*n* is followed by a list of several strings and/or numbers separated by commas, then all the strings and numbers appear as before, separated by commas. After each Write # statement is executed, the "carriage return" and "line feed" characters are placed into the file.

5. After all the data have been recorded in the file, execute

```
Close #n
```

where *n* is the reference number. This statement breaks the communications link with the file and dissociates the number *n* from the file.

EXAMPLE 1

The following program illustrates the different aspects of the Write # statement. Notice the absence of leading and trailing spaces for numbers and the presence of quotation marks surrounding strings.

```
Private Sub cmdCreateFile_Click()
   Dim name1 As String, name2 As String
   'Demonstrate use of Write # statement
   Open "PIONEER.TXT" For Output As #1
   Write #1, "ENIAC" Write #1, 1946
   Write #1, "ENIAC", 1946 name1 = "Eckert" name2 = "Mauchly"
   Write #1, 14 * 139, "J.P. " & name1, name2, "John"
   Close #1
End Sub
```

[Run, click the command button, and then load the file PIONEER.TXT into Windows' Notepad. The following will appear on the screen.]

```
"ENIAC"
1946
"ENIAC",1946
1946,"J.P. Eckert","Mauchly","John"
```

Caution: If an existing sequential file is opened for output, the computer will erase the existing data and create a new empty file.

Write # statements allow us to create files just like the Notepad files that appear throughout this text. We already know how to read such files with Input # statements. The remaining major task is adding data to the end of sequential files.

■ Adding Items to a Sequential File

Data can be added to the end of an existing sequential file with the following steps.

1. Choose a number from 1 through 511 to be the reference number for the file. It need not be the number that was used when the file was created.
2. Execute the statement

```
Open "filespec" For Append As #n
```

where *n* is the reference number. This procedure is called **opening a file for append**. It allows data to be output and recorded at the end of the specified file.

3. Place data into the file with Write # statements.

4. After all the data have been recorded into the file, close the file with the statement Close #*n*.

The Append option for opening a file is intended to add data to an existing file. However, it also can be used to create a new file. If the file does not exist, then the Append option acts just like the Output option and creates the file.

The three options, Output, Input, and Append, are referred to as **modes**. A file should not be open in two modes at the same time. For instance, after a file has been opened for output and data have been written to the file, the file should be closed before being opened for input.

An attempt to open a nonexistent file for input terminates the program with the "File not found" error message. There is a function that tells us whether a certain file has already been created. If the value of

```
Dir("filespec")
```

is the empty string "", then the specified file does not exist. (If the file exists, the value will be the file name.) Therefore, prudence often dictates that files be opened for input with code such as

```
If Dir("filespec") <> "" Then
     Open "filespec" For Input As #1
   Else
     message = "Either no file has yet been created or "
     message = message & "the file is not where expected."
     MsgBox message, , "File Not Found"
End If
```

There is one file-management operation that we have yet to discuss—deleting an item of information from a file. An individual item of a file cannot be changed or deleted directly. A new file must be created by reading each item from the original file and recording it, with the single item changed or deleted, into the new file. The old file is then erased and the new file renamed with the name of the original file. Regarding these last two tasks, the Visual Basic statement

```
Kill "filespec"
```

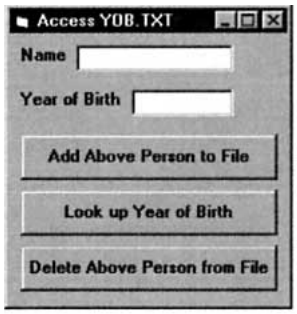
removes the specified file from the disk and the statement

```
Name "oldfilespec" As "newfilespec"
```

changes the filespec of a file. (***Note:*** The Kill and Name statements cannot be used with open files. So doing generates a "File already open" message.)

EXAMPLE 2

The following program creates and manages a file of names and years of birth.



Object	Property	Setting
frm8_1_2	Caption	Access YOB.TXT
lblName	Caption	Name
txtName	Text	(blank)
lblYOB	Caption	Year of Birth
txtYOB	Text	(blank)
cmdAdd	Caption	Add Above Person to File
cmdLookUp	Caption	Look up Year of Birth
cmdDelete	Caption	Delete Above Person from File

```
Private Sub cmdAdd_Click()
  Dim message As String
```

```
   'Add a person's name and year of birth to file
   If (txtName.Text <> "") And (txtYOB.Text "") Then
        Open "YOB.TXT" For Append As #1
        Write #1, txtName.Text, Val(txtYOB.Text)
        Close #1
        txtName.Text = ""
        txtYOB.Text = ""
        txtName.SetFocus
     Else
        message = "You must enter a name and year of birth."
        MsgBox message, , "Information Incomplete"
   End If
End Sub

Private Sub cmdLookUp_Click()
   Dim message As String
   'Determine a person's year of birth
   If txtName.Text <> "" Then
     If Dir("YOB.TXT") <> "" Then
        Call DisplayYearOfBirth
          Else
             message = "Either no file has yet been created or "
             message = message & "the file is not where expected."
             MsgBox message, , "File Not Found"
        End If
     Else
        MsgBox "You must enter a name.", , "Information Incomplete"
   End If
   txtName.SetFocus
End Sub

Private Sub cmdDelete_Click()
   Dim message As String
   'Remove a person from the file
   If txtName.Text <> "" Then
     If Dir("YOB.TXT") <> "" Then
        Call DeletePerson
          Else
             message = "Either no file has yet been created or "
             message = message & "the file is not where expected."
             MsgBox message, , "File Not Found."
        End If
     Else
        MsgBox "You must enter a name.", , "Information Incomplete"
   End If
   txtName.SetFocus
End Sub

Private Sub DeletePerson()
   Dim nom As String, yob As Integer, foundFlag As Boolean
   foundFlag = False Open "YOB.TXT" For Input As #1
   Open "TEMP" For Output As #2
   Do While Not EOF(1)
     Input #1, nom, yob
     If nom <> txtName.Text Then
```

```
            Write #2, nom, yob
         Else
            foundFlag = True
      End If
   Loop
   Close #1
   Close #2
   Kill "YOB.TXT"
   Name "TEMP" As "YOB.TXT"
   If Not foundFlag Then
        MsgBox "The name was not found.", , ""
      Else
        txtName.Text = ""
        txtYOB.Text = ""
   End If
End Sub

Private Sub DisplayYearOfBirth()
   Dim nom As String, yob As Integer
   'Find the year of birth for the name in txtName
   txtYOB.Text = ""
   Open "YOB.TXT" For Input As #1
   nom = ""
   Do While (nom <> txtName.Text) And (Not EOF(1))
      Input #1, nom, yob
   Loop
   If nom = txtName.Text Then
        txtYOB.Text = Str(yob)
      Else
        MsgBox "Person is not in file.", , ""
        txtName.Text = ""
   End If
   Close #1
End Sub
```

[Run. After several names have been added, the file might look as shown in Figure 7-1.]

```
"Barbra",1942
"Ringo",1940
"Sylvester",1946
```

FIGURE 7-1 *Sample Contents of YOB.TXT*

■ ERROR TRAPPING

If you try to Open a file on a diskette in drive A and there is no diskette in drive A, the program will crash with the error message "Disk not ready." Visual Basic has a device, called **error-trapping**, for preventing this and many other types of errors. If an error occurs while error-trapping is active, two things happen. An identifying number is assigned to the Number property of an object called Err, and the program jumps to some lines of code called an **error-handling routine**, which takes corrective measures based on the value of Err.Number. Some errors and the values they generate are as follows:

Type of error	Value of Err.Number
Subscript out of range	9
Division by zero	11
File not found	53
File already open	55
File already exists	58
Disk full	61
Disk not ready	71

To set up error-trapping inside a procedure, do the following:

1. Make the first line of the procedure

```
On Error GoTo ErrorHandler
```

2. Type in the lines to carry out the purpose of the procedure.
3. Make the last lines of the procedure

```
Exit SubErrorHandler:error-handling routineResume
```

The statement "On Error GoTo ErrorHandler" activates error-trapping. If an error occurs during the execution of a line of the procedure, the program will jump to the error-handling routine. The statement "Resume" causes the program to jump back to the line causing the error. The statement "Exit Sub", which causes an early exit from the procedure, prevents the error-handling routine from being entered when no error occurs. For instance, the following procedure has an error-handling routine that is called when a file cannot be found.

```
Private Sub OpenFile()
  On Error GoTo ErrorHandler
  Dim fileName As String
    fileName = InputBox("Enter the name of the file to be opened.")
    Open fileName For Input As #1
    Exit Sub
  ErrorHandler:
  Select Case Err.Number
    Case 53
      MsgBox "File not found. Try Again."
      fileName = InputBox("Enter the name of the file to be opened.")
    Case 71
      MsgBox "The drive door might be open - please check."
  End Select
  Resume
End Sub
```

The word "ErrorHandler", which is called a **line label**, can be replaced by any word of at most 40 letters. The line, which is placed just before the error-handling routine, must start at the left margin and must end with a colon. If "Resume" is replaced by "Resume Next", then the program will jump to the line following the line causing the error.

The line label must be in the same procedure as the On Error statement. However, the error-handling routine can call another procedure.

There are two variations of the On Error statement. The statement "On Error GoTo 0" turns off error-trapping. The statement "On Error Resume Next" specifies that when a run-time error occurs, execution continues with the statement following the statement where the error occurred.

COMMENTS

1. Sequential files make efficient use of disk space and are easy to create and use. Their disadvantages are as follows:

(a) Often a large portion of the file must be read in order to find one specific item.

(b) An individual item of the file cannot be changed or deleted easily.

Another type of file, known as a **random-access file**, has neither of the disadvantages of sequential files; however, random-access files typically use more disk space, require greater effort to program, and are not flexible in the variety and format of the stored data. Random-access files are discussed in Section 8.

2. Consider the sequential file shown in Figure 7-1 at the end of Example 2. This file is said to consist of three records of two fields each. A **record** holds all the data about a single individual. Each item of data is called a **field**. The three records are

   "Barbra", 1942

   "Ringo", 1940

   "Sylvester", 1946

   and the two fields are

   name field, year of birth field

7.2 USING SEQUENTIAL FILES

In addition to being accessed for information, sequential files are regularly updated by modifying certain pieces of data, removing some records, and adding new records. These tasks can be performed most efficiently if the files are first sorted.

SORTING SEQUENTIAL FILES

The records of a sequential file can be sorted on any field by first reading the data into parallel arrays and then sorting on a specific array.

EXAMPLE 1

The following program sorts the sequential file YOB.TXT of the previous section by year of birth.

```
Private Sub cmdSort_Click()
   Dim numPeople As Integer
   'Sort data from YOB.TXT file by year of birth
   numPeople = NumberOfRecords("YOB.TXT")
   ReDim nom(1 To numPeople) As String
   ReDim yearBorn(1 To numPeople) As Integer
   Call ReadData(nom(), yearBorn(), numPeople)
   Call SortData(nom(), yearBorn(), numPeople)
   Call ShowData(nom(), yearBorn(), numPeople)
   Call WriteData(nom(), yearBorn(), numPeople)
End Sub

Private Function NumberOfRecords(filespec As String) As Integer
   Dim nom As String, yearBorn As Integer
   Dim n As Integer 'Used to count records
   n = 0
   Open filespec For Input As #1
   Do While Not EOF(1)
      Input #1, nom, yearBorn
      n = n + 1
   Loop
   Close #1
```

```
  NumberOfRecords = n
End Function

Private Sub ReadData(nom() As String, yearBorn() As Integer, numPeople As _
Integer)
  Dim index As Integer
  'Read data from file into arrays
  Open "YOB.TXT" For Input As #1
    For index = 1 To numPeople Input #1, nom(index), yearBorn(index)
  Next index
  Close #1
End Sub

Private Sub ShowData(nom() As String, yearBorn() As Integer, numPeople As _
Integer)
  Dim index As Integer
  'Display the sorted list
  picShowData.Cls
  For index = 1 To numPeople
    picShowData.Print nom(index), yearBorn(index)
  Next index
End Sub

Private Sub SortData(nom() As String, yearBorn() As Integer, numPeople As
Integer)
  Dim passNum As Integer, index As Integer
  'Bubble sort arrays by year of birth
  For passNum = 1 To numPeople - 1
    For index = 1 To numPeople - passNum
      If yearBorn(index) > yearBorn(index + 1) Then
        Call SwapData(nom(), yearBorn(), index)
      End If
    Next index
  Next passNum
End Sub

Private Sub SwapData(nom() As String, yearBorn() As Integer, index As Integer)
  Dim stemp As String, ntemp As Integer
  'Swap names and years
  stemp = nom(index)
  nom(index) = nom(index + 1)
  nom(index + 1) = stemp
  ntemp = yearBorn(index)
  yearBorn(index) = yearBorn(index + 1)
  yearBorn(index + 1) = ntemp
End Sub

Private Sub WriteData(nom() As String, yearBorn() As Integer, numPeople As
Integer)
  Dim index As Integer
  'Write data back into file
  Open "YOB.TXT" For Output As #1
  For index = 1 To numPeople
    Write #1, nom(index), yearBorn(index)
  Next index
```

```
   Close #1
End Sub
```

[Run, and then click on the command button. The following is displayed in the picture box.]

Ringo 1940
Barbra 1942
Sylvester 1946

Merging Sequential Files

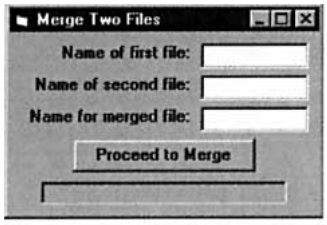
In Section 6.2, we considered an algorithm for merging two arrays. This same algorithm can be applied to merging two ordered files.

Suppose you have two ordered files (possibly with certain items appearing in both files), and you want to merge them into a third ordered file (without duplications). The technique for creating the third file is as follows.

1. Open the two ordered files For Input and open a third file For Output.
2. Try to get an item of data from each file.
3. Repeat steps (a) and (b) below until an item of data is not available in one of the files.
   (a) If one item precedes the other, write it into the third file and try to get another item of data from its file.
   (b) If the two items are identical, write one into the third file and try to get another item of data from each of the two ordered files.
4. At this point, an item of data has most likely been retrieved from one of the files and not yet written to the third file. In this case, write that item and all remaining items in that file to the third file.
5. Close the three files.

EXAMPLE 2

The following program merges two ordered files of numbers into a third file.



Object	Property	Setting
frmMerge	Caption	Merge Two Files
lblNameFirst	Caption	Name of first file:
txtNameFirst	Text	(blank)
lblNameSecond	Caption	Name of second file:
txtNameSecond	Text	(blank)
lblNameMerged	Caption	Name for merged file:
txtNameMerged	Text	(blank)
cmdProceed	Caption	Proceed to Merge
picProgress		

```
Private Sub cmdProceed_Click()
   Dim file1 As String, file2 As String, file3 As String
   Dim have1data As Boolean, have2data As Boolean
   Dim num1 As Single, num2 As Single
   Dim recCount As Integer 'Number of records in merged file
   'Merge two ordered files
   picProgress.Cls
   file1 = txtNameFirst.Text
   file2 = txtNameSecond.Text
   file3 = txtNameMerged.Text
   Open file1 For Input As #1
```

```
   Open file2 For Input As #2
   Open file3 For Output As #3
   have1data = Get1data(num1)
   have2data = Get2data(num2)
   recCount = 0
   Do While have1data And have2data
      Select Case num1
         Case Is < num2
            Write #3, num1
            have1data = Get1data(num1)
         Case Is > num2
            Write #3, num2
            have2data = Get2data(num2)
         Case num2
            Write #3, num1
            have1data = Get1data(num1)
            have2data = Get2data(num2)
      End Select
      recCount = recCount + 1
   Loop
   Do While have1data
      Write #3, num1
      recCount = recCount + 1
      have1data = Get1data(num1)
   Loop
   Do While have2data
      Write #3, num2
      recCount = recCount + 1
      have2data = Get2data(num2)
   Loop
   Close #1, #2, #3
   picProgress.Print recCount; "records written to "; file3
End Sub

Private Function Get1data(num1 As Single) As Boolean
   'If possible, read next value from file 1
   'Return value of True when new data are read; False if data not available
   If Not EOF(1) Then
        Input #1, num1
        Get1data = True
      Else
        Get1data = False
   End If
End Function

Private Function Get2data(num2 As Single) As Boolean
   'If possible, read next value from file 2
   'Return value True when new data are read; False if data not available
   If Not EOF(2) Then
        Input #2, num2
        Get2data = True
      Else
        Get2data = False
   End If
End Function
```

Control Break Processing

Suppose a small real estate company stores its sales data for a year in a sequential file in which each record contains four fields: month of sale, day of sale (1 through 31), address, and price. Typical data for the sales of the first quarter of a year are shown in Figure 7-2. The records are ordered by date of sale.

Month	Day	Address	Price
January	9	102 Elm Street	$203,000
January	20	1 Main Street	$315,200
January	25	5 Maple Street	$123,450
February	15	1 Center Street	$100,000
February	23	2 Vista Drive	$145,320
March	15	205 Rodeo Circle	$389,100

FIGURE 7-2 *Real Estate Sales for First Quarter of Year*

Figure 7-3 shows the output of a program that displays the total sales for the quarter year, with a subtotal for each month.

January	9	102 Elm Street	$203,000.00
January	20	1 Main Street	$315,200.00
January	25	5 Maple Street	$123,450.00
		Subtotal for January:	$641,650.00
February	15	1 Center Street	$100,000.00
February	23	2 Vista Drive	$145,320.00
		Subtotal for February:	$245,320.00
March	15	205 Rodeo Circle	$389,100.00
		Subtotal for March:	$389,100.00
Total for First Quarter: $1,276,070.00			

FIGURE 7-3 *Output of Example 3*

A program to produce the output of Figure 7-3 must calculate a subtotal at the end of each month. The variable holding the month triggers a subtotal whenever its value changes. Such a variable is called a **control variable** and each change of its value is called a **break**.

EXAMPLE 3

The following program produces the output of Figure 7-2. The data of Figure 7-2 are stored in the sequential file HOMESALE.TXT. The program allows for months with no sales. Because monthly subtotals will be printed, the month-of-sale field is an appropriate control variable.

```
Private Sub cmdCreateReport_Click()
   Dim currentMonth As String, newMonth As String
   Dim dayNum As Integer, address As String
   Dim price As Single, monthTotal As Single
   Dim yearTotal As Single, doneFlag As Boolean
   'Display home sales by month
   picReport.Cls
   Open "HOMESALE.TXT" For Input As #1
   currentMonth = "" 'Name of month being subtotaled
   monthTotal = 0
   yearTotal = 0
   doneFlag = False 'Flag to indicate end of list
   Do While Not doneFlag
```

```
      If Not EOF(1) Then
           Input #1, newMonth, dayNum, address, price
        Else
           doneFlag = True 'End of list
      End If
      If (newMonth <> currentMonth) Or (doneFlag) Then 'Control break processing
           If currentMonth <> "" Then 'Don't print subtotal before 1st month
              picReport.Print
              picReport.Print Tab(15); "Subtotal for "; currentMonth; ":";
              picReport.Print Tab(38); FormatCurrency(monthTotal)
              picReport.Print
        End If
        currentMonth = newMonth
        monthTotal = 0
      End If
      If Not doneFlag Then
           picReport.Print newMonth;
           picReport.Print Tab(11); FormatNumber(dayNum, 0);
           picReport.Print Tab(18); address;
           picReport.Print Tab(38); FormatCurrency(price)
           yearTotal = yearTotal + price
      End If
      monthTotal = monthTotal + price
   Loop
   Close #1
   picReport.Print "Total for First Quarter: "; FormatCurrency(yearTotal)
End Sub
```

COMMENTS

1. In the examples of this and the previous section, the files to be processed have been opened and closed within a single procedure. However, the solution to some programming problems requires that a file be opened just once the instant the program is run and stay open until the program is terminated. This is easily accomplished by placing the Open statement in the Form_Load event procedure and the Close and End statements in the click event procedure for a command button labeled "Quit."

7.3 A CASE STUDY: RECORDING CHECKS AND DEPOSITS

The purpose of this section is to take you through the design and implementation of a quality program for personal checkbook management. Nothing in this chapter shows off the power of Visual Basic better than the program in this section. That a user-friendly checkbook management program can be written in less than five pages of code clearly shows Visual Basic's ability to improve the productivity of programmers. It is easy to imagine an entire finance program, similar to programs that have generated millions of dollars of sales, being written in only a few weeks by using Visual Basic!

THE DESIGN OF THE PROGRAM

Though there are many commercial programs available for personal financial management, they include so many bells and whistles that their original purposes—keeping track of transactions

and reporting balances—have become obscured. The program in this section was designed specifically as a checkbook program. It keeps track of expenditures and deposits and produces a printed report. Adding a reconciliation feature would be easy enough, although we did not include one.

The program is supposed to be user-friendly. Therefore, it showcases many of the techniques and tools available in Visual Basic.

The general design goals for the program included the abilities to

- Automatically enter the user's name on each check and deposit slip.
- Automatically provide the next consecutive check or deposit slip number. (The user can override this feature if necessary.)
- Automatically provide the date. (Again, this feature can be overridden.)
- For each check, record the payee, the amount, and optionally a memo.
- For each deposit slip, record the source, the amount, and optionally a memo.
- Display the current balance at all times.
- Produce a printout detailing all transactions.

The User Interface

With Visual Basic we can place a replica of a check or deposit slip on the screen and let the user supply the information as if actually filling out a check or deposit slip. Figure 7-4 shows the form in its check mode. A picture box forms the boundary of the check. Below the picture box are two labels for the current balance and four command buttons.

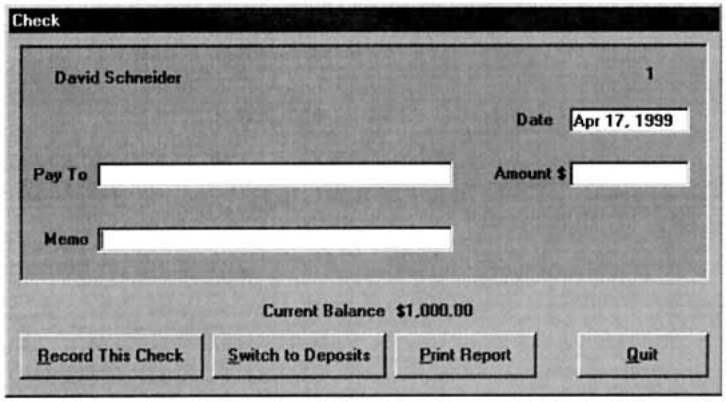


FIGURE 7-4 *Template for Entering a Check*

The first time the program is run, the user is asked for his or her name, the starting balance, and the numbers of the first check and deposit slip. Suppose the user's name is David Schneider, the first check has number 1, the starting balance is $1000, and the first deposit slip is also number 1. Figure 7-4 shows the form after the three pieces of input. The upper part of the form looks like a check. The check has a color of light turquoise blue (or cyan). The Date box is automatically set to today's date, but can be altered by the user. The user fills in the payee, amount, and optionally a memo. When the user pushes the Record This Check button, the information is written to a file, the balance is updated, and check number 2 appears.

To record a deposit, the user pushes the Switch to Deposits button. The form then appears as in Figure 7-5. The form's title bar now reads Deposit Slip, the words Pay To changes to Source, and the color of the slip changes to yellow. Also, in the buttons at the bottom of the form, the words Check and Deposit are interchanged. A deposit is recorded in much the same way as a check. When the Print Report button is pushed, a printout similar to the one in Figure 7-5 is printed on the printer.

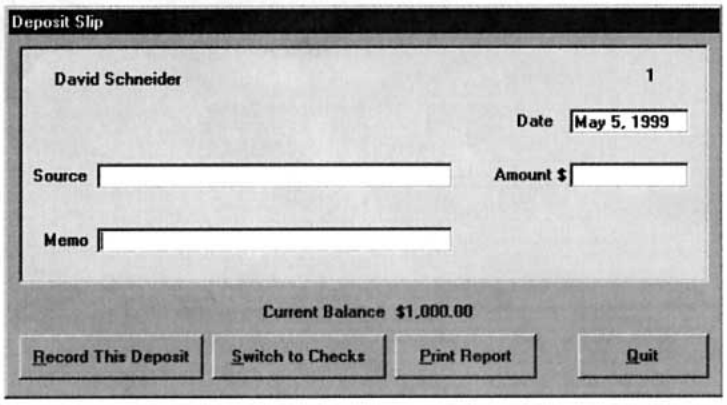


FIGURE 7-5 *Template for Entering a Deposit*

May 5, 1999

Name: David Schneider Starting balance: $1,000.00

Date	Transaction	Amount	Balance
Apr 21, 1999	Check #: 1	$75.95 Paid to: Land's End Memo: shirts	$924.05
Apr 29,1999	Check #: 2	$125.00 Paid to: Bethesda Coop Memo: groceries	$799.05
May 5,1999	Deposit #: 1	$245.00 Source: Prentice Hall Memo: typing expenses Ending Balance: $1,044.05	$1,044.05

FIGURE 7-6 *Sample Printout of Transactions*

The common design for the check and deposit slip allows one set of controls to be used for both items. Figure 7-7 shows the controls and their suggestive names. The caption of the label lblToFrom will change back and forth between Pay To and Source.

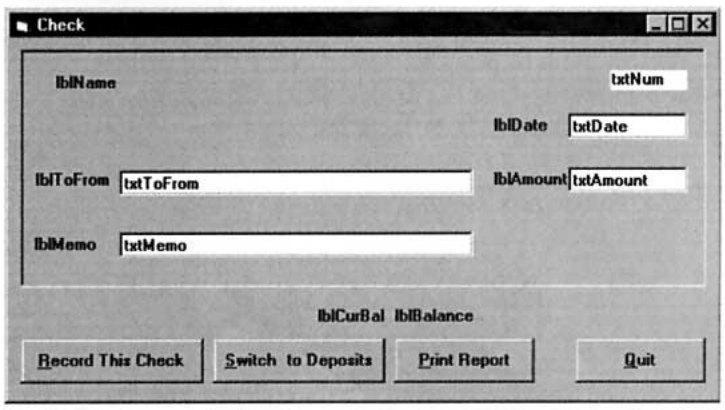


FIGURE 7-7 *Control Names for Checkbook Management Program.*

Table 7.1 lists the objects and their initial properties. Because the program will always begin by displaying the next check, the various captions and the BackColor property of the picture box could have been set at design time. We chose instead to leave these assignments

to the SetupCheck Sub procedure, which is normally used to switch from deposit entry to check entry, but also can be called by the Form_Load event procedure to prepare the initial mode (check or deposit) for the form.

TABLE 7.1
Objects and Initial Properties for the Checkbook Management Program

Object	Property	Setting
frmCheckbook		
picBox		
lblName	BackStyle	0 – Transparent
txtNum	BorderStyle	0 – None
lblDate	BackStyle	0 – Transparent
	Caption	Date
txtDate		
lblToFrom	BackStyle	0 – Transparent
txtToFrom		
lblAmount	BackStyle	0 – Transparent
	Caption	Amount $
txtAmount		
lblMemo	BackStyle	0 – Transparent
	Caption	Memo
txtMemo		
lblCurBal	Caption	Current Balance
lblBalance		
cmdRecord		
cmdMode		
cmdPrint	Caption	&Print Report
cmdQuit	Caption	&Quit

The transactions are stored in a data file named CHKBOOK.TXT. The first four entries of the file are the name to appear on the check or deposit slip, the starting balance, the number of the first check, and the number of the first deposit slip. After that, each transaction is recorded as a sequence of eight items—the type of transaction, the contents of txtToFrom, the current balance, the number of the last check, the number of the last deposit slip, the amount of money, the memo, and the date.

CODING THE PROGRAM

The top row of Figure 7-8 shows the different events to which the program must respond. Table 7.2 identifies the corresponding event procedures and the general procedures they call.

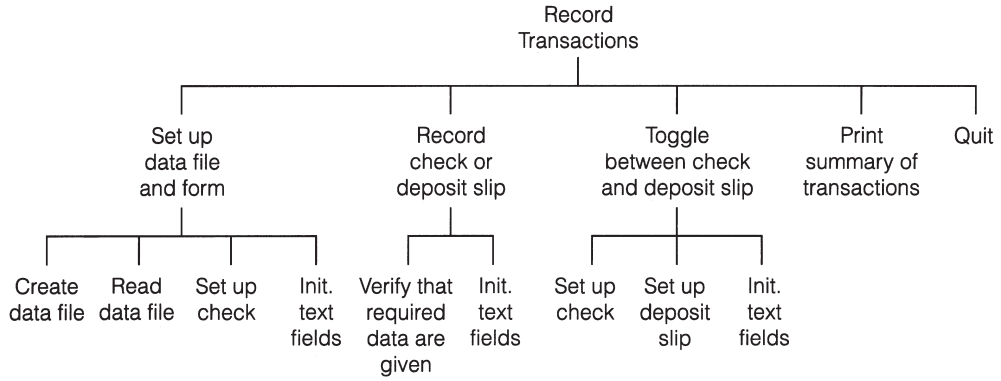


FIGURE 7-8 *Hierarchy Chart for Checkbook Management Program*

TABLE 7.2
Tasks and Their Procedures

Task	Procedure
1. Set up data file and form	Form_Load
1.1 Create data file	CreateDataFile
1.2 Read data file	ReadDataFile
1.3 Set up check	SetupCheck
1.4 Initialize text fields	InitializeFields
2. Record check or deposit slip	cmdRecord_Click
2.1 Verify that required data are given	AllDataGiven
2.2 Initialize text fields	InitializeFields
3. Toggle between check & deposit slip	cmdMode_Click
3.1 Set up check	SetupCheck
3.2 Set up deposit slip	SetupDeposit
3.3 Initialize text fields	InitializeFields
4. Print summary of transaction	cmdPrint_Click
5. Quit	cmdQuit_Click

Let's examine each event procedure.

1. **Form_Load** first looks to see if the file CHKBOOK.TXT has been created. The function Dir returns "CHKBOOK.TXT" if the file exists and otherwise returns the empty string. If CHKBOOK.TXT does not exist, the routine CreateDataFile is called. CreateDataFile prompts the user for the name to appear on the checks and deposit slips, the starting balance, and the numbers of the first check and deposit slip, and then writes these items to the data file. If CHKBOOK.TXT does exist, the routine ReadDataFile is called to read through the entire file to determine all information needed to proceed. The event procedure calls SetupCheck to set the transaction type to Check and set the appropriate captions and background colors for a check. The event procedure then calls InitializeFields, which initializes all the text boxes.

   In the first assignment statement of the procedure, the drive is specified as the A drive. Therefore, the data file will be written to and read from a diskette on the A drive. Feel free to change the letter A to whatever drive you prefer. You may even want to specify an entire path.

2. **cmdRecord_Click** first confirms that the required fields contain entries. This is accomplished by calling the function AllDataGiven. If the value returned is True, then cmdRecord_Click opens the data file for output as Append, sends eight pieces of data to the file, and then closes the file. When AllDataGiven returns False, the function itself pops up a message box to tell the user where information is needed. The user must type in the information and then press the Record button again.

3. **cmdMode_Click** toggles back and forth from a check to a deposit slip. It calls SetupCheck, or its analog SetupDeposit, and then calls InitializeFields.

4. **cmdPrint_Click** prints out a complete history of all transactions, as shown in Figure 7-6.

5. **cmdQuit_Click** ends the program.

```
Dim fileName As String 'Name of data file
Dim nameOnChk As String 'Name to appear on checks
Dim lastCkNum As Integer 'Number of last check written
Dim lastDpNum As Integer 'Number of last deposit slip processed
Dim curBal As Single 'Current balance in account
Dim transType As String 'Type of transaction, check or deposit

Private Function AllDataGiven() As Boolean
```

```
   Dim message As String
   'If one of the four required pieces of information
   'is missing, assign its name to message
   message = ""
   If txtDate.Text = "" Then
          message = "Date"
          txtDate.SetFocus
     ElseIf txtToFrom.Text = "" Then
     If transType = "Check" Then
          message = "Pay To"
       Else
          message = "Source"
       End If
       txtToFrom.SetFocus
     ElseIf txtAmount.Text = "" Then
       message = "Amount"
       txtAmount.SetFocus
     ElseIf txtNum.Text = "" Then
       If transType = "Check" Then
            message = "Check Number"
          Else
            message = "Deposit Number"
       End If
       txtNum.SetFocus
     End If
   If message = "" Then
        'All required data fields have been filled; recording can proceed
        AllDataGiven = True
     Else
        'Advise user of required data that are missing
        MsgBox "The '" & message & " 'field must be filled", , "Error"
        AllDataGiven = False
   End If
End Function

Private Sub cmdMode_Click()
   'Toggle from Check to/from Deposit Slip
   If transType = "Check" Then
        Call SetupDeposit
     Else 'transType = "Deposit"
        Call SetupCheck
   End If
   Call InitializeFields
   txtToFrom.SetFocus
End Sub

Private Sub cmdPrint_Click()
   Dim temp As String, lineNo As Integer
   Dim nameOnChk As String, balance As Single, ck As Integer, dp As Integer
   Dim toFrom As String, amount As String, memo As String, theDate As String
   'Print out a detailed list of all transactions.
   temp = frmCheckbook.Caption                'Save the current form caption
   frmCheckbook.Caption = "Printing..."       'Set form caption to indicate printing
   lineNo = 1                                 'Line number being printed
   Open fileName For Input As #1              'Open the file
   Input #1, nameOnChk, balance, ck, dp       'Read in the file header
```

```
    'Print the details of the individual transactions.
    Do Until EOF(1)
      If lineNo >= 57 Then
    '57 or more lines have been printed; start a new page
    Printer.NewPage
    lineNo = 1
      End If
      If lineNo = 1 Then
        'Print the report header
        Printer.Print
        Printer.Print "Name: "; nameOnChk; Tab(65); FormatDateTime(Now,
        vbLongDate)
        Printer.Print
        Printer.Print , "Starting balance: "; FormatCurrency(balance)
        Printer.Print
        Printer.Print "Date", "Transaction"; Tab(50); "Amount";
        Printer.Print Tab(65); "Balance"
        Printer.Print "____", "___________"; Tab(50); "______";
        Printer.Print Tab(65); "_______"
        Printer.Print
        Printer.Print
        lineNo = 10
      End If
      Input #1, transType, toFrom, balance, ck, dp, amount, memo, theDate
      If transType = "Check" Then
           Printer.Print theDate, "Check #: "; ck; Tab(50); amount;
           Printer.Print Tab(65); FormatCurrency(balance)
           Printer.Print , "Paid to: "; toFrom
        Else 'Transaction was a deposit
           Printer.Print theDate, "Deposit #: "; dp; Tab(50); amount;
           Printer.Print Tab(65); FormatCurrency(balance)
           Printer.Print , "Source: "; toFrom
      End If
      lineNo = lineNo + 2
      'If there was a memo, then print it.
      If memo <> "" Then
           Printer.Print , "Memo: "; memo
           lineNo = lineNo + 1
      End If
      Printer.Print
      lineNo = lineNo + 1
    Loop
    Close #1 'Close the file
    'Print the ending balance
    Printer.Print
    Printer.Print , "Ending Balance: "; FormatCurrency(balance)
    Printer.EndDoc                          'Send the output to the Printer
    frmCheckbook.Caption = temp             'Restore the form caption
    txtToFrom.SetFocus                      'Set focus for the next entry
End Sub

Private Sub cmdQuit_Click()
    'Exit the program
    End
End Sub
```

```
Private Sub cmdRecord_Click()
   Dim amt As String, amount As Single, itemNum As Integer
   'Check to ensure all required fields are filled
   If AllDataGiven Then
      amt = txtAmount.Text 'Amount of transaction as string
      amount = Val(amt) 'Amount of transaction as number
      amt = FormatCurrency(amt) itemNum = Val(txtNum.Text)
      If transType = "Check" Then
           curBal = curBal - amount
           lastCkNum = itemNum
        Else 'transType = "Deposit"
           curBal = curBal + amount
           lastDpNum = itemNum
   End If
           lblBalance.Caption = FormatCurrency(curBal)
           Open fileName For Append As #1
           Write #1, transType, txtToFrom.Text, curBal, lastCkNum, lastDpNum, amt, _
              txtMemo.Text, txtDate.Text
           Close #1
           Call InitializeFields
           txtToFrom.SetFocus
   End If
End Sub

Private Sub CreateDataFile()
   Dim startBal As Single, ckNum As integer
   'The first time the program is run, create a data file
   Open fileName For Output As #1
   nameOnChk = InputBox("Name to appear on checks:")
   startBal = Val(InputBox("Starting balance:"))
   ckNum = Val(InputBox("Number of the first check:"))
   lastCkNum = ckNum - 1 'Number of "last" check written
   ckNum = Val(InputBox("Number of the first deposit slip:"))
   lastDpNum = ckNum - 1 'Number of "last" deposit slip processed
   curBal = startBal 'Set current balance
   'First record in data file records name to appear on checks
   'plus initial data for account
   Write #1, nameOnChk, startBal, lastCkNum, lastDpNum
   Close #1
End Sub

Private Sub Form_Load()
   Dim drive As String, today As String
   'If no data file exists, create one. Otherwise, open the
   'data file and get the user's name, last used check and
   'deposit slip numbers, and current balance.
   'In next line adjust drive as necessary drive = "A:"
   'Drive (or path) for data file
   fileName = drive & "CHKBOOK.TXT" 'Program uses one data file
   If Dir(fileName) = "" Then
        'Data file does not exist, so create it and obtain initial data
        Call CreateDataFile
     Else
        Call ReadDataFile
   End If
```

```
   'Set name and balance labels
   lblName.Caption = nameOnChk
   lblBalance.Caption = FormatCurrency(curBal)
   'Set the date field to the current date
   today = FormatDateTime(Now, vbLongDate)
   txtDate.Text = Mid(today, 2 + InStr(today, ","))
   Call SetupCheck 'Always start session with checks
   Call InitializeFields
End Sub

Private Sub InitializeFields()
   'Initialize all text entry fields except date
   txtToFrom.Text = "" txtAmount.Text = ""
   txtMemo.Text = ""
   If transType = "Check" Then
        'Make  txtNum text box reflect next check number
        txtNum.Text = Str(lastCkNum + 1)
     Else 'transType = "Deposit"
        'Make txtNum text box reflect next deposit slip number
        txtNum.Text = Str(lastDpNum + 1)
   End If
End Sub

Private Sub ReadDataFile()
   Dim t As String, s As String, n As String, m As String, d As String
   'Recover name to appear on checks, current balance,
   'number of last check written, and number of last deposit slip processed
   Open fileName For Input As #1
   Input #1, nameOnChk, curBal, lastCkNum, lastDpNum
   Do Until EOF(1)
      'Read to the end of the file to recover the current balance and the
      'last values recorded for ckNum and dpNum.
      't, s, n, m and d are dummy variables and are not used at this point
      Input #1, t, s, curBal, lastCkNum, lastDpNum, n, m, d
   Loop
   Close #1
End Sub

Private Sub SetupCheck()
   'Prepare form for the entry of a check
   transType = "Check"
   frmCheckbook.Caption = "Check"
   lblToFrom.Caption = "Pay To"
   cmdRecord.Caption = "&Record This Check"
   cmdMode.Caption = "&Switch to Deposits"
   picBox.BackColor = vbCyan 'color of check is light turquoise blue
   txtNum.BackColor = vbCyan
End Sub

Private Sub SetupDeposit()
   'Prepare form for the entry of a deposit
   transType = "Deposit"
   frmCheckbook.Caption = "Deposit Slip"
   lblToFrom.Caption = "Source"
```

```
   cmdRecord.Caption = "&Record This Deposit"
   cmdMode.Caption = "&Switch to Checks"
   picBox.BackColor = vbYellow 'color of deposit slip is yellow
   txtNum.BackColor = vbYellow
End Sub
```

SUMMARY

1. When sequential files are opened, we must specify whether they will be created and written to, added to, or read from by use of the terms Output, Append, or Input. The file must be *closed* before the operation is changed. Data are written to the file with Write # statements and retrieved with Input # statements. The EOF function tells if we have read to the end of the file.
2. A sequential file can be ordered by placing its data in arrays, sorting the arrays, and then writing the ordered data into a file.

PROGRAMMING PROJECTS

1. Table 7.3 gives the leading eight soft drinks in 1997 and their percentage share of the market. Write and execute a program to place these data into a sequential file. Then write a second program to use the file to

   (a) display the eight brands and their gross sales in billions. (The entire soft drink industry grosses about $42 billion.)

   (b) calculate the total percentage market share of the leading eight soft drinks.

TABLE 7.3
Leading Soft Drinks and Percentages of 1997 Market Share

Coke Classic	20.6	Sprite	6.2
Pepsi-Cola	14.5	Dr. Pepper	5.9
Diet Coke	8.5	Diet Pepsi	5.5
Mountain Dew	6.3	7 Up	2.3

Source: Beverage Digest, 2/12/98

2. Suppose the sequential file ALE.TXT contains the information shown in Table 7.4. Write a program to use the file to produce Table 7.5 in which the baseball teams are in descending order by the percentage of games won. ***Note:*** A batting average can be displayed in standard form with `Format-Number(ave, 3, vbFalse)`.

TABLE 7.4
American League East Games Won and Lost in 1998

Team	Won	Lost
Baltimore	79	83
Boston	92	70
New York	114	48
Tampa Bay	63	99
Toronto	88	74

TABLE 7.5
Final 1998 American League East Standings

	American League East		
	W	L	Pct
New York	114	48	.704
Boston	92	70	.568
Toronto	88	74	.543
Baltimore	79	83	.488
Tampa Bay	63	99	.389

3. Write a rudimentary word processing program. The program should do the following:
   (a) Use InputBox to request the name of the sequential file to hold the document being created.
   (b) Set the label for a text box to "Enter Line 1" and allow the user to enter the first line of the document into a text box.
   (c) When the Enter key is pressed or a "Record Line" command button is clicked, determine if the line is acceptable. Blank lines are acceptable input, but lines exceeding 60 characters in length should not be accepted. Advise the user of the problem with a message box, and then set the focus back to the text box so that the user can edit the line to an acceptable length.
   (d) When an acceptable line is entered, write this line to the file and display it in a picture box.
   (e) Change the prompt to "Enter Line 2", clear the text box, allow the user to enter the second line of the document into the text box, and carry out (c) for this line using the same picture box. (Determine in advance how many lines the picture box can display and only clear the picture box when the lines already displayed do not leave room for a new line.)
   (f) Continue as in (d) with appropriate labels for subsequent lines until the user clicks on a "Finished" command button.
   (g) Clear the picture box and display the number of lines written and the name of the data file created.
4. Write a program that counts the number of times a word occurs in the sequential file created in Programming Project 3. The file name and word should be read from text boxes. The search should not be sensitive to the case of the letters. For instance, opening a file that contained the first three sentences of the directions to this problem and searching for "the" would produce the output: "the" occurs six times.
5. *Create and Maintain Telephone Directories.* Write a program to create and maintain telephone directories. Each directory will be a separate sequential file. The following command buttons should be available:
   (a) Select a directory to access. A list of directories that have been created should be stored in a separate sequential file. When a request is made to open a directory, the list of available directories should be displayed as part of an InputBox prompt requesting the name of the directory to be accessed. If the user responds with a directory name not listed, the desire to create a new directory should be confirmed, and then the new directory created and added to the list of existing directories.
   (b) Add name and phone number (as given in the text boxes) to the end of the current directory.
   (c) Delete name (as given in the text box) from the current directory.
   (d) Sort the current directory into name order.

(e) Print out the names and phone numbers contained in the current directory.

(f) Terminate the program.

6. Table 7.6 contains the statistics for a stock portfolio. (The current prices are given for October 1, 1998.)

TABLE 7.6
Stock Portfolio

Stock	Number of Shares	Date Purchased	Purchase Price/Share	Current Price/Share
Amgen	200	8/19/97	50.750	75.625
Delta Airlines	100	12/3/97	111.750	97.250
Novell	500	8/27/97	10.375	12.250
PPG	100	12/18/97	56.750	54.500
Timken	300	3/13/98	34.625	15.125

(a) Compose a program to create the sequential file STOCKS.TXT containing the information in Table 7.6.

(b) Compose a program to perform the following tasks. A possible form design is shown in Figure 7-9.

(1) Display the information in the file STOCKS.TXT as in Table 7.6 when the user clicks on a "Display Stocks" command button.

(2) Add an additional stock onto the end of the file STOCKS.TXT when the user clicks on an "Add Stock" command button. The data for the new stock should be read from appropriately labeled text boxes.

(3) Update the Current Price/Share of a stock in the file STOCKS.TXT when the user clicks on an "Update Price" command button. The name of the stock to be updated and the new price should be read from the appropriate text boxes. The file STOCKS.TXT should then be copied to a temp file until the specified stock is found. The update record for this stock should then be written to the temp file, followed by all remaining records in STOCKS.TXT. Finally, the original STOCKS.TXT file should be erased and the temp file renamed to STOCKS.TXT.

(4) Process the data in the file STOCKS.TXT and produce the display shown in Figure 7-10 when a "Show Profit/Loss" command button is clicked.

(5) Quit.

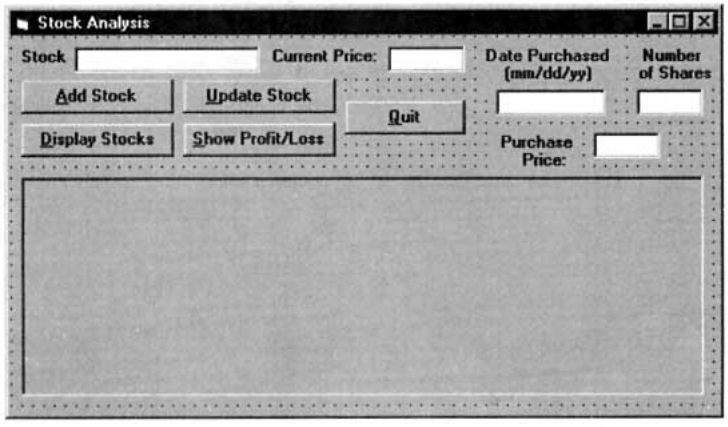


FIGURE 7-9 *Possible Form Design for Programming Project 6*

Stock	Cost	Current Value	Profit (or Loss)
Amgen	$10,150.00	$15,125.00	$4,975.00
Delta Airlines	$11,175.00	$9,725.00	($1,450.00)
Novell	$5,187.50	$6,125.00	$937.50
PPG	$5,675.00	$5,450.00	($225.00)
Timken	$10,387.50	$4,537.50	($5,850.00)

FIGURE 7-10 *Output of Project 6*

7. A department store has a file containing all sales transacted for a year. Each record contains a customer's name, zip code, and amount of the sale. The file is ordered first by zip code and then by name. Write a program to display the total sales for each customer, the total sales for each zip code, and the total sales for the store. For instance, if the first six records of the file are

```
"Adams, John", 10023, 34.50
"Adams, John", 10023, 60.00
"Jones, Bob", 10023, 62.45
"Green, Mary", 12345, 54.00
"Howard, Sue", 12345, 79.25
"Smith, George", 20001, 25.10
```

then the output in the picture box will begin as shown in Figure 7-11.

Customer	Total Sales
Adams, John	$94.50
Jones, Bob	$62.45
Total sales of zip code 10023: $156.95	
Green, Mary	$54.00

FIGURE 7-11 *Sample Output for Programming Project 7*

8. *Savings Account.* NAMES.TXT is a sequential file containing the name, account number, and beginning-of-month balance for each depositor. TRANS.TXT is a sequential file containing all the transactions (deposits and withdrawals) for the month. Use TRANS.TXT to upgrade NAMES.TXT. For each customer, print a statement similar to the one received from banks that shows all transactions and the end-of-month balance. Also, record all overdrawn accounts in a file. (As an optional embellishment, deduct a penalty if the balance fell below a certain level any time during the month. The penalty could include a fixed fee of $10 plus a charge of $1 for each check and deposit.) ***Hint:*** Assume no more than 500 transactions have occurred.
9. A fuel economy study was carried out for five models of cars. Each car was driven for 100 miles of city driving, and then the model of the car and the number of gallons used were placed in the sequential file MILEAGE.TXT with the statement

```
Write #1, modelName, gallons
```

Table 7.7 shows the first entries of the file. Write a program to display the models and their average miles per gallon in decreasing order with respect to mileage. The program should utilize three parallel arrays of range 1 to 5. The first array should record the name of each model of car. This array is initially empty; each car model name is added when first encountered in reading the file. The second array should record the number of test vehicles for each model. The third array should record the total number of gallons used by that model.

Note: The first array must be searched each time a record is read to determine the appropriate index to use with the other two arrays.

TABLE 7.7
Gallons of Gasoline Used in 100 Miles of City Driving

Model	Gal	Model	Gal	Model	Gal
LeBaron	4.9	Cutlass	4.5	Cutlass	4.6
Escort	4.1	Escort	3.8	LeBaron	5.1
Beretta	4.3	Escort	3.9	Escort	3.8
Skylark	4.5	Skylark	4.6	Cutlass	4.4

SECTION

8

RANDOM-ACCESS FILES

8.1 USER-DEFINED DATA TYPES

Records provide a convenient way of packaging as a single unit several related variables of different types. Before we can explore this powerful variable type, we must first explore a new category of variable, the fixed-length string.

FIXED-LENGTH STRINGS

Fixed-length string variables are named following the same rules as other variable types. They are declared by statements of the form

```
Dim var As String * n
```

where *n* is a positive integer. After such a declaration, the value of *var* will always be a string of length *n*. Suppose *info* is an ordinary string and a statement of the form

```
var = info
```

is executed. If *info* has more than *n* characters, then only the first *n* characters will be assigned to *var*. If *info* has less than *n* characters, then spaces will be added to the end of the string to guarantee that *var* has length *n*.

EXAMPLE 1

The following program uses fixed-length strings. In the output, San Francisco is truncated to a string of length 9 and Detroit is padded on the right with two blank spaces.

```
Private Sub cmdGo_Click()
  Dim city As String * 9
  'Illustrate fixed-length strings
  picOutput.Cls
  picOutput.Print "123456789"
  city = "San Francisco"
  picOutput.Print city
  city = "Detroit"
  picOutput.Print city; "MI"
  picOutput.Print Len(city)
End Sub
```

[Set picOutput's Font property to Courier. Run, and click the command button.]

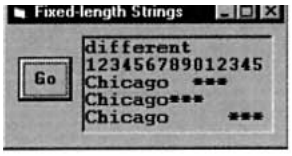
Care must be taken when comparing an ordinary (variable-length) string with a fixed-length string or comparing two fixed-length strings of different lengths.

EXAMPLE 2

In the following program, the strings assigned to the variables *town*, *city*, and *municipality* have lengths 7, 9, and 12, respectively, and therefore are all different.

```
Private Sub cmdGo_Click()
   Dim town As String * 7
   Dim city As String * 9
   Dim municipality As String * 12
   'Illustrate fixed-length strings
   town = "Chicago"
   city = "Chicago"
   municipality = "Chicago"
   picOutput.Cls
   If (city = town) Or (city = municipality) Then
       picOutput.Print "same"
     Else
       picOutput.Print "different"
   End If
   picOutput.Print "123456789012345"
   picOutput.Print city & "***"
   picOutput.Print town & "***"
   picOutput.Print municipality & "***"
End Sub
```

[Set picOutput's Font property to Courier. Run, and click the command button.]



There are times when we want to consider the values assigned to variables of different types as being the same, such as *city* and *town* in Example 2. In this situation, the function RTrim comes to the rescue. If *info* is an ordinary string or a fixed-length string, then the value of

RTrim(info)

is the (variable-length) string consisting of *info* with all right-hand spaces removed. For instance, the value of RTrim("hello") is the string "hello". In Example 2, if the If block is changed to

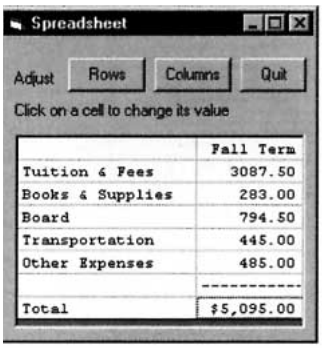
```
If (RTrim(city) = town) And (RTrim(city) = RTrim(municipality)) Then
     picOutput.Print "same"
  Else
     picOutput.Print "different"
End If
```

then the first line of the output will be "same".

```
Private Sub cmdQuit_Click()
   End
End Sub

Private Sub Form_Load()
   'Set default number of rows and columns
   numRows = 8 'row 0 is for headings, last 2 rows are for totals
   numCols = 2 'column 0 is for category names
   Call SetUpGrid
   Call ShowValues
   Call ShowTotals
End Sub

Private Sub msgSprdSht_Click()
   Dim temp As String, message As String
   'Obtain new value for cell if it is not in the "total" row
   If messageSprdSht.Row < numRows - 2 Then
      message = "Enter new value for the row "
      message = message & Str(msgSprdSht.Row + 1) & " column "
      message = message & Str(msgSprdSht.Col + 1) & " cell:"
      temp = InputBox(message,,msgSprdSht.Text) 'Propose old value as default
      If msgSprdSht.Col = 0 Then
           msgSprdSht.Text = temp
        ElseIf msgSprdsht.Row = 0 Then
   msgSprdSht.Text = temp
        Else
           msgSprdsht.Text = FormatNumber(Val(temp)vbFalse)
           Call ShowTotals
      End If
   End If
End Sub

Private Sub SetUpGrid()
   Dim colNum As Integer
   'Set up grid msgSprdSht.Col = 0
   msgSprdSht.Row = msgSprdSht.Rows - 1
   msgSprdSht.Text = "" 'erase "Total" in case increasing rows
   msgSprdSht.Rows = numRows
   msgSprdSht.Cols = numCols
   'Set column widths; right-justify columns with numeric data
   msgSprdSht.ColWidth(0) = 2000 'space for category names
   msgSprdSht.ColAlignment(0) = 1 'show data left-justified
   For colNum = 1 To numCols - 1
      msgSprdSht.ColWidth(colNum) = 1200 'space for dollar amounts
      msgSprdSht.ColAlignment(colNum) = 7 'show data right-justified
   Next colNum 'Set overall grid size to minimum needed for the data
   msgSprdSht.Width = 2000 + (numCols - 1) * 1200 + 15 * (numCols + 1) + 8
   msgSprdSht.Height = numRows*msgSprdSht.RowHeight(0)+15*(numRows + 1)+8
   'Adjust form to accommodate grid and other controls
   frmSprdSht.Width = msgSprdSht.Left + msgSprdSht.Width + 200
   frmSprdSht.Height = msgSprdSht.Top + msgSprdSht.Height + 500
   frmSprdSht.Top = 0 frmSprdSht.Left = 0
End Sub
```

```
Private Sub ShowTotals()
   Dim colNum As Integer, rowNum As Integer, total As Single
   'Compute and display total of each numeric column
   msgSprdSht.Row = numRows - 1
   msgSprdSht.Col = 0
   msgSprdSht.Text = "Total"
   For colNum = 1 To numCols - 1
      total = 0
      For rowNum = 1 To numRows - 3
         msgSprdSht.Row = rowNum
         msgSprdSht.Col = colNum
         total = total + Val(msgSprdSht.Text)
      Next rowNum
      msgSprdSht.Row = numRows - 2
      msgSprdSht.Text = "---------"
      msgSprdSht.Row = numRows - 1
      msgSprdSht.Text = FormatCurrency(total)
   Next colNum
End Sub

Private Sub ShowValues()
   Dim rowNum As Integer, colNum As Integer
   'Refresh values displayed in cells
   For rowNum = 1 To numRows - 1
      For colNum = 1 To numCols - 1
         msgSprdSht.Row = rowNum
         msgSprdSht.Col = colNum
         msgSprdSht.Text = FormatNumber(Val(msgSprdSht.Text)vbFalse)
      Next colNum
   Next rowNum
End Sub
```

[A possible run of the program is shown.]



So far we have used the Text property of grids to place strings into cells. Grids also have a Picture property. A picture (such as a .BMP file created with Paint or an .ICO file from Visual Basic's icon directory) is placed into the current cell with a statement of the form

```
Set msgFlex.CellPicture = LoadPicture("filespec")
```

If both text and a picture are assigned to a cell, then the picture appears in the upper left portion of the cell, and the text appears to the right of the picture.

The Menu Control

Visual Basic forms can have menu bars similar to those in most Windows applications. Figure 10-11 shows a typical menu, with the submenu for the Font menu item dropped down.

Here, the menu bar contains two menu items (Font and Size), referred to as **top-level** menu items. When the Font menu item is clicked, a dropdown list of two second-level menu items (Courier and TimesRm) appears. Although not visible here, the dropdown list under Size contains the two second-level menu items "12" and "24". Each menu item is treated as a distinct control that responds to only one event—the click event. The click event is triggered not only by the click of the mouse button, but also for top-level items by pressing Alt+*accessKey* and for second-level items by just pressing the access key. The click event for the Courier menu item in Figure 10-11 can be activated directly by pressing the shortcut key F1.

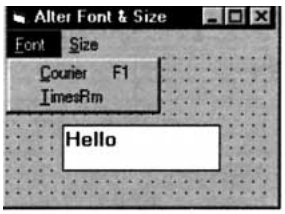


FIGURE 10-11 *A Simple Menu*

Menus are created with the Menu Editor window available from the Tools menu on the Visual Basic main menu bar. Figure 10-12 shows the Menu Design window used to create the menu in Figure 10-11. Each menu item has a Caption property (what the user sees) and a Name property (used to refer to the item in the code.) For instance, the last menu item in Figure 10-12 has Caption property "24" and Name property "mnu24". The following steps are used to create the Font-Size menu:

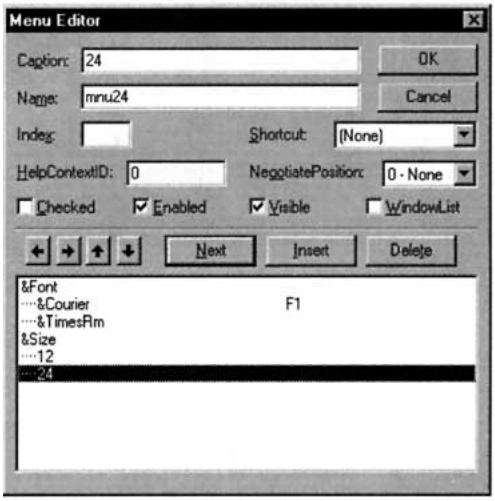


FIGURE 10-12 *The Menu Editor Window Used to Create the Menu in Figure 10-11*

1. Type &Font into the Caption box and type mnuFont into the Name box.
2. Click on the Next button.
3. Click on the Right Arrow button. (This creates the ellipses and indents the next menu item, which will be a second-level item.)
4. Type &Courier into the Caption box and type mnuCourier into the Name box.
5. Click on the arrow to the right of the Shortcut box and select F1 from the dropdown list.
6. Click on the Next button.
7. Type &TimesRm into the Caption box and type mnuTimesRm into the Name box.
8. Click on the Next button:

9. Click on the Left Arrow button. (This causes the next item to appear flush left to indicate that it is a top-level menu item.)
10. Type &Size into the Caption box and type mnuSize into the Name box.
11. Click on the Next button and then click on the Right Arrow button.
12. Type 12 into the Caption box and type mnu12 into the Name box.
13. Click on the Next button.
14. Type 24 into the Caption box and type mnu24 into the Name box. Your Menu Editor window should now appear as in Figure 10-12.
15. Click the OK button to close the Menu Editor window.

Three of the check boxes on the Menu Editor window are especially useful. When the Checked box is checked, a checkmark appears in front of the menu item. This checkmark can be altered in code with statements such as mnuItem.Checked = False and mnuItem.Checked = True. When the Enable box is unchecked, the menu item appears gray and does not respond to the click event. The enabled state can be altered in code with statements such as mnuItem.Enabled = False and mnuItem.Enabled = True. When the Visible property is unchecked, the menu item is invisible.

EXAMPLE 3

The following program creates the application in Figure 10-11, in which the menu is used to alter the appearance of the contents of a text box. The form has caption "Alter Font & Size" and the properties of the menu items are as created before.

```
Private Sub mnu12_Click()
  txtInfo.Font.Size = 12
End Sub

Private Sub mnu24_Click()
  txtInfo.Font.Size = 24
End Sub

Private Sub mnuCourier_Click()
  txtInfo.Font.Name = "Courier"
End Sub

Private Sub mnuTimesRm_Click()
  txtInfo.Font.Name = "Times New Roman"
End Sub
```

THE CLIPBOARD OBJECT

The clipboard object is used to copy or move text from one location to another. It is maintained by Windows and therefore even can be used to transfer information from one Windows application to another. It is actually a portion of memory that holds text and has no properties or events.

If *str* is a string, then the statement

```
Clipboard.SetText str
```

replaces any text currently in the clipboard with *str*. The statement

```
str = Clipboard.GetText()
```

assigns the text in the clipboard to the string variable *str*.

The statement

```
Clipboard.Clear
```

deletes the contents of the clipboard.

A portion of the text in a text box or combo box can be **selected** by dragging the mouse across it or by moving the cursor across it while holding down the Shift key. After you select text, you can place it into the clipboard by pressing Ctrl+C. Also, if the cursor is in a text box and you press Ctrl+V, the contents of the clipboard will be inserted at the cursor position. These tasks also can be carried out in code. The SelText property of a text box holds the selected string from the text box and a statement such as

```
Clipboard.SetText txtBox.SelText
```

copies this selected string into the clipboard. The statement

```
txtBox.SelText = Clipboard.GetText()
```

replaces the selected portion of txtBox with the contents of the clipboard. If nothing has been selected, the statement inserts the contents of the clipboard into txtBox at the cursor position. The clipboard can actually hold any type of data, including graphics. Any time you use the Copy menu item, you are putting information into the clipboard. The Paste menu item sends that data to your program.

Multiple Forms

A Visual Basic program can contain more than one form. Additional forms are created from the Project menu with Add Form (Alt/P/F/Enter). The name of each form appears in the Project Explorer window, and any form can be made the active form by double-clicking on its name in the Project Explorer window. (***Hint:*** After creating a new form, move it down slightly so that you can see at least the title bars of the other forms. Then you can activate any form by just clicking on its title bar.) The second form has default name Form2, the third form has default name Form3, and so on. Forms are hidden or activated with statements such as

```
Form1.Hide
```

or

```
Form2.Show
```

When a program is run, the first form created is the only one visible. After that, the Hide and Show methods can be used to determine what forms appear.

Often, additional forms, such as message and dialog boxes, are displayed to present a special message or request specific information. When a message or dialog box appears, the user cannot shift the focus to another form without first hiding the message or dialog box by clicking an OK or Cancel command button. If a form is displayed with a statement of the type

```
formName.Show 1    or    formName.Show vbModal
```

then the form will exhibit this same behavior. The user will not be allowed to shift the focus back to the calling form until *formName* is hidden. Such a form is said to be **modal**. It is customary to set the BorderStyle property of modal forms to "3-Fixed Dialog".

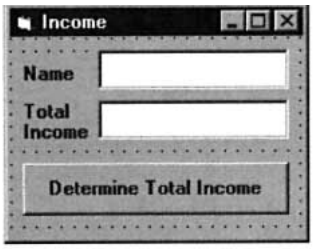
Each form has its own controls and code. However, code from one form can refer to a control in another form. If so, the control must be prefixed with the name of the other form, followed by a period. For instance, the statement

```
Form2.txtBox.Text = "Hello"
```

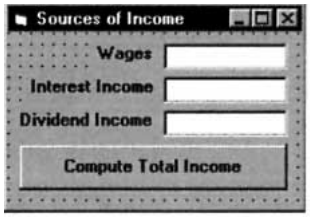
in Form1 causes text to be displayed in a text box on Form2. (***Note:*** Two forms can have a text box named txtBox. Code using the name txtBox refers to the text box in its own form unless prefixed with the name of another form.)

EXAMPLE 4

The following program uses a second form as a dialog box to total the different sources of income. Initially, only frmIncome is visible. The user types in his or her name and then either can type in the income or click on the command button for assistance in totaling the different sources of income. Clicking on the command button from frmIncome causes frmSources to appear and be active. The user fills in the three text boxes and then clicks on the command button to have the amounts totaled and displayed in the income text box of the first form.



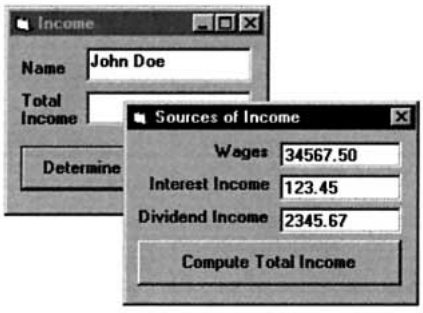
Object	Property	Setting
frmIncome	Caption	Income
lblName	Caption	Name
txtName	Text	(blank)
lblTotal	Caption	Total Income
txtTotal	Text	(blank)
cmdShowTot	Caption	Determine Total Income



Object	Property	Setting
frmSources	Caption	Sources of Income
	BorderStyle	3 – Fixed Dialog
lblWages	Caption	Wages
txtWages	Text	(blank)
lblInterest	Caption	Interest Income
txtInterest	Text	(blank)
lblDividend	Caption	Dividend Income
txtDividend	Text	(blank)
cmdCompute	Caption	Compute Total Income

```
Private Sub cmdShowTot_Click()
   frmSources.Show vbModal
End Sub

Private Sub cmdCompute_Click()
   Dim sum As Single
   sum = Val(txtWages.Text) + Val(txtInterest.Text) + Val(txtDividend.Text)
   frmIncome.txtTotal.Text = FormatCurrency(Str(sum))
   frmSources.Hide
End Sub
```

[Run, enter name, click the command button, and fill in the sources of income.]



All variables declared and general procedures created in the (General) object of a form are local to that form; that is, they are not available to any other form. Such variables and procedures are said to be of **form level**. However, you can declare global variables and pro-

cedures that are available to all forms. To do so, select Add Module from the Project menu and then double-click on the Module icon. A code window will appear with Module1 in the title bar. Procedures created in this window with the Public keyword will be available to all forms. To declare a variable that is available to all forms, declare it in the Module1 window, but use the word Public instead of Dim. For instance, if the statement

```
Public person As String
```

appears in the Module1 code window, the variable *person* can be accessed anywhere in the program.

The contents of this code window are said to form a (Standard) code module. Project Explorer shows the names of all forms and code modules and gives you access to the code from any form or module. When a code module is selected, you can save it by choosing Save Module As from the File menu. You add an existing code module to a program by choosing Add File from the Project menu.

The Common Dialog Control

The common dialog control does not initially appear in your Toolbox. To add the control, select Components from the Project menu, click the Controls tab, and click on the check box to the left of "Microsoft Common Dialog Control 6.0." Then press the OK button. By convention, names of common dialog boxes have the prefix dlg.

The common dialog control can produce each of the useful dialog boxes in Figures 10.13 through 10.17, thereby saving the programmer the trouble of designing custom dialog boxes for these purposes. The common dialog control has no events, only methods and properties. Actually, like the Timer control, the common dialog box control is invisible. However, when you execute a statement of the form

```
CommonDialog1.Show_____
```

where the blank line is filled with Open, Save, Color, Font, or Printer, the specified dialog box is produced. Table 10.1 gives the purposes of the various dialog boxes.

TABLE 10.1
The Different Types of Dialog Boxes

Type of Dialog box	Purpose of Dialog Box
Open	Determine what file to open
Save As	Determine where and with what name to save a file
Color	Select any available color
Font	Select a font for the screen or printer
Print	Help control the printer

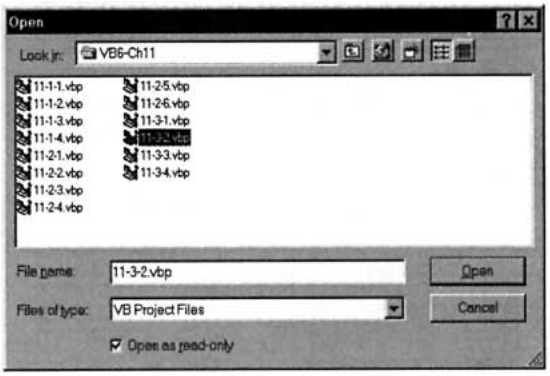


FIGURE 10-13 *An Open Common Dialog Box*

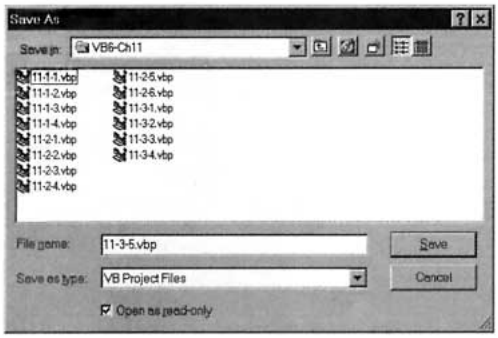


FIGURE 10-14 *A Save As Common Dialog Box*

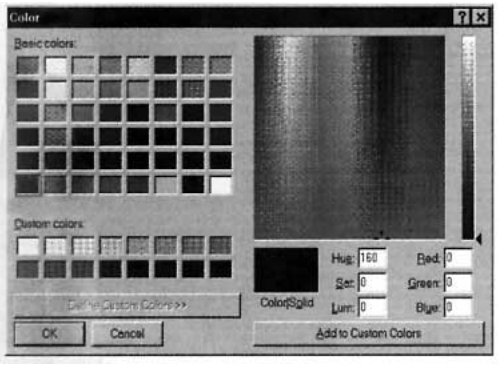


FIGURE 10-15 *A Color Common Dialog Box*

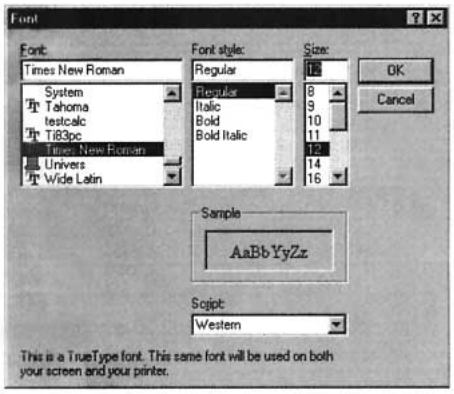


FIGURE 10-16 *A Font Common Dialog Box*

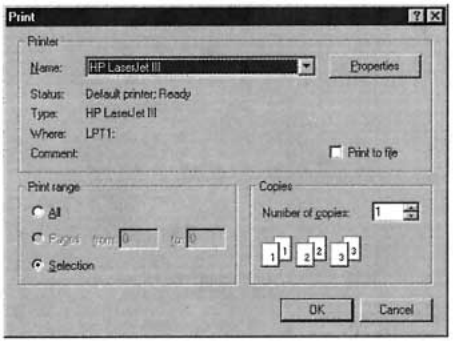


FIGURE 10-17 *A Print Common Dialog Box*

The Flags property influences certain features of the dialog box and should be set prior to setting the Show method. A complete discussion of the Flags property would be too great a digression. For our purposes, we will be well served by always setting the Flags property to 3 with the statement

```
dlgBox.Flags = 3
```

After selections are made from a common dialog box and the OK button is clicked, the values of the selections are stored in properties such as FileName, Color, FontName, FontItalic, FontSize and Copies. For instance, the following event procedure specifies the font for the contents of a text box.

```
Private Sub cmdButton_Click()
  dlgFont.Flags = 3
  dlgFont.ShowFont                'invoke Font common dialog box
  'Select Font, Font style, and Size and then click on OK
  txtBox.Font.Name = dlgFont.FontName
  txtBox.Font.Bold = dlgFont.FontBold
  txtBox.Font.Italic = dlgFont.FontItalic
  txtBox.Font.Size = dlgFont.FontSize
End Sub
```

Table 10.2 gives the principal properties whose setting are garnered from the common dialog boxes.

TABLE 10.2
Principal Properties of the Common Dialog Boxes

Type of Common Dialog Box	Principal Properties
Open	FileName
Save As	FileName
Color	Color
Font	FontName, FontSize, FontBold, FontItalic
Print	Copies, FromPage, ToPage

With Open and Save As common dialog boxes, a property is needed to specify what types of files should be displayed. A statement of the form

```
dlgFile.Filter = "dscrpt1|filter1|dscrpt2|filter2|dscrpt3|filter3"
```

provides verbal descriptions for the Type box and strings using wildcard characters (filters) to identify the files. A specific statement might be

```
dlgFile.Filter = "Text Files|*.TXT|FRM Files|*.FRM|All Files|*.*"
```

After the filter property is set, the FilterIndex property can be used to set the default filter. For instance, if the preceding statement is followed with

```
dlgFile.FilterIndex = 1
```

the default will be the first pair of filter items. That is, when the dialog box pops up, the Files of type box will display Text Files, and the large box will show only files with the extension TXT.

COMMENT

1. In the Properties window of a FlexGrid control, one setting of the GridLines property is "0 – flexGridNone." In code, this setting can be invoked with either

```
msgFlex.GridLines = 0
```

or

```
msgFlex.GridLines = flexGridNone
```

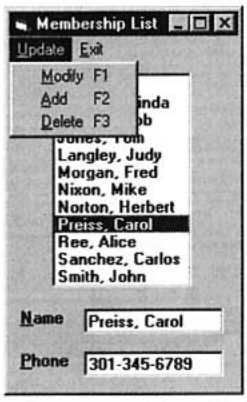
SUMMARY

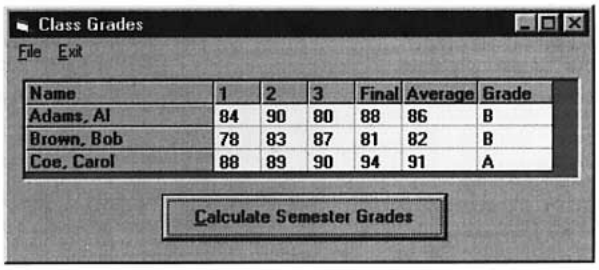
1. *List boxes* provide easy access to lists of strings. The lists can be automatically sorted (Sorted property = True), altered (AddItem, RemoveItem, and Clear methods), the currently highlighted item identified (Text property), and the number of items determined (ListCount property). The array List() holds the items stored in the list. Each item is identified by an index number (0, 1, 2, . . .). The most recently inserted item can be determined with the NewIndex property. The ItemData property associates a number with each item of text.
2. *Combo boxes* are enhanced text boxes. They not only allow the user to enter information by typing it into a text box (read with the Text property), but allow the user to select the information from a list of items.
3. *Drive*, *directory*, and *file list boxes* are specialized list boxes managed largely by Windows. The selected items are identified by the Drive, Path, and FileName properties, respectively. A directory list box always displays the subdirectories of the directory identified by its Path property, and a files list box displays the files in the directory identified by its Path property.
4. Selections are made with *check boxes* (allow several) and *option buttons* (allow at most one). The state of the control (*checked* vs. *unchecked* or *on* vs. *off*) is stored in the Value property. Clicking on a check box toggles its state. Clicking on an option button gives it the *on* state and turns *off* the other option buttons in its group.
5. Frames are used to group controls, especially option buttons, as a unit.
6. *Horizontal* and *vertical scroll bars* permit the user to select from among a range of numbers by clicking or dragging with the mouse. The range is specified by the Min and Max properties, and new settings trigger the Click and Change events.
7. The *timer control* triggers an event after a specified amount of time.
8. The *shape* and *line controls* enhance the visual look of a form with rectangles, ovals, circles, and lines of different size, thickness, and color.
9. The *image control*, which displays pictures or icons, can either expand to accommodate the size of the drawing or have the drawing alter its size to fit the image control.
10. A Microsoft FlexGrid control is a rectangular array of cells, each identified by a row and column number. The numbers of rows and columns are specified by the Rows and Cols properties. If the size of the grid is larger than provided by the control, scroll bars can be used to look at different parts of the grid. The FixedRows and FixedCols properties fix a certain number of the top rows and leftmost columns so that they will not scroll. The Row and Col properties are used to designate one cell as *current*. The Text property is used to read or place text into the current cell.
11. *Menus*, similar to the menus of Visual Basic itself, can be created with the Menu Design window.
12. The *clipboard* is filled with the SetText method or by pressing Ctrl+C, and is copied with the GetText function or with Ctrl+V.

13. *Additional forms* serve as new windows or dialog boxes. They are revealed with the Show method and concealed with the Hide method.
14. *Common dialog boxes* provide a standard way of specifying files, colors, and fonts, and of communicating with the printer.

PROGRAMMING PROJECTS

1. *Membership List.* Write a menu-driven program to manage a membership list. (See the following Membership List form.) Assume that the names and phone numbers of all members are stored in the sequential file MEMBERS.TXT. The names should be read into the list box when the form is loaded and the phone numbers should be read into an array. When a name is highlighted, both the name and phone number of the person should appear in the text boxes at the bottom of the screen. To delete a person, highlight his or her name and click on the Delete menu item. To change either the phone number or the spelling of the person's name, make the corrections in the text boxes and click on the menu item Modify. To add a new member, type his or her name and phone number into the text boxes and click on the menu item Add. When Exit is clicked, the new membership list should be written to a file and the program should terminate.





2. *Grade Book*. Write a comprehensive program that a professor could use to record student grades for several classes and save the records in sequential files. (See the preceding Class Grades form.) Each class has three hourly exams and a final exam. The file for a single class should consist of the number of students in the class, call it *n*, and a record of five fields (name, grade1, grade2, grade3, final) for each student, with the records ordered alphabetically by the student's name. (A typical name might appear as "Doe, John".) Initially, the four grade fields should contain zeros. The program should contain a top-level menu item, File, with second-level subitems for Open, Save, Add Student, Remove Student. When a file is opened (via a file list directory or common dialog box), the data for the students should be loaded into a grid of $n + 1$ rows and 7 columns. (The last two columns should remain blank.) The professor should be able to enter (or alter) exam data by clicking on the cell and responding to an input box. When a student is added, the grid should be enlarged by one row and the student inserted in proper alphabetical position. When a student is deleted, the grid should be reduced by one row. When the Calculate Semester Grades button is clicked, the last two columns should be filled in by the program. (Assume that the final exam counts as two hour exams.) If a grade is changed after the last

two columns have been filled in, the corresponding average and grade should be recomputed.

3. *Tic-Tac-Toe.* Write a program that "officiates" a game of tic-tac-toe. That is, the program should allow two players to alternate entering X's and O's into a tic-tac-toe board until either someone wins or a draw is reached. If one of the players wins, the program should announce the winner immediately; in case of a draw, the program should display "Cat's game". The players should enter their plays by clicking on the desired cell in the tic-tac-toe grid, and the program should check that each play is valid. ***Optional Enhancement:*** Allow the players to enter a number *n*. The program should officiate a best-of-*n* tournament, keeping track of the number of games won by each player until one of them wins more than half of the games. Ignore draws.

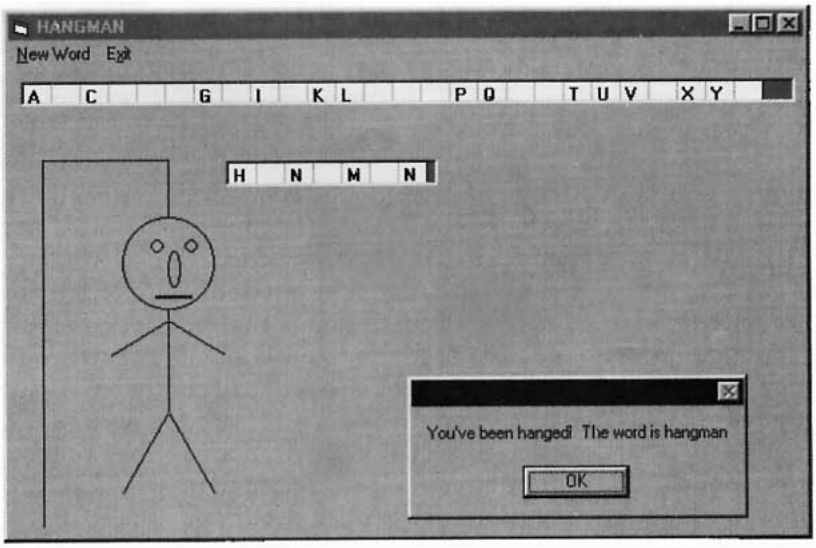
4. *Hangman.* Write a program to play Hangman. (See the following Hangman form.) A list of 20 words should be placed in a sequential file and one selected at random with Rnd. The program should do the following:

   (a) Draw a gallows on the screen with three line controls.

   (b) Create a grid having 1 row and 26 columns, and fill the grid with the 26 letters of the alphabet.

   (c) Create a second grid of one row and the number of columns equal to the length of the word selected.

   (d) Each time the user clicks on one of the letters of the alphabet, that letter should be removed. If the letter is in the selected word, its location(s) should be revealed in the second grid. If the letter is not in the word, another piece of the man should be drawn with a shape control.



5. *Inventory Control.* Write an inventory program for a book store and save the information in a sequential file. Each record should consist of five fields—title, author, category, wholesale price, and number in stock. (The two categories are fiction and nonfiction.) At any time, the program should display the titles of the books in stock in a list box, for which the user should have the option of displaying either all titles or just those in one of the two categories. When a book is selected from the list, its title, author, category, wholesale price, and number in stock should be displayed in a picture box. The user should be able to add a new book, delete a book, or change the inventory level of a book in stock. At any time, the user should be able to calculate the total value of all books, or the total value of the books in either category.

6. *Voting Machine.* The members of the local Gilligan's Island fan club bring a computer to their annual meeting to use in the election of a new president. Write a program to handle the election. The program should add each candidate to a list box as he or she is nominated. After the nomination process is complete, club members should be able to approach the computer one at a time and double-click on the candidate of their choice. When a "Tally Votes" command button is clicked, a second list box, showing the number of votes received by each candidate, should appear alongside the first list box. Also, the name(s) of the candidate(s) with the highest number of votes should be displayed in a picture box.
7. *Airplane Seating Chart*. An airplane has 30 rows (numbered 1 through 30), with 6 seats (labeled A, B, C, D, E, and F) in each row. Write a program to display a 7-by-31 grid with a cell for each seat. As each passenger selects a seat and a meal (regular, low-calorie, or vegetarian), the ticket agent clicks on the cell corresponding to the seat. A dialog box requests the type of meal and then one of the letters R, L, or V is placed in the cell clicked. At any time, the agent can request the number of seats filled, the number of window seats vacant, and the numbers of each type of meal ordered.

SECTION 11

DATABASE MANAGEMENT

11.1 AN INTRODUCTION TO DATABASES

The management of databases is the number one use of computers today. Airlines use databases to handle nearly 1.5 billion passenger reservations per year. The 6500 hospitals in the United States use databases to document the care of over 30 million patients per year. Banks in the United States use databases to monitor the use of 350 million credit cards. Although databases vary considerably in size and complexity, most of them adhere to the fundamental principles of design discussed in this chapter. That is, they are composed of a collection of interrelated tables.

A **table** is a rectangular array of data. Table 11.1 provides information about large cities. Each column of the table, called a **field**, contains the same type of information. (The third column gives the 1995 population in millions and the fourth column gives the projected 2015 population in millions.) The names of the fields are *city*, *country*, *pop1995*, and *pop2015*. Each row, called a **record**, contains the same type of information as every other row. Also, the pieces of information in each row are related; they all apply to a specific city. Table 11.2, Countries, has three fields and nine records.

TABLE 11.1
Cities

city	country	pop1995	pop2015
Beijing	China	12.4	19.4
Bombay	India	15.1	27.4
Calcutta	India	11.7	17.6
Los Angeles	USA	12.4	14.3
Mexico City	Mexico	15.6	18.8
New York	USA	16.3	17.6
Sao Paulo	Brazil	16.4	20.8
Shanghai	China	15.1	23.4
Tianjin	China	10.7	17.0
Tokyo	Japan	26.8	28.7

TABLE 11.2
Countries

Country	pop1995	currency
Brazil	155.8	real
China	1185.2	yuan
India	846.3	rupee
Indonesia	195.3	rupiah
Japan	125.0	yen
Mexico	85.6	peso
Nigeria	95.4	naira
Russia	148.2	ruble
USA	263.4	dollar

Source: An Urbanized World—Global Report on Human Settlements 1996, a report presented at Habitat II, a UN conference on the world's cities held in Istanbul in June 1996.

A **database** (or **relational database)** is a collection of one or more (usually related) tables that has been created with **database management software**. The best known dedicated database management products are Access, Btrieve, dBase, FoxPro, and Paradox. Every version of Visual Basic 6.0 can manage, revise, and analyze a database that has been created with one of these products. Section 11.3 shows how to create a database with Visual Data Manager, a miniversion of Access that is supplied with Visual Basic. Section 11.3 also gives a code template for creating a database programmatically.

The databases used in this chapter can be found in the collection of files accompanying this text. The database files have the extension .MDB. For instance, the file MEGACTY1.MDB is a database file containing the two tables presented on the preceding page. (***Note:*** MDB files should be copied from the CD onto a hard drive and accessed from the hard drive.)

The Data Control

Visual Basic communicates with databases through the data control. Data controls can read, modify, delete, and add records to databases. The following walkthrough uses a data control to connect Visual Basic to the database MEGACTY1.MDB.

A Data Control Walkthrough

1. Press Alt/File/New Project and double-click on Standard EXE.
2. Double-click on the data control icon. Set its Name property to datCities and its Caption property to Cities.
3. Stretch it horizontally to see the caption Cities.
4. Select the DatabaseName property and set it to the filespec for the file MEGACTY1.MDB.

   An Open File dialog box will help you locate the file.
5. Select the RecordSource property and click on the down-arrow button at the right of the Settings window.

   The names of the two tables in the database, Cities and Countries, are displayed.
6. Select Cities.
7. Place a text box, txtCity, on the form.

   Text boxes are said to be **data-aware** because they can be bound to a data control and access its data.
8. In the Properties window, select the DataSource property of txtCity.

9. Click on the down arrow to the right of the Settings box and select datCities.
10. Select the DataField property and click on the down arrow at the right of the Settings box.

    You will see the names of the different fields in the table.
11. Select the field *city*.

    The text box now is said to be **bound** to the data control. It can now display data from the *city* field of the Cities table.
12. Place another text box, txtPop1995, on the form.
13. Select txtPop1995's DataSource property.
14. Click on the down arrow to the right of the Settings box and select datCities.
15. Select the DataField property, click on the down arrow at the right of the Settings box, and select *pop1995*.
16. Run the program.

    The form will appear as in Figure 11-1. The arrows on the data control, called **navigation arrows**, look and act like VCR buttons. The arrows have been identified by the tasks they perform.
17. Click on the various navigation arrows on the data control to see the different cities and their populations in the Cities table displayed in the text boxes.
18. Change the name of a city or change its population and then move to another record.

    If you look back through the records, you will see that the data have been permanently changed.

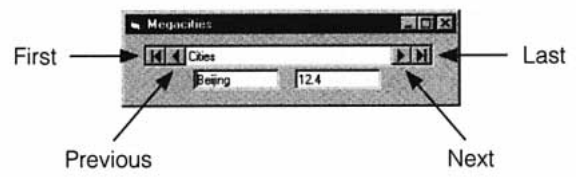


FIGURE 11-1 *A Data Control with Two Text Boxes Bound to It*

Using Code with a Data Control

Only one record can be accessed at a time; this record is called the **current record**. In this walk-through, the text boxes bound to the data control showed the contents of the *city* and *pop1995* fields of the current record. The user clicked on the navigation arrows of the data control to select a new current record.

Code can be used to designate another record as the current record. The methods MoveNext, MovePrevious, MoveLast, and MoveFirst select a new current record as suggested by their names. For instance, the statement

```
Data1.Recordset.MoveLast
```

specifies the last record of the table to be the current record. (The word Recordset is inserted in most data-control statements that manipulate records for reasons that needn't concern us now.)

The entry of the field *fieldName* of the current record is

```
Data1.Recordset.Fields("fieldName").Value
```

For instance, with the status as in Figure 11-1, the statement

```
strVar = datCities.Recordset.Fields("city").Value
```

assigns "Beijing" to the variable *strVar* and the statements

```
datCities.Recordset.Edit
datCities.Recordset.Fields("city").Value = "Peking"
datCities.Recordset.Update
```

change the *city* field of the current record to "Peking". (The first statement makes a copy of the current record for editing. The second statement alters the copy, and the third statement sends the new copy of the record to the database.)

The number of previously accessed records in the table is given by the RecordCount property. The EOF (End Of File) and BOF (Beginning Of File) run-time properties indicate whether the end or beginning of the file has been reached. For instance, the following two sets of statements each place the cities into a list box.

```
datCities.Recordset.MoveLast 'Needed to set value of RecordCount
  datCities.Recordset.MoveFirst
  For i = 1 to datCities.Recordset.RecordCount
    lstBox.AddItem datCities.Recordset.Fields("city").Value
    datCities.Recordset.MoveNext
Next i
```

```
datCities.Recordset.MoveFirst
```

```
Do
While Not datCities.Recordset.EOF
  lstBox.AddItem datCities.Recordset.Fields("city").Value
  datCities.Recordset.MoveNext
Loop
```

The current record can be marked for removal with the statement

```
Data1.Recordset.Delete
```

The record will be removed when a data control navigation arrow is clicked or a Move method is executed. A new record can be added to the end of the table with the statement

```
Data1.Recordset.AddNew
```

followed by

```
Data1.Recordset.Fields("fieldName").Value = entryForField
```

statements for each field and a

```
Data1.Recordset.Update
```

statement. Alternately, the AddNew method can be followed by the user typing the information into text boxes bound to the data control and then moving to another record. (***Note:*** When you add a record and then click on the MovePrevious arrow, you will not see the next-to-last record, but rather will see the record preceding the record that was current when AddNew was executed.)

The Validation Event

Visual Basic has a device called **validation** that lets you restrict the values a user can enter into a table. For instance, if the Cities table is only to contain cities with a population of more than 1 million, you can use validation to prevent any record with a number less than 1 in the *pop1995* field from being entered. Validation also allows you to catch (and prevent) errors that might cause a program to crash.

Data controls have an event procedure called Validate that is activated whenever the current record is about to be changed. For instance, it is called when a navigation arrow is

clicked or a Move, Update, Delete, or AddNew method is executed. The general form of the Validate event procedure is

```
Private Sub Data1_Validate(Action As Integer, Save As Integer)
   statement(s)
End Sub
```

The value of Action identifies the specific operation that triggered the event and the value of Save specifies whether data bound to the control has changed. You can change the value of the Action argument to perform a different action. Some values of the Action argument are shown in Table 11.3.

TABLE 11.3
Some Values of the Action Argument

Value	Description
1	MoveFirst method
2	MovePrevious method
3	MoveNext method
4	MoveLast method
5	AddNew method
6	Update operation (not UpdateRecord)
7	Delete method
10	Close method

If you assign 0 to the Action argument, the operation will be canceled when the Validate event procedure is exited.

The value of Save is –1 (True) if the data in any control attached to the data control have changed and is 0 (False) otherwise. If you set the value of Save to 0 in the Validate event procedure, any changes will not be saved.

Consider the form created in the walkthrough. Suppose the contents of txtPop1995, the 1995 population text box, is changed to .8 and then a navigator arrow is clicked in an attempt to move to another record. The following code prevents the move.

```
Private Sub datCities_Validate(Action As Integer, Save As Integer)
   Dim strMsg As String
   If val(txtPop1995) < 1 then
      strMsg = "We only allow cities having a population " & _ "at least one
               million."
      MsgBox strMsg"City too small!"
      Action = 0
   End If
End Sub
```

If the statement

```
Action = 0
```

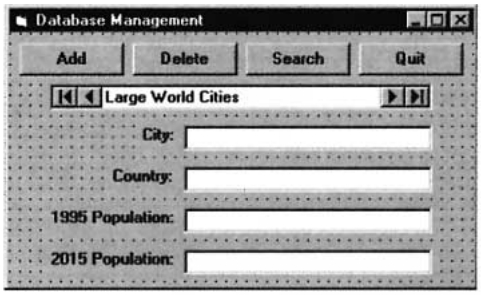
is changed to

```
Save = 0
```

the move will take place, but the previous record will retain its original values. That is, the number .8 will not appear in the table.

EXAMPLE 1

The following program is a general database manager for the Cities table in the MEGACTY1.MDB database. It allows the user to edit the Cities table as needed and to locate information based on the city name. (In the Validate event procedure, the inner If block keeps the message box from appearing when the first or last record is deleted.)

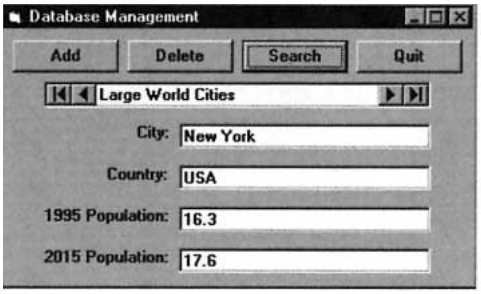


Object	Property	Setting
frmDBMan	Caption	Database Management
cmdAdd	Caption	Add
cmdDelete	Caption	Delete
cmdSearch	Caption	Search
cmdQuit	Caption	Quit
datCities	Caption	Large World Cities
	Database Name	MEGACTY1.MDB
	Record Source	Cities
lblCity	Caption	City:
txtCity	Text	(blank)
	Data Source	datCities
	DataField	City
lblCountry	Caption	Country:
txtCountry	Text	(blank)
	Data Source	datCities
	DataField	Country
lblPop1995	Caption	1995 Population:
txtPop1995	Text	(blank)
	Data Source	datCities
	DataField	pop1995
lblPop2015	Caption	2015 Population:
txtPop2015	Text	(blank)
	Data Source	datCities
	DataField	pop2015

```
Private Sub cmdAdd_Click()
   'Add a new record
   datCities.Recordset.AddNew
   'Data must be entered and a new record moved to
End Sub

Private Sub cmdDelete_Click ()
   'Delete the currently displayed record
   datCities.Recordset.Delete
   'Move so that user sees deleted record disappear
   datCities.Recordset.MoveNext
   If datCities.Recordset.EOF Then
      datCities.Recordset.MovePrevious
   End If
End Sub

Private Sub cmdSearch_Click()
   Dim strSearchFor As String, foundFlag As Boolean
   'Search for the city specified by the user
   strSearchFor = UCase(InputBox("Name of city to find:"))
   If Len(strSearchFor) > 0 Then
      datCities.Recordset.MoveFirst
      foundFlag = False
```

```
    Do While (Not foundFlag) And (Not datCities.Recordset.EOF)
       If UCase(datCities.Recordset.Fields("City").Value) = strSearchFor Then
            foundFlag = True
         Else
            datCities.Recordset.MoveNext
         End If
    Loop
    If Not foundFlag Then
         MsgBox "Unable to locate requested city.""Not Found"
         datCities.Recordset.MoveLast 'move so that EOF is no longer true
    End If
  Else
       MsgBox "Must enter a city.", ,""
  End If
End Sub

Private Sub cmdQuit_Click ()
  End
End Sub

Private Sub datCities_Validate(Action As Integer, Save As Integer)
  'Prevent a user from adding a city of population under 1 million
  Dim strMsg As String
  If Val(txtPop1995) < 1 Then
    If (Not datCities.Recordset.EOF) And (Not datCities.Recordset.BOF) Then
      strMsg = "We only allow cities having a population of " & _
               "at least one million."
      MsgBox strMsg"City too small!"
      Action = 0
    End If
  End If
End Sub
```

[Run, click the Search button, and enter New York.]



COMMENTS

1. App.Path cannot be used when you set the DatabaseName property of a data control in the Properties window at design time. However, App.Path can be used in the Form_Load event procedure. For instance, if you add the lines

```
Private Sub Form_Load()
  datCities.DatabaseName = App.Path & "\MEGACTY1.MDB"
End Sub
```

to Example 1, the program will find the database when it is located in the same folder as the program. The assignment made in the Form_Load event procedure will override any setting made in the Properties window. (The setting in the Properties window even can be left blank.) The programs for Section 11 on the CD accompanying this textbook contain the above Form_Load event procedure.

2. You will most likely alter the file MEGACTY1.MDB while experimenting with the data control or running the program in Example 1. You can always obtain a fresh copy of MEGACTY1.MDB by recopying it from the CD.
3. You can prevent the user from altering the data in a table by setting the Read-Only property of its data control to True.
4. The following controls can be bound to a data control: text box, check box, image, label, picture box, list box, combo box, data bound list box, data bound combo box, and FlexGrid.
5. A form can contain more than one data control.
6. Some entries in a table can be empty. For instance, in the Cities table, if the 2015 projected value is not known for a particular city, it can be omitted.
7. Do not use a method such as Move, Delete, or AddNew inside the Validate event procedure. Otherwise, an infinite loop will occur.
8. Field names can be up to 64 characters in length and can consist of letters, numbers, and spaces. If spaces are used in a name, then the name must be enclosed in brackets when used in Visual Basic.
9. Both tables in the database MEGACTY1.MDB have fields called *country*. If there is ever any question about which is being referred to, we can distinguish them by using the two (full) names Cities.country and Countries. country.
10. In the MEGACTY1.MDB database, the values in the field *city* are all of data type String and the values in the field *pop1995* are all of data type Single. We say that field *city* has type String (also known as Text) and the field *pop1995* has type Single. Two other common data types are Date/Time and Boolean (also known as Yes/No).
11. When a field is first created, a type must be specified. When that type is String (or Text), a maximum length must also be specified. In the MEGACTY1.MDB database, the fields *city* and *country* have maximum length 20 and the field *currency* has maximum length 10.
12. The database MEGACTY1.MDB was created with Visual Data Manager, which has the same format as Access. When the database to be used has been created with other software, such as FoxPro 3.0 or dBase 5.0, then the walkthrough requires an additional step. Namely, between Steps 3 and 4, the Connect property of the data control has to be set to the name of the software product. (This step was not necessary in our case because Access is the default software.) ***Note:*** Access database file names end with .MDB, which is an abbreviation for Microsoft Data Base. Btrieve, FoxPro, dBase, and Paradox database file names end with .DAT, .DBF, .DBF, and .DB, respectively.

11.2 RELATIONAL DATABASES AND SQL

■ Primary and Foreign Keys

A well-designed table should have a field (or set of fields) that can be used to uniquely identify each record. Such a field (or set of fields) is called a **primary key**. For instance, in the Countries table of Section 11.1, the *country* field is a primary key. In the Cities table, because we are only considering very large cities (of over 1 million population), the *city* field is a pri-

mary key. Databases of student enrollments in a college usually use a field of social security numbers as the primary key. Names would not be a good choice because there could easily be two students having the same name.

When a database is created, a field can be specified as a primary key. If so, Visual Basic will insist that every record have an entry in the primary key field and that the same entry does not appear in two different records. If the user tries to enter a record with no data in the primary key, the error message "Index or primary key can't contain a null record." will be generated. If the user tries to enter a record with the same primary key data as another record, the error message "Duplicate value in index, primary key, or relationship. Changes were unsuccessful." will be displayed.

When a database contains two or more tables, the tables are usually related. For instance, the two tables Cities and Countries are related by their *country* field. Let's refer to these two fields as Cities.country and Countries.country. Notice that every entry in Cities.country appears uniquely in Countries.country and Countries.country is a primary key. We say that Cities.country is a **foreign key** of Countries.country. Foreign keys can be specified when a table is first created. If so, Visual Basic will insist on the **Rule of Referential Integrity**, namely, that each value in the foreign key must also appear in the primary key of the other table.

The CD accompanying this book contains a database named MEGACTY2.MDB. It has the same information as MEGACTY1.MDB except that Cities.city and Countries.country have been specified as primary keys for their respective tables, and Cities.country has been specified as a foreign key of Countries.country. If the user tries to add a city to the Cities table whose country does not appear in the Countries table, then the error message "Can't add or change record. Referential integrity rules require a related record in table 'Countries'." will be displayed. The message will also be generated if the user tries to delete a country from the Countries.country field that appears in the Cities.country field. Due to the interdependence of the two tables in MEGACTY2.MDB, this database is called a **relational database**.

A foreign key allows Visual Basic to link (or **join**) together two tables from a relational database in a meaningful way. For instance, when the two tables Cities and Countries from MEGACTY2.MDB are joined based on the foreign key Cities.country, the result is Table 11.4. The record for each city is expanded to show its country's population and its currency. This joined table is very handy if, say, we wanted to click on navigation arrows and display a city's name and currency. We only have to create the original two tables; Visual Basic creates the joined table as needed. The request for a joined table is made in a language called SQL.

Table 11.4
A Join of Two Tables

city	Cities. country	Cities. pop1995	Countries. pop2015	Country. country	pop1995	currency
Tokyo	Japan	26.8	28.7	Japan	125.0	yen
Sao Paulo	Brazil	16.4	20.8	Brazil	155.8	real
New York	USA	16.3	17.6	USA	263.4	dollar
Mexico City	Mexico	15.6	18.8	Mexico	85.6	peso
Bombay	India	15.1	27.4	India	846.3	rupee
Shanghai	China	15.1	23.4	China	1185.2	yuan
Los Angeles	USA	12.4	14.3	USA	263.4	dollar
Beijing	China	12.4	19.4	China	1185.2	yuan
Calcutta	India	11.7	17.6	India	846.3	rupee
Tianjin	China	10.7	17.0	China	1185.2	yuan

SQL

Structured Query Language (SQL) was developed in the early 1970s at IBM for use with relational databases. The language was standardized in 1986 by ANSI (American National

Standards Institute). Visual Basic uses a version of SQL that is compliant with ANSI-89 SQL. There are some minor variations that are of no concern in this book.

SQL is a very powerful language. One use of SQL is to request specialized information from an existing database and/or to have the information presented in a specified order.

Four SQL Requests

We will focus on four basic types of requests that can be made with SQL.

Request I: Show the records of a table in a specified order.
Some examples of orders with MEGACTY2.MDB are

(a) Alphabetical order based on the name of the city.

(b) Alphabetical order based on the name of the country, and within each country group, the name of the city.

(c) In descending order based on the projected 2015 population.

Request II: Show just the records that meet certain criteria.
Some examples of criteria with MEGACTY2.MDB are

(a) Cities that are in China.

(b) Cities whose 2015 population is projected to be at least 20 million.

(c) Cities whose name begins with the letter S.

Request III: Join the tables together, connected by a foreign key, and present the records as in Requests I and II.
Some examples with MEGACTY2.MDB are

(a) Show the cities in descending order of the populations of their countries.

(b) Show the cities whose currency has "u" as its second letter.

Request IV: Make available just *some* of the fields of either the basic tables or the joined table. (For now, this type of request just conserves space and effort by Visual Basic. However, it will be very useful in Section 11.3 when used with a FlexGrid control.)

Some examples with MEGACTY2.MDB are

(a) Make available just the city and country fields of the table Cities.

(b) Make available just the city and currency fields of the joined table.

Normally, we set the RecordSource property of a data control to an entire table. Also, the records of the table are normally presented in the order they are physically stored in the table. We make the requests discussed above by specifying the RecordSource property as one of the following kinds of settings.

Request I:	SELECT * FROM *Table1* ORDER BY *field1* ASC or SELECT * FROM *Table1* ORDER BY *field1* DESC
Request II:	SELECT * FROM *Table1* WHERE *criteria*
Request III:	SELECT * FROM *Table1* INNER JOIN *Table2* ON *foreign field* = *primary field* WHERE *criteria*
Request IV:	SELECT *field1, field2, . . . fieldN* FROM *Table1* WHERE *criteria*

The words ASC and DESC specify ASCending and DESCending orders, respectively. A *criteria* clause is a string containing a condition of the type used with If blocks. In addition to the standard operators <, >, and =, *criteria* strings frequently contain the operator Like. Essentially, Like uses the wildcard characters ? and * to compare a string to a pattern. A

question mark stands for a single character in the same position as the question mark. For instance, the pattern "B?d" is matched by "Bid", "Bud", and "Bad". An asterisk stands for any number of characters in the same position as the asterisk. For instance, the pattern "C*r" is matched by "Computer", "Chair", and "Car". See Comments 3 through 5 for further information about Like.

In the sentence

```
SELECT fields FROM clause
```

fields is either * (to indicate all fields) or a sequence of the fields to be available (separated by commas), and *clause* is either a single table or a join of two tables. A join of two tables is indicated by a *clause* of the form

```
table1 INNER JOIN table2 ON foreign key of table1=primary key of table2
```

Appending

```
WHERE criteria
```

to the end of the sentence restricts the records to those satisfying *criteria*. Appending

```
ORDER BY field(s) ASC (or DESC)
```

presents the records ordered by the specified *field* or *fields*.

In general, the SQL statements we consider will look like

```
SELECT www FROM xxx WHERE yyy ORDER BY zzz
```

where SELECT *www* FROM *xxx* is always present and accompanied by one or both of WHERE *yyy* and ORDER BY *zzz*. In addition, the *xxx* portion might contain an INNER JOIN phrase.

The settings for the examples mentioned earlier are as follows:

I (a) Show the records from Cities in alphabetical order based on the name of the city.

```
SELECT * FROM Cities ORDER BY city ASC
```

I (b) Show the records from Cities in alphabetical order based first on the name of the country and,, within each country group,, the name of the city.

```
SELECT * FROM Cities ORDER BY country,, city ASC
```

I (c) Show the records from Cities in descending order based on the projected 2015 population.

```
SELECT * FROM Cities ORDER BY pop2015 DESC
```

II (a) Show the records for the Cities in China.

```
SELECT * FROM Cities WHERE country = 'China'
```

II (b) Show the records from Cities whose 2015 population is projected to be at least 20 million.

```
SELECT * FROM Cities WHERE pop2015 >= 20
```

II (c) Show the records from Cities whose name begins with the letter S.

```
SELECT * FROM Cities WHERE city Like 'S*'
```

III (a) Show the records from the joined table in descending order of the populations of their countries.

```
SELECT * FROM Cities INNER JOIN Countries ON Cities.country =
Countries.country ORDER BY Countries.pop1995 DESC
```

III (b) Show the records from the joined table whose currency has "u" as its second letter.

```
SELECT * FROM Cities INNER JOIN Countries ON Cities.country =
Countries.country WHERE currency Like '?u*'
```

IV (a) Make available just the city and country fields of the table Cities.

```
SELECT city, country FROM Cities
```

IV (b) Make available just the city and currency fields of the joined table.

```
SELECT city,, currency FROM Cities INNER JOIN Countries ON Cities.coun-
try = Countries.country
```

Note: In several of the statements, the single quote, rather than the normal double quote was used to surround strings. This is standard practice with SQL statements.

We can think of an SQL statement as creating in essence a new "virtual" table from existing tables. For instance, we might regard the statement

```
SELECT city, pop2015 FROM Cities WHERE pop2015 >= 20
```

as creating the "virtual" table

city	pop2015
Tokyo	28.7
Sao Paulo	20.8
Bombay	27.4
Shanghai	23.4

This table is a subtable of the original table Cities, that is, it consists of what is left after certain columns and rows are deleted.

As another example, the statement

```
SELECT Cities.city, Cities.Country, Country.currency FROM Cities INNER JOIN
Countries ON Cities.country = Countries.country WHERE Countries.country>'K'
```

creates in essence the "virtual" table

Cities.city	Cities.country	currency
New York	USA	dollar
Mexico City	Mexico	peso
Los Angeles	USA	dollar

which is a subtable of a join of the two tables Cities and Countries.

These "virtual" tables don't really exist physically. However, for all practical purposes, Visual Basic acts as if they did. In Visual Basic terminology, a "virtual" table is called a **recordset** and SQL statements are said to create a recordset. In standard relational database books, a "virtual" table is called a **view**.

SQL also can be used in code with a statement of the form

```
Data1.RecordSource = " SELECT ... FROM ..."
```

to alter the order and kinds of records presented from a database. However, such a statement must be followed by the statement

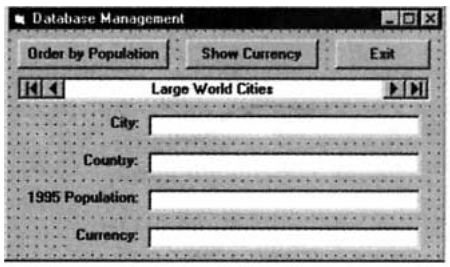
```
Data1.Refresh
```

to reset the information processed by the data control.

EXAMPLE 1

The following program allows the user to alter the order and kinds of information displayed from a database. When the first command button is pressed, the cities are presented in ascending order based on their

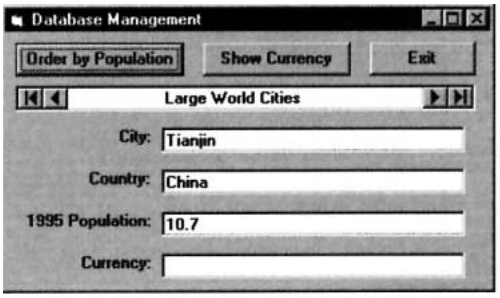
1995 populations. When the second command button is pressed, the cities are presented in alphabetical order along with their currencies.



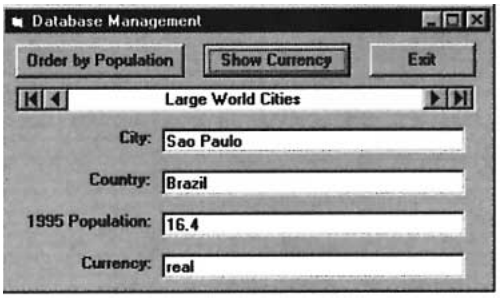
Object	Property	Setting
frmDBMan	Caption	Database Management
cmdOrder ByPop	Caption	Order by Population
cmdShow Currency	Caption	Show Currency
cmdQuit	Caption	Exit
datCities	Caption	Large World Cities
	Database Name	MEGACTY2.MDB
	Record Source	Cities
lblCity	Caption	City:
txtCity	Data Source	datCities
	DataField	city
	Text	(blank)
lblCountry	Caption	Country:
txtCountry	Data Source	datCities
	DataField	country
	Text	(blank)
lblPopulation	Caption	1995 Population:
txtPopulation	Data Source	datCities
	DataField	pop1995
	Text	(blank)
lblCurrency	Caption	Currency:
txtCurrency	Data Source	datCities
	Text	(blank)

```
Private Sub cmdOrderByPop_Click()
   Dim strSQL As String
   txtCurrency.DataField = ""
   txtCurrency.Text = ""
   strSQL = "SELECT * FROM Cities ORDER BY pop1995 ASC"
   datCities.RecordSource = strSQL
   datCities.Refresh
End Sub

Private Sub cmdQuit_Click()
   End
End Sub

Private Sub cmdShowCurrency_Click()

   Dim strSQL As String
   strSQL = "SELECT city, Cities.country, Cities.pop1995, currency " & _ "FROM
            Cities INNER JOIN Countries " & _ "ON
            Cities.country=Countries.country " & "ORDER BY city ASC"
   datCities.RecordSource = strSQL
   datCities.Refresh txtCurrency.DataField = "currency"
End Sub
```

[Run, and click on Order by Population.]



[Click on Show Currency, and then click on the Next navigator arrow six times.]



The program in Example 1 of Section 11.1 searched a table for a specific record by looping through all the records. Whereas this technique is fine for small tables, it is not efficient for searches of large tables. Visual Basic provides a better way with Find methods.

FIND METHODS

Suppose a table has been attached to the data control Data1, and an SQL statement has been used to create and order a recordset. Then a statement of the form

Data1.RecordSet.FindFirst *criteria*

starts at the beginning of the recordset, searches for the first record in the recordset that satisfies the criteria, and makes that record the current record. (Here, *criteria* is a string just like a *criteria* phrase that follows WHERE in an SQL statement.) The related methods FindLast, FindNext, and FindPrevious function as their names suggest. (FindLast starts at the end of the recordset. FindNext and FindPrevious start at the current record.) For instance, suppose an SQL statement ordered the cities alphabetically by name. The following statements and their outcomes show the effects of the various Find methods. (In each case, assume that the current record is Mexico City.)

Statement	New Current Record
datCities.Recordset.FindFirst "pop2015 < 20"	Beijing
datCities.Recordset.FindLast "pop2015 < 20"	Tianjin
datCities.Recordset.FindNext "pop2015 < 20"	New York
datCities.Recordset.FindPrevious "pop2015 < 20"	Los Angeles

Visual Basic has two properties, NoMatch and Bookmark, that help when a Find method fails to locate a suitable record.

If BkMk is a string variable, a statement of the form

```
BkMk = Data1.Recordset.Bookmark
```

assigns the location of the current record to the variable BkMk. When desired, the statement

```
Data1.Recordset.Bookmark = BkMk
```

will return to the original location in the table.

If a Find method does not locate a record matching the criteria, the first or last record (depending on where the search is heading) in the recordset becomes the current record and the NoMatch property is set to True. Therefore, the following lines of code can be used to keep the current record current whenever the Find method is unsuccessful.

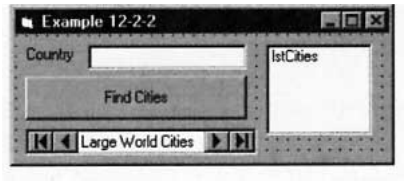
```
BkMk = Data1.Recordset.Bookmark
   Data1.Recordset.FindNext criteria
   If Data1.Recordset.NoMatch = True Then
   Data1.Recordset.Bookmark = BkMk
End If
```

EXAMPLE 2

The following program displays the large cities in a country specified by the user. Due to the SQL statement in the setting for datCities.RecordSource, the cities will be presented alphabetically. Notice the handling of the string variable *criteria*. Had the Find statement been

```
datCities.Recordset.FindFirst "country = nom"
```

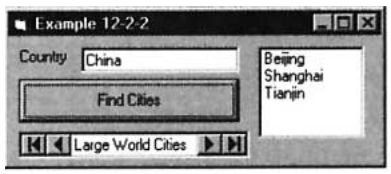
the error message "Can't find name 'nom'." would have been generated.



Object	Property	Setting
frmDBMan	Caption	EXAMPLE 12-2-2
lstCities		
cmdFind	Caption	Find Cities
lblCountry	Caption	Country
txtCountry	Caption	(blank)
datCities	Caption	Large World Cities
	Database Name	MEGACTY2.MDB
	Record Source	SELECT * FROM Cities ORDER BY city ASC

```
Private Sub cmdFind_Click()
   Dim nom As String, criteria As String
   lstCities.Clear
   If txtCountry.Text<> "" Then
        nom = txtCountry.Text
        criteria = "country = " & "'" & nom & "'"
        datCities.Recordset.FindFirst criteria
        Do While datCities.Recordset.NoMatch = False
           lstCities.AddItem datCities.Recordset.Fields("city").Value
           datCities.Recordset.FindNext criteria
        Loop
   If lstCities.ListCount = 0 Then lstCities.AddItem "None"
     Else
        MsgBox "You must enter a country.", , ""
        txtCountry.SetFocus
   End If
End Sub
```

[Run, type China into the text box, and press the command button.]



COMMENTS

1. Each record of the Countries table is related to one or more records of the Cities table, but each record of the Cities table is related to only one record of the Countries table. Therefore, we say that there is a **one-to-many relationship** from the Countries table to the Cities table.
2. SQL statements are insensitive to case. For instance, the following choices for *criteria* have the same effect: City='China', city='china', CITY='china', CiTy='CHINA'.
3. When the Like operator is used, the "pattern" must appear on the right of the operator. For instance, the SQL statement

   ```
   SELECT * FROM Cities WHERE city Like 'S*'
   ```

   cannot be replaced by

   ```
   SELECT * FROM Cities WHERE 'S*' Like city
   ```

4. The operator Like permits a refinement of the wildcard character "?". Whereas "?" is a placeholder for any letter, an expression such as "[*letter1*-*letter2*]" is a placeholder for any letter from *letter1* to *letter2*. For instance, the pattern "[A-F]ad" is matched by Bad and Dad, but not Sad.
5. The Like operator can be used in If blocks in much the same way as the operators >, =, and <. In this situation, the operator is case-sensitive. For instance, the condition ("bad" Like "[A-F]ad") is False. However, when Like is used in SQL statements, it is case-insensitive. That is, ("bad" Like "[A-F]ad") is True.
6. Sometimes a pair of fields is specified as a primary key. For instance, in a table of college courses, a single course might have several sections—a course might be identified as CMSC 102, Section 3. In this case, the pair of fields *course*, *section* would serve as a primary key for the table.
7. The requirement that no record may have a null primary key is called the **Rule of Entity Integrity**.
8. If there is no field with unique entries, database designers usually add a "counter field" containing the numbers 1, 2, 3, and so on. This field then can serve as a primary key.

11.3 THREE ADDITIONAL DATA-BOUND CONTROLS; CREATING AND DESIGNING DATABASES

So far, we have used text boxes to display the contents of a single field in a single row. The FlexGrid control can display an entire table (or recordset). Data-bound list boxes and data-bound combo boxes can display the contents of a single field for an entire table or recordset.

There are three ways you can create a database:

1. Use Visual Data Manager, a program supplied with all editions of Visual Basic.
2. Use code.
3. Use database management software, such as Access, Btrieve, FoxPro, or Paradox.

In this section we give a detailed explanation of how to use Visual Data Manager and include a code template that you can modify to create a database programmatically.

Before you can use the three additional data-bound controls, you must add them to the toolbar using the Components dialog box that is invoked from the Project menu. (Place an x in the check box next to "Microsoft FlexGrid Control 6.0" for the FlexGrid control, and

place an x in the check box next to "Microsoft Data Bound List Controls 6.0" for the other two controls.)

Using the FlexGrid Control

When you attach the FlexGrid control to a table via a data control, the FlexGrid control will display the entire contents of the table, including the field names. If the table is too large to be displayed in entirety, scroll bars will automatically appear to allow you to scroll to other parts of the table. The FlexGrid control is discussed in Section 10.3. The gray fixed row at the top will show the field names. You should set the FixedCols property to 0 to avoid having a blank fixed column at the left. If some of the fields have long entries, you should widen the corresponding columns with the ColWidth method in Form_Load. Although you can use code to alter the contents of the cells in a FlexGrid (as shown in Section 10.3), changes in the cells of the FlexGrid will not alter the contents of the database.

You can use an SQL statement to specify the fields displayed in a FlexGrid control. For instance, if the control has datCities as its DataSource and the DatabaseName setting for datCities is MEGACTY2.MDB, then the statement

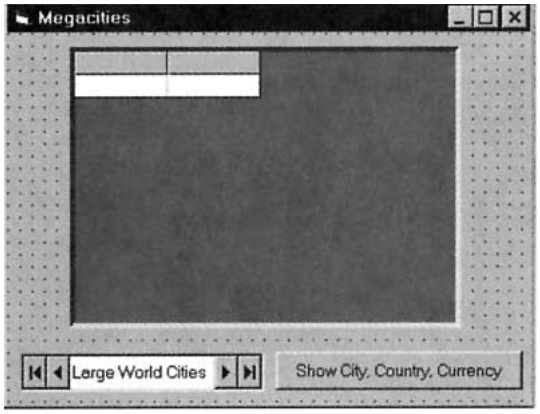
```
datCities.RecordSource = "SELECT city, country FROM Cities"
```

causes the FlexGrid to display only the first two columns of the Cities table. The same effect can be achieved at design time by setting the RecordSource property of datCities to

```
SELECT city, country FROM Cities
```

EXAMPLE 1

The following program displays the contents of the Cities table of the MEGACTY2.MDB database. When you click on the command button, the FlexGrid displays the cities, their countries, and currency. Also, the caption of the command button changes to "Show City, Country, Populations". The next time you click the command button, the contents of the FlexGrid returns to its original state.

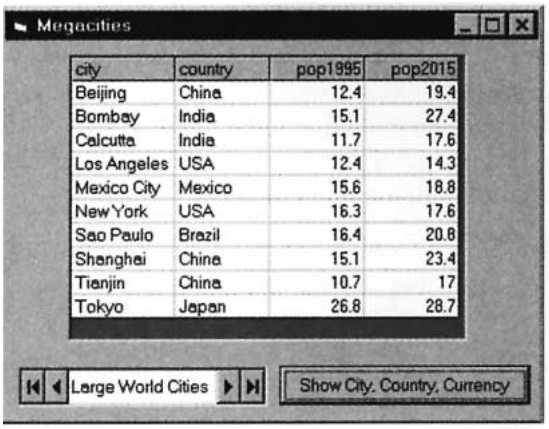


Object	Property	Setting
frm12_3_1	Caption	Megacities
datCities	Caption	Large World Cities
	Database Name	MEGACTY2.MDB
	Record Source	Cities
msgCities	Data Source	datCities
	FixedCols	0
cmdShow	Caption	Show City, Country, Currency

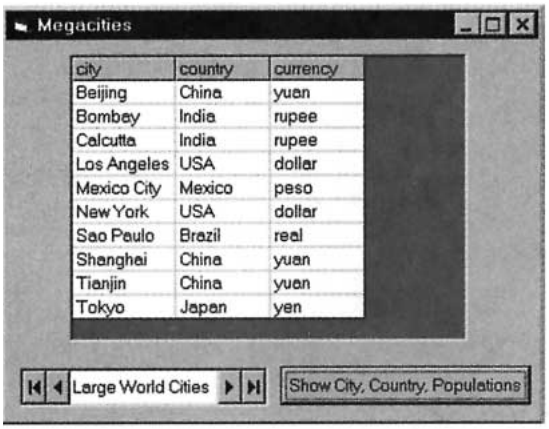
```
Private Sub cmdShow_Click()
If cmdShow.Caption = "Show City, Country, Currency" Then '
     Join the two tables and display cities, countries, and currency
     datCities.RecordSource = "SELECT city, Cities.country, currency FROM " & _
        "Cities INNER JOIN Countries ON Countries.country = Cities.country " & _
        "ORDER BY city"
     datCities.Refresh
     cmdShow.Caption = "Show City, Country, Populations"
  Else
     datCities.RecordSource = "Cities"
     datCities.Refresh
     cmdShow.Caption = "Show City, Country, Currency"
```

```
  End If
End Sub

Private Sub Form_Load()
  msgCities.ColWidth(0) = 1000 'Widen the first column slightly
End Sub
```

[Run]



[Click on the command button.]



■ Using the Data-Bound List Box and Combo Box Controls

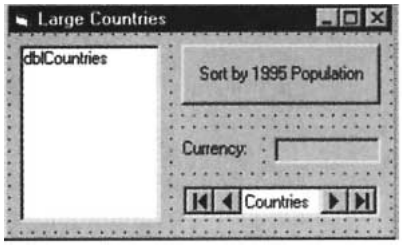
The data-bound list box and data-bound combo box controls look like the standard list box and combo box controls. They are often used with data-entry forms to speed data entry and ensure that valid data is entered. These controls automatically fill with a column from a table or recordset after you set a few properties. Note that the data-bound controls display data from a table or a recordset. Therefore, three methods (AddItem, Clear, and RemoveItem) and two properties (ListCount and Sorted) used with the regular list box and combo box are not available with the data-bound controls. The count of items displayed in a data-bound control is determined by the RecordCount property of the recordset.

Two key properties determine the entries of a data-bound list or combo box—**RowSource** and **ListField**. RowSource specifies a data control, and ListField specifies a field from the data control's table or recordset. That field will be used to fill the list.

EXAMPLE 2

The data-bound list box in the following example displays the countries in the Countries table of MEGACTY2.MDB. When the command button is clicked, SQL is used to sort the countries by their 1995

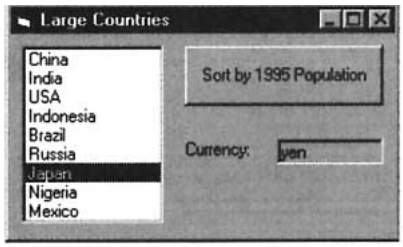
population (in descending order). When an item in the list box is double-clicked, the list box's Text property is used to display the name of the country's unit of currency in a label. (Recall that the value of the Text property is the currently highlighted item.) The data control is made invisible since it is not needed by the user.



Object	Property	Setting
frmCountries	Caption	Large Countries
datCountries	Caption	Countries
	Database Name	MEGACTY2.MDB
	Record Source	Countries
dblCountries	RowSource	datCountries
	ListField	Country
cmdSort	Caption	Sort by 1995 Population
lblCurrency	Caption	Currency:
lblUnit	Caption	(blank)
	BorderStyle	1- Fixed Single

```
Private Sub CmdSort_Click()
   datCountries.RecordSource = "SELECT * FROM Countries " & _ "ORDER by pop1995
                               DESC"
   datCountries.Refresh
End Sub

Private Sub dblCountries_dblClick()
   datCountries.Recordset.FindFirst _
                          "Country =" & "'" & dblCountries.Text & "'"
   lblUnit.Caption = datCountries.Recordset.Fields("currency").Value
End Sub
```

[Run, click on the command button, and then double-click on Japan.]

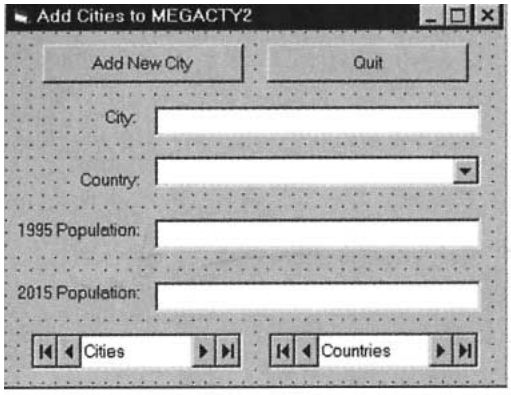


When used to enhance data entry, a data-bound list or combo control is usually linked to two data controls. The first data control is used to fill the list and the second to update a field in a table. As demonstrated in Example 2, the first data control fills the list as designated by the data-bound control's ListField and RowSource properties. The second control updates a field in a table specified by the data-bound control's DataSource and DataField properties. Another property of the data-bound control, the **BoundColumn** property, specifies the name of a field in the first data control's recordset. Once the user chooses one of the items in the list (and thereby one of the rows), the value in the row's specified field is passed to the field to be updated. Usually, the BoundColumn property has the same setting as the DataField property.

EXAMPLE 3

The following program adds cities to the Cities table of MEGACTY2.MDB. Of course, only cities in a country found in the Countries table are acceptable. A data-bound combo box is employed to show the user a list of the acceptable countries. (The experienced data-entry person can just type a name directly into the text box portion of the combo box, whereas the novice must refer to the list.) When the program

is first run, the user should press the command button to clear the text boxes. After that, for each new city to be entered into the database, the user should fill the text boxes and then press the command button.



Object	Property	Setting
frm12_3_3	Caption	Add Cities to MEGACTY2
datCountries	Caption	Countries
	DatabaseName	MEGACTY2.MDB
	RecordSource	Countries
Visible	0 – False	
datCities	Caption	Cities
	DatabaseName	MEGACTY2.MDB
	RecordSource	Cities
	Visible	0 – False
cmdAddNew	Caption	Add New City
cmdQuit	Caption	Quit
lblCity	Caption	City:
txtCity	DataSource	datCities
	DataField	city
lblCountry	Caption	Country:
dbcCountry	DataSource	datCities
	DataField	country
	BoundColumn	country
	RowSource	datCountries
	ListField	country
lblPop1995	Caption	1995 Population:
txtPop1995	DataSource	datCities
	DataField	pop1995
lblPop2015	Caption	2015 Population:
txtPop2015	DataSource	datCities
	DataField	pop2015

```
Private Sub cmdAddNew_Click()
  datCities.Recordset.AddNew
End Sub

Private Sub cmdQuit_Click()
  End
End Sub
```

[Run, click on the Add New City command button, add the data for a couple of cities, and click on Quit. Then place the cities table of MEGACTY2.MDB in a FlexGrid to confirm that the cities have indeed been added.]

Some possible cities to add are:

City	country	pop1995	pop2015
Jakarta	Indonesia	11.5	21.2
Osaka	Japan	10.6	10.6
Lagos	Nigeria	10.3	24.4

Creating a Database with Visual Data Manager

You invoke Visual Data Manager (VisData) from Visual Basic by pressing Alt/Add-Ins/Visual Data Manager. The first two entries of the File menu of VisData are Open Database, used to view and alter an existing database, and New, used to create a new database. See Figure 11-3.

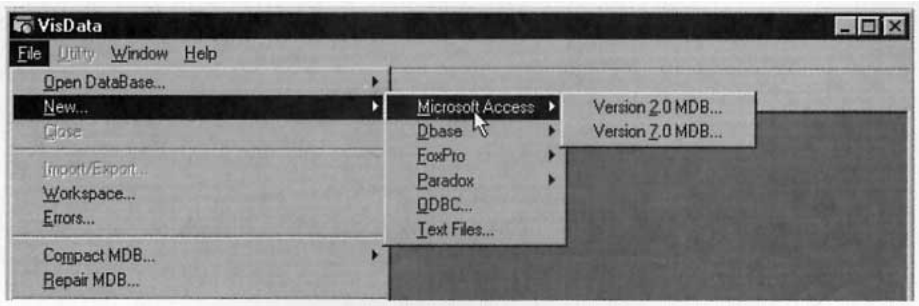


FIGURE 11-2 *Visual Data Manager's File Menu*

Let's focus on creating a new database. After you select New, you are presented with a drop-down menu used to choose the type of database as shown in Figure 11-2. Choose Microsoft Access and then specify a version. (Version 7.0 is the latest version of Access, the one that comes with the Professional Edition of Office 97.) Then a standard file-naming dialog box titled "Select Microsoft Access Database to Create" appears. See Figure 11-3.

FIGURE 11-3 *Dialog Box Used to Name the Database*

After you name the database, say as MEGACITY.MDB, and click Save, the Database window and SQL Statement box appear. We will work solely with the Database window.

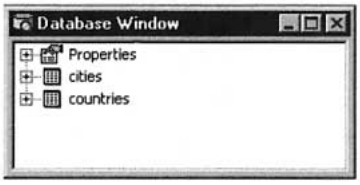


FIGURE 11-4 *Database Window. (Appears After the Database is Named. Initially the Window Will Contain Only the Properties Line. Additional Lines Will Appear as Tables are Created.)*

Suppose you want to create a database with two tables. Starting from the Database window, the basic steps are as follows:

1. Create the database and the two tables.
   (a) Click on the right mouse button within the Database window. Click on New Table and use the Table Structure window (Figure 11-5) to name the first table and to specify the fields and their data types. The steps listed after Figure 11-5 show how to carry out these tasks.
   (b) Repeat Step 1 for the second table.

2. (Optional) Specify a primary key for a table.

   (a)Highlight the table name in the Database window.

   (b) Press the right mouse button and choose Design to invoke the Table Structure window (Figure 11-5).

   (c) Press the Add Index button to invoke the Add Index window (Figure 11-6) and follow the steps listed after the figure.

3. Place records into a table.

   (***Note:*** The VisData toolbar contains three sets of icons. This discussion assumes that the left icon of each of the first two sets has been selected. These are the "Table type Recordset" and "Use Data Control on New Form" icons.)

   (a) Double-click on the table in the Database window to invoke the Table window (Figure 11-7).

   (b) Follow the directions listed after Figure 11-7.

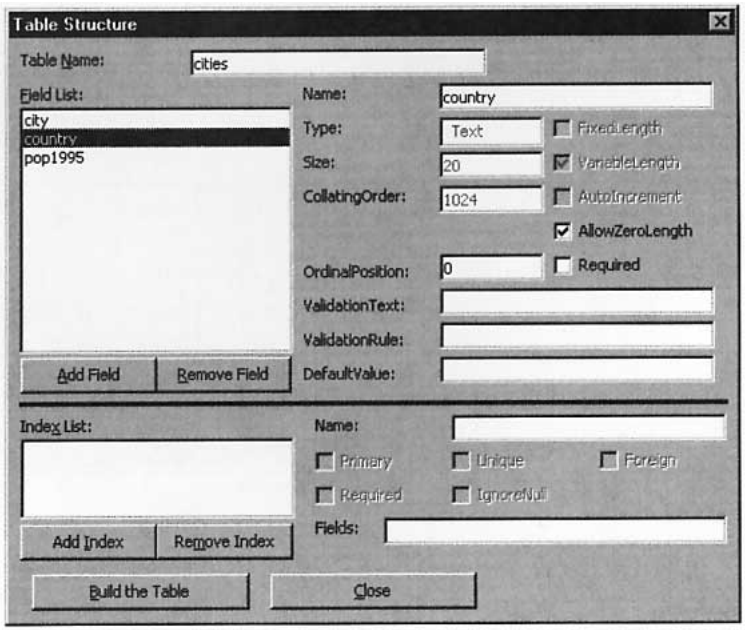


FIGURE 11-5 *Table Structure Window. (Invoked from the Database Window with the Right Mouse Button by Choosing New or Design.)*

How to use the Table Structure window

1. Type the name of the table in the Table Name text box.
2. Click on the Add Field button. (An Add Field window will be displayed.)
3. Type the name of a field in the Name text box.
4. Click on the down arrow of the Type combo box and select a type from the dropdown list. (We use primarily the type "Text" for string, and "Single" or "Integer" for numbers.)
5. If the data type is "Text," type the length of the largest possible string needed in the Size box. (For instance, length 20 should suffice for names of cities. The length must be no longer than 255 characters.)
6. Press OK to add the field to the table.
7. Repeat Steps 3 through 6 until you have added all the fields to the table. When you are through, click on the Close button to return to the Database window.
8. To delete a field, highlight the field by clicking on it in the list box and then press the Remove Field button.

9. When all fields have been specified, press the "Build the Table" button to save the table and return to the Database window. (If we later decide to add a new field or delete an existing field, we can return to the Table Structure window by highlighting the table in the Database window, clicking the right mouse button, and choosing Design.)

   (***Note:*** The "Build the Table" button appears only for new tables, otherwise, use the Close button return to the Database window.)

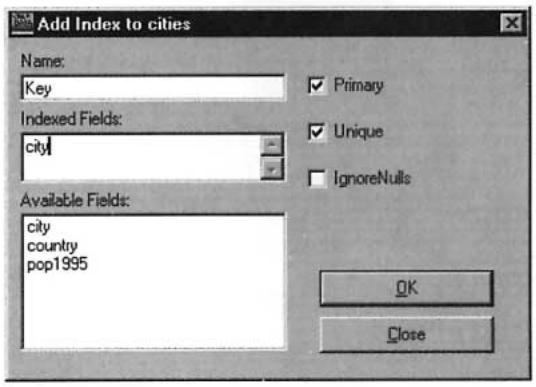


FIGURE 11-6 *Add Index Window (Invoked from the Table Structure Window by Pressing Add Index.)*

How to use the Add Index window to specify a primary key

1. Type in a name for an index, such as Principal, click on the field which is to be the primary field, and place check marks in the Primary and Unique check boxes.
2. Click OK and then click Close.

   Note: To specify an ordinary index, follow all steps except turning on the Primary check box.
3. Press "Build the Table" or Close to return to the Database window.

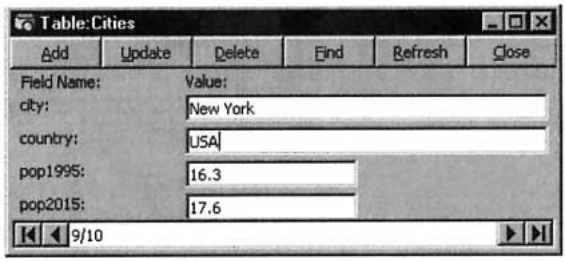


FIGURE 11-7 *Table window (Invoked from the Database Window by Double-Clicking on the Table name. Or, Choose the Table, Click on the Right Mouse Button, and Click on Open.)*

How to use the Table window

1. If the table has no records, the Value text boxes will be blank. To add a record, type data into the fields and press the Update button.
2. To add a new record to the end of a table that already contains records, click the Add button, type data into the fields, and press the Update button. If you make a mistake entering the data, click on the Cancel button.

3. To view an existing record, use the navigator buttons (identical to those of the data control) or the Find button to move to a record of a specified value.
4. To remove the current record from the table, click the Delete button. The record is immediately deleted and cannot be recovered.
5. To edit an existing record, make changes in the data and click on the Update button.
6. Press the Close button to return to the Database window.

   At any time, the main section of the Database window contains a list of the tables you have specified. Initially the main section is blank except for Properties. (This is a list of properties pertaining to the database.) If you click the right mouse button while highlighting a table, a menu will appear with the following options:

Menu Item	Use
Open	Open the table to allow records to be added.
Design	Specify the design (number and types of fields).
Rename	Rename the given table.
Delete	Remove the given table from the database.
Copy Structure	Copies the structure of the given database with or without the data currently contained in the database.
Refresh List	Redisplay the list in the Database window.
New Table	Open the Table Structure window.
New Query	Open the Query Builder window.

Creating a Database in Code

The following lines of code create a database named "DBNAME.MDB" with two tables, TABLE1 and TABLE2. TABLE1 has the fields FIELD1A (type Text with maximum length 50) and FIELD1B (numeric of type Single). TABLE2 has analogous fields called FIELD2A and FIELD2B, and also FIELD2C of type Integer. FIELD1A is a primary key of TABLE1 and FIELD2A is a foreign key in TABLE2 referring to FIELD1A of TABLE1. FIELD2C is a primary key of TABLE2. The record ("alpha", 1997) is placed in TABLE1 and the record ("alpha", 2000, 1) is placed in TABLE2. This code, which is contained in the file CREATEDB.TXT accompanying this textbook, is intended as a template that you can modify to create a database programmatically. (***Note:*** The code can be placed into an event procedure. Before you run the program, click on References in the Project menu and make sure that "DAO 3.51 Object Library" is selected.)

```
Dim MyDB As Database, MyWs As Workspace
Dim T1, T2 As TableDef
Dim T1Flds(1 To 2), T2Flds(1 To 3) As Field
Dim TempFld As Field
Dim T1Idx, T2Idx As Index
Dim Rel As Relation
Dim MyRec As Recordset
'Create Database
Set MyWs = DBEngine.Workspaces(0)
Set MyDB = MyWs.
CreateDatabase("C:\DBNAME.MDB", dbLangGeneral)'Create first Table, TABLE1
Set T1 = MyDB.CreateTableDef("TABLE1")
'Specify fields for TABLE1
'Note the use of the optional parameter 50 for field size
'If 50 is omitted, the size will default to 20
Set T1Flds(1) = T1.CreateField("FIELD1A", dbText, 50)
Set T1Flds(2) = T1.CreateField("FIELD1B", dbSingle)
'Add the New fields to the field list in the Table
```

```
T1.Fields.Append T1Flds(1)
T1.Fields.Append T1Flds(2)
'Specify a primary field for TABLE1
Set T1Idx = T1.CreateIndex("FIELD1A")
T1Idx.Primary = True
T1Idx.Unique = True
T1Idx.Required = True
Set T1Flds(1) = T1Idx.CreateField("FIELD1A")
'Add this field to the field list of the Index
T1Idx.Fields.Append T1Flds(1)
'Add this Index to the index list of the Table
T1.Indexes.Append T1Idx
'Add the Table to the Database
MyDB.TableDefs.Append T1
'Create TABLE2
Set T2 = MyDB.CreateTableDef("TABLE2")
'Specify fields for TABLE2
Set T2Flds(1) = T2.CreateField("FIELD2A", dbText, 50)
Set T2Flds(2) = T2.CreateField("FIELD2B", dbSingle)
Set T2Flds(3) = T2.CreateField("FIELD2C", dbInteger)
'Add the new fields to the field list of the Table
T2.Fields.Append T2Flds(1)
T2.Fields.Append T2Flds(2)
T2.Fields.Append T2Flds(3)
'Set the primary field for TABLE2Set T2Idx = T2.CreateIndex("FIELD2C")
T2Idx.Primary = True
T2Idx.Unique = True
T2Idx.Required = True
Set T2Flds(3) = T2Idx.CreateField("FIELD2C")
'Add this field to the field list of the Index
T2Idx.Fields.Append T2Flds(3)
'Add this index to the index list of TABLE2
T2.Indexes.Append T2Idx
'Add TABLE2 to the Database
MyDB.TableDefs.Append T2
'Set up the relation between the tables
Set Rel = MyDB.CreateRelation("foreign", "TABLE1", "TABLE2")
Rel.Attributes = 0
'Mark the primary field in TABLE1
Set T2Flds(1) = Rel.CreateField("FIELD1A")
'Mark the foreign key field in TABLE2
T2Flds(1).ForeignName = "FIELD2A"
'Add the field to the field list of the relation
Rel.Fields.Append T2Flds(1)
'Add the relation to the database
MyDB.Relations.Append Rel
'Add a record to each table
'Open a recordset referring to TABLE1
Set MyRec = T1.OpenRecordset
'Create a record
MyRec.AddNew
MyRec("FIELD1A") = "alpha"
MyRec("FIELD1B") = 1997
'Update the recordset
MyRec.Update
```

```
'Close the recordset referring to TABLE1
MyRec.Close
'Open a recordset referring to TABLE2
Set MyRec = T2.OpenRecordset
'Create a record
MyRec.AddNewMyRec("FIELD2A") = "alpha"
MyRec("FIELD2B") = 2000
MyRec("FIELD2C") = 1
'Update the recordset
MyRec.Update
'Close the recordset
MyRec.Close
'Close the database
MyDB.Close
```

Principles of Database Design

In order to design a database, you must decide how many tables to use, what fields to include in each table, what fields to make primary or foreign keys, and what validation criteria to specify. The programming paradigm "Plan first, code later" also applies to databases. Before you design a database, you first must understand how the database will be used. For instance, you must know in what way and how often the user will update the data, and what types of reports the user will want to be generated from the database. Failure to plan ahead can be very costly.

You have no doubt read about the "year 2000 crisis." Databases designed in the 1960s and 1970s saved space by using just two digits to refer to each year. As a result, they cannot distinguish between the year 2000 and the year 1900. Correcting this oversight is predicted to cost government and industry billions of dollars.

Good relational database design is more an art than a science. However, there are certain fundamental guidelines that the designer should keep in mind.

Include the necessary data.

After you have a clear picture of the types of reports the user must generate, you will know what fields and relationships to include. On the other hand, some data that seem relevant do not have to be kept.

Be aware that data should often be stored in their smallest parts.

For instance, city, state, and zip code are usually best stored in three fields. So doing will allow you to sort a mailing by zip code or target a mailing to the residents of a specific city.

Avoid redundancy.

The process of avoiding redundancy by splitting a table into two or more related tables is called **data normalization**. For instance, the excessive duplication in Table 11.5(a) can be avoided by replacing the table with the two related tables, Tables 11.5(b) and 11.5(c).

TABLE 11.5(A)
A Table with Redundant Data

course	section	name	time	credits	prerequisite
CS102	1001	Intro to Databases	MWF 8–9	3	CS101
CS102	1002	Intro to Databases	MWF 1–2	3	CS101
CS102	1003	Intro to Databases	MWF 2–3	3	CS101
CS102	1004	Intro to Databases	MWF 3–4	3	CS101
CS105	1001	Visual Basic	MWF 1–2	4	CS200

TABLE 11.5(B)

course	section	time
CS102	1001	MWF 8–9
CS102	1002	MWF 1–2
CS102	1003	MWF 2–3
CS102	1004	MWF 3–4
CS105	1001	MWF 1–2

TABLE 11.5(C)

course	name	credits	prerequisite
CS102	Intro to Databases	3	CS101
CS105	Visual Basic	4	CS200

Avoid tables with intentionally blank entries.

Tables with entries that are intentionally left blank waste space and are inefficient to use. Table 11.6, which serves as a directory of faculty and students, has an excessive number of blank entries. The table should be split into two tables, each dealing with just one of the groups.

TABLE 11.6
A Table with an Excessive Number of Blank Entries

name	ssn	classification	date hired	dept	office number	gpa	credits earned
Sarah Brown	816-34-9012	student				3.7	78
Pat Riley	409-22-1234	faculty	9/1/90	biology	Y-3014		
Joe Russo	690-32-1108	faculty	9/1/88	math	T-2008		
Juan Lopez	509-43-4110	student				3.2	42

Strive for table clarity.

Each table should have a basic topic and all the data in the table should be connected to that topic.

Don't let a table get unnecessarily large.

A bookstore might keep a permanent record of each purchase for the purpose of targeting mailings. A better solution is to add a couple of extra fields to the customer table that identify the types of books of interest to the customer.

Avoid fields whose values can be calculated from existing fields.

A calculated field is one whose value can be determined from other fields. For instance, if a table has fields for both the population and area of a state, then there is no need to include a field for the population density.

COMMENTS

1. Visual Data Manager can not create foreign keys.
2. You can add additional fields to a database table, but you cannot rename, modify, or delete fields without deleting the entire table and rebuilding each field. For this reason, design your tables carefully.
3. The total length of a record in a table is limited to 2000 bytes.
4. Be careful if you create a database with empty tables. If you pull it into a program and try to move through it with the navigator arrows, an error will be generated. You must add at least one record before starting to navigate.

SUMMARY

1. A *table* is a group of data items arranged in a rectangular array, each containing the same categories of information. Each data item (row) is called a *record*. Each category (column) is called a *field*. Two tables with a common field are said to be *related*. A *database* is a collection of one or more, usually related, tables.
2. The *data control* is used to access a database. When a text box is bound to a data control through its DataSource and DataField properties, the user can read and edit a field of the database. At any time, one record is specified as the *current record*. The user can change the current record with the data control's *navigator arrows* or with *Move* statements. The property *RecordCount* count records, the property *BOF* indicates whether the beginning of the recordset has been reached, and the property *EOF* indicates whether the end of the recordset has been reached. The *Value* property of Fields ("*fieldName*") reads the contents of a field of the current record. The *Validate event*, which can be used to control edits, is triggered whenever the current record is about to be changed.
3. A *primary key* is a field or set of fields that uniquely identifies each row of a table. The *rule of entity integrity* states that no record can have a null entry in a primary key. A *foreign key* is a field or set of fields in one table that refers to a primary key in another table. The *rule of referential integrity* states that each value in the foreign key must also appear in the primary key.
4. Structured Query Language (SQL) is used to create a "virtual" table consisting of a subtable of a table or of a join of two tables and imposes an order on the records. The subtable is specified with the reserved words SELECT, FROM, WHERE, ORDER BY, and INNER JOIN . . . ON. The WHERE clause of an SQL statement commonly uses the Like operator in addition to the standard operators. SQL statements are either employed at design time or run time as the setting of the RecordSource property. During run time, the Refresh method for the data control should be executed after the RecordSource property is set.
5. The *MS FlexGrid grid control* can show an entire "virtual" table in a spreadsheet-like display. *Data-bound list box* and *data-bound combo box* controls can display the contents of a single field for an entire table or recordset.
6. *Visual Data Manager*, a database management program supplied with most versions of Visual Basic, can be used to create a database and specify primary keys and validation criteria. A database can be created by code with the Professional or Enterprise editions of Visual Basic.
7. Although good database design is an art, there are several fundamental principles that usually should be followed.

PROGRAMMING PROJECTS

1. The database MICROLND.MDB (on the accompanying CD) is maintained by the Microland Computer Warehouse, a mail order computer supply company. Tables 11.7 through 11.9 show parts of three tables in the database. The table Customers identifies each customer by an ID number and gives, in addition to the name and address, the total amount of purchases during the current year. The table Inventory identifies each product in stock by an ID number and gives, in addition to its description and price (per unit), the quantity currently in stock. The table Orders gives the orders received today. Suppose it is now the end of the day. Write a program that uses the three tables to do the following:

(a) Update the *quantity* field of the Inventory table.

(b) Display in a list box the items that are out of stock and therefore must be reordered.

(c) Update the *amtOfSales* field of the Customers table.

(d) Print bills to all customers who ordered during the day. (You can assume that each customer only calls once during a particular day and therefore that all items ordered by a single customer are grouped together. The bill should indicate if an item is currently out of stock. You can just display the bills one at a time in a picture window instead of actually printing them.)

TABLE 11.7
First Three Records of the Customers Table

custID	name	street	city	amtPurchases
1	Michael Smith	2 Park St.	Dallas, TX 75201	234.50
2	Brittany Jones	5 Second Ave	Tampa, FL 33602	121.90
3	Warren Pease	7 Maple St	Boston, MA 02101	387.20

TABLE 11.8
First Three Records of the Inventory Table

itemID	description	price	quantity
PL208	Visual Basic – Standard	89.50	12
SW109	MS Office Upgrade	195.95	2
HW913	PaperPort ix	300.25	8

TABLE 11.9
First Four Records of the Orders Table

custID	itemID	quantity
3	SW109	1
1	PL208	3
1	HW913	2
2	PL208	1

2. Most college libraries have a computerized online catalog that allows you to look up books by author or title. Use the database BIBLIO.MDB to design such a catalog. You should create a new database with the necessary tables and fields and copy all needed information from BIBLIO.MDB into the new database. (One field should hold the number of copies that the library owns and another field should hold the number of copies currently on the shelf. Use the Rnd function to fill the first field with numbers from 1 to 3.) The user should be able to do the following.

   (a) View the books by author in either alphabetical or chronological order and then obtain relevant information (publisher, ISBN, copyright year, number of copies available and owned) on a specific book.

   (b) Determine if a book with a specified title is owned by the library.

   (c) Search for all books containing a certain word in its title.

   (d) Check out a book that is on the shelf.

   (e) Reserve a book that is currently not on the shelf. (A number can be assigned to each reservation to determine priority.)

   The librarian should be able to generate a listing of all books for which there is a waiting list.

SECTION 12

OBJECT-ORIENTED PROGRAMMING

12.1 CLASSES AND OBJECTS

noun A word used to denote or name a person, place, thing, quality, or act.
verb That part of speech that expresses existence, action, or occurrence.
adjective Any of a class of words used to modify a noun or other substantive by limiting, qualifying, or specifying.

The American Heritage Dictionary of the English Language

"A good rule of thumb for object-oriented programming is that classes are the nouns in your analysis of the problem. The methods in your object correspond to verbs that the noun does. The properties are the adjectives that describe the noun."

Gary Cornell & David Jezak[1]

Practical experience in the financial, scientific, engineering, and software design industries has revealed some difficulties with traditional program design methodologies. As programs grow in size and become more complex, and as the number of programmers working on the same project increases, the number of dependencies and interrelationships throughout the code increases exponentially. A small change made by one programmer in one place may have many effects, both intended and unintended, in many other places. The effects of this change may ripple throughout the entire program, requiring the rewriting of a great deal of code along the way.

A partial solution to this problem is "data hiding" where, within a unit, as much implementation detail as possible is hidden. Data hiding is an important principle underlying object-oriented programming. An object is an encapsulation of data and procedures that act on the data. The only thing of concern to a programmer using an object is the tasks that the object can perform and the parameters used by these tasks. The details of the data structures and procedures are hidden within the object.

Two types of objects will be of concern to us, **control objects** and **code objects**. Examples of control objects are text boxes, picture boxes, command buttons and all the other controls that can be created from the Visual Basic toolbox. Code objects are specific instances of user-defined types that are defined similarly to record types in a separate module. Both types of objects have properties and respond to methods. The main differences are that control objects are predefined and have physical manifestations, whereas code objects must be created by the programmer and exist solely in a portion of memory. In this section, when we use the word "object" without a qualifier, we mean "code object."

[1]FActiveX Visual Basic 5 Control Creation Edition, Prentice-Hall, 1997.

Whenever you double-click on the TextBox icon in the toolbar, a new text box is created. Although each text box is a separate entity, they all have the same properties and methods. Each text box is said to be an instance of the class TextBox. In some sense, the TextBox icon in the toolbox is a template for creating text boxes. (When you look at the properties window for a text box, the dropdown list box at the top of the window says something like "Text1 TextBox". "Text1" is the name of the control object and "TextBox" is the name of its class.) You can't set properties or invoke methods of the TextBox icon, only of the specific text boxes that it creates. The analogy is often made between the TextBox icon and a cookie cutter. The cookie cutter is used to create cookies that you can eat, but you can't eat the cookie cutter.

Object-oriented programs are populated with objects that hold data, have properties, respond to methods, and raise events. (The generation of events will be discussed in the next section.) Six examples are as follows:

1. In a professor's program to assign and display semester grades, a student object might hold a single student's name, social security number, midterm grade, and final exam grade. A SemGrade method might calculate the student's semester grade. Events might be raised when improper data is passed to the object.
2. In a payroll program, an employee object might hold an employee's name, hourly wage, and hours worked. A CalculatePay method would tell the object to calculate the wages for the current pay period.
3. In a checking account program, a check register object might record and total the checks written during a certain month, a deposit slip object might record and total the deposits made during a certain month, and an account object might keep a running total of the balance in the account. The account object would raise an event to alert the bank when the balance gets too low.
4. In a bookstore inventory program, a textbook object might hold the name, author, quantity in stock, and wholesale price of an individual textbook. A RetailPrice method might instruct the book object to calculate the selling price of the textbook. An event could be triggered when the book goes out of stock.
5. In a game program, an airplane object might hold the location of an airplane. At any time, the program could tell the object to display the airplane at its current location or to drop a bomb. An event can be triggered each time a bomb moves so that the program can determine if anything was hit.
6. In a card game, a card object might hold the denomination and suit of a specific card. An IdentifyCard method might return a string such as "Ace of Spades." A deck of cards object might consist of an array of card objects. A ShuffleDeck method might thoroughly shuffle the deck and a Shuffling event might indicate the progress of the shuffle.

The most important object-oriented term is **class**. A class is a template from which objects are created. The class specifies the properties and methods that will be common to all objects that are instances of that class. Classes are formulated in class modules. An object, which is an instance of a class, can be created in a program with a pair of statements of the form

```
Private objectName As className  'In General Declarations section
Set objectName = New className   'In procedure
```

In the program, properties of the object is accessed with statements of the form shown in the following table.

Task	Statement
Assign a value to a property	objectName.propertyName = value
Display the value of a property	picBox.Print objectName.propertyName
Carry out a method	objectName.methodName(arg1, ...)
Raise an event	RaiseEvent eventName

The following walkthrough creates a student class and a program that uses that class. The data stored by an object of this class are name, social security number, and grades on two exams (midterm and final).

1. Start a new program.
2. From the Project menu on the toolbar, click on Add Class Module.
3. Double-click on Class Module in the Add Class Module dialog box. (The window that appears looks like an ordinary code window.)
4. If the Properties window is not visible, press F4 to display it. (Notice that the class has the default name Class1.)
5. Change the setting of the Name property to CStudent. (We will follow the common convention of beginning each class name with the uppercase letter C.)
6. Type the following lines into the code module.

```
Private m_name As String
Private m_ssn As String
Private m_midterm As Single
Private m_final As Single
```

   (These lines of code declare four variables that will be used to hold data. The word Private guarantees that the variables cannot be accessed directly from outside the object. In object-oriented programming terminology, these variables are called **member variables** (or **instance variables**). We will follow the common convention of beginning the name of each member variable with the prefix "m_".)
7. From the Tools menu on the toolbar, click on Add Procedure. (As before, an Add Procedure dialog box will appear.)
8. Type "Name" into the Name text box, click on Property in the Type frame, and click on OK. The following lines will appear in the class module window.

```
Public Property Get Name() As Variant

End Property

Public Property Let Name(ByVal vNewValue As Variant)

End Property
```

9. Change the words Variant to String, the word vNewValue to vName, and type code into the two property procedures as shown below.

```
Public Property Get Name() As String
  Name = m_name
End Property
Public Property Let Name(ByVal vName As String)
  m_name = vName
End Property
```

   The first procedure will be called by our program to retrieve the value of the variable *m_name* and the second procedure will be called to assign a value to the variable *m_name*.

10. In the same manner as in Steps 7–9, create the following pair of property procedures that will be used to retrieve and assign values to the variable *m_ssn*.

```
Public Property Get SocSecNum() As String
  SocSecNum = m_ssn
End Property

Public Property Let SocSecNum(ByVal vNum As String)
  m_ssn = vNum
End Property
```

11. Property procedures can be typed directly into the class module without the use of Add Procedure. Also, property procedures needn't come in pairs. For instance, if we wanted the value of a member variable to be "write only," we would use a Property Let procedure and have no Property Get procedure. Type the following two property procedures into the class module. The inclusion of the word Public is optional.

```
Property Let midGrade(ByVal vGrade As Single)
  m_midterm = vGrade
End Property

Property Let finGrade(ByVal vGrade As Single)
  m_final = vGrade
End Property
```

12. Create the following ordinary Public function with the name SemGrade.

```
Public Function SemGrade() As String
   Dim grade As Single
   grade = (m_midterm + m_final) / 2
   grade = Round(grade) 'Round the grade
   Select Case grade
      Case Is = 90
        SemGrade = "A"
      Case Is = 80
        SemGrade = "B"
      Case Is = 70
        SemGrade = "C"
      Case Is = 60
        SemGrade = "D"
      Case Else
        SemGrade = "F"
   End Select
End Function
```

    (This function will be used by our program to invoke a method requesting an object to calculate a student's semester grade.)

13. From the File menu, click on Save CStudent As and save the class module with the name 13-1-1S.cls. (We chose this name since the class will be used in Example 1. Another good choice of name would have been Student.cls. The extension cls is normally given to files holding class modules.)

14. Click on the form window to activate it. We can now write a program that creates an object, call it *pupil*, that is an instance of the class and uses the object to calculate a student's semester grade. The object variable is declared (in the general declarations section of the code module) with the statement

```
Private pupil As CStudent
```

and then an instance of the class is created inside a procedure with the statement

```
Set pupil = New CStudent
```

The object *pupil* will be local to the procedure. That is, it will cease to exist after the procedure has ended. The Property Let procedures are used to assign values to the member variables, and the Property Get procedures are used to retrieve values. The function SemGrade becomes a method for obtaining the student's grade.

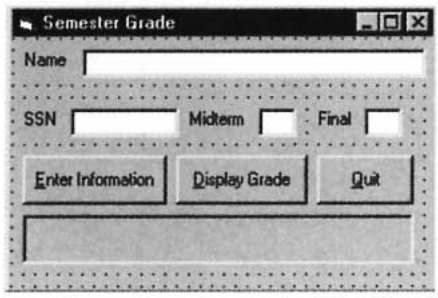
Some examples are

```
pupil.Name = "Adams, Al"     'Assign a value to m_name
picBox.Print pupil.Name      'Display the student's name
picBox.Print pupil.SemGrade  'Display the student's semester grade
```

The first statement calls the Property Let Name procedure, the second statement calls the Property Get Name procedure, and the third statement calls the method procedure SemGrade.

EXAMPLE 1

The following program uses the class CStudent to calculate and display a student's semester grade.

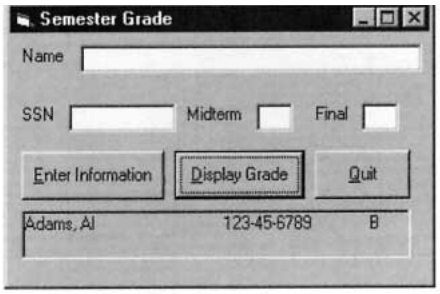


Object	Property	Setting
frm13_1_1	Caption	Semester Grade
lblName	Caption	Name
txtName	Text	(blank)
lblSSN	Caption	SSN
txtSSN	Text	(blank)
lblMidterm	Caption	Midterm
txtMidterm	Text	(blank)
lblFinal	Caption	Final
txtFinal	Text	(blank)
cmdEnter	Caption	&Enter Information
cmdDisplay	Caption	&Display Grade
cmdQuit	Caption	&Quit
picGrade		

```
'Student Class (CStudent)
    Private m_name As String
    Private m_ssn As String
    Private m_midterm As Single
    Private m_final As Single
    Property Get Name() As String
   Name = m_name
End Property

Property Let Name(ByVal vName As String)
   m_name = vName
End Property

Property Get SocSecNum() As String
   SocSecNum = m_ssn
End Property

   Property Let SocSecNum(ByVal vNum As String)
   m_ssn = vNum
End Property
```

```
Property Let midGrade(ByVal vGrade As Single)
  m_midterm = vGrade
End Property

Property Let finGrade(ByVal vGrade As Single)
  m_final = vGrade
End Property

Public Function SemGrade() As String
  Dim grade As Single
  grade = (m_midterm + m_final) / 2
  grade = Round(grade) 'Round the grade
  Select Case grade
       Case Is = 90
         SemGrade = "A"
       Case Is = 80
         SemGrade = "B"
       Case Is = 70
         SemGrade = "C"
       Case Is = 60
         SemGrade = "D"
       Case Else SemGrade = "F"
  End Select
End Function

'Form Code
Private pupil As CStudent 'pupil is an object of class CStudent

Private Sub Form_Load()
  Set pupil = New CStudent
End Sub

Private Sub cmdEnter_Click()
  'Read the values stored in the text boxes
  pupil.Name = txtName
  pupil.SocSecNum = txtSSN
  pupil.midGrade = Val(txtMidterm)
  pupil.finGrade = Val(txtFinal)
  'Clear Text Boxes
  txtName.Text = ""
  txtSSN.Text = ""
  txtMidterm.Text = ""
  txtFinal.Text = ""
  picGrade.Cls
  picGrade.Print "Student recorded."
End Sub

Private Sub cmdDisplay_Click()
  picGrade.Cls
  picGrade.Print pupil.Name; Tab(28); pupil.SocSecNum(); _
                        Tab(48); pupil.SemGrade
End Sub

Private Sub cmdQuit_Click()
  End
End Sub
```

[Run, enter the data for a student (such as "Adams, Al", "123-45-6789", "82", "87"), press the Enter Information button to send the data to the object, and press the Display Grade button to display the student's name, social security number, and semester grade.]



In summary, the following seven steps are used to create a class.

1. Identify a *thing* in your program that is to become an object.
2. Determine the properties and methods you would like the object to have.
3. A class will serve as a template for the object. Add a class module from the Project menu.
4. Set the name property of the class. (A common convention is to begin the name with the letter C.)
5. For each of the properties in Step 2, declare a private member variable with a statement of the form

```
Private m_variableName As dataType
```

6. For each of the member variables in Step 5, create one or two public property procedures to retrieve and assign values of the variable. The general forms of the procedures are

```
Public Property Get ProcedureName() As DataType
  ProcedureName = m_variableName
  (Possibly additional code.)
End Property

Public Property Let Procedurename(ByVal vNewValue As DataType)
  m_variableName = vNewValue
  (Possibly additional code.)
End Property
```

   Note: Since the member variables were declared as Private, they cannot be accessed directly from outside an object. They can only be accessed through Property procedures which allow values to be checked and perhaps modified. Also, a Property procedure is able to take other steps necessitated by a change in a member variable.

7. For each method in Step 2, create a Sub procedure or Function procedure to carry out the task.

EXAMPLE 2

Modify the program in Example 1 to calculate semester grades for students who have registered on a "Pass/Fail" basis. Use a class module to calculate the semester grade.

SOLUTION:

We will create a new class, named CPFStudent, with the same member variables and property procedures as the class CStudent. The only change needed in the class module occurs in the SemGrade method. The new code for this method is

```
Public Function SemGrade() As String
   Dim grade As Single
   grade = (m_midterm + m_final) / 2
   grade = Round(grade) 'Round the grade
   If grade = 60 Then
      SemGrade = "Pass"
    Else
      SemGrade = "Fail"
   End If
End Function
```

The only change needed in the Form code is to replace the two occurrences of CStudent with CPFStudent. When the program is run with the same input as in Example 1, the output will be

```
Adams, Al 123-45-6789 Pass
```

THE INITIALIZE EVENT PROCEDURE

The Object drop-down combo box in a class module window displays two items General and Class. When you click on Class, the following template appears:

```
Private Sub Class_Initialize()


End Sub
```

This event procedure is automatically invoked when an object is created from the class. Any code you type into the procedure is then executed. This procedure is used to set default values for member variables and to create other objects associated with this object.

Since methods are created with ordinary Function or Sub procedures, arguments can be passed to them when they are called. The graphical program in Example 3 makes use of arguments. The program involves "twips," which are a unit of screen measurement. (One inch is about 1440 twips.) The settings for the Top, Left, Height, and Width properties of a control are given in twips. For instance, the statements

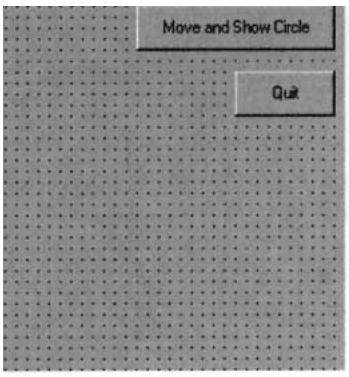
```
Image1.Width = 1440    '1 inch
Image1.Height = 2160   '1.5 inch
Image1.Top = 2880      '2 inches
Image1.Left = 7200     '5 inches
```

set the size of the image control as 1" by 1.5", and place the control 2 inches from the top of the form and 5 inches from the left side of the form. (See Figure 6-6 in Section 6.3.) Programs using the Height property of the form should have the BorderStyle property of the form set to "0-None" since otherwise the height of the border is included in the height of the form.

EXAMPLE 3

Write a program containing a Circle object. The object should keep track of the center and radius of the circle. (The center is specified by two numbers, called the coordinates, giving the distance from the left side and top of the form. Distances and the radius should be measured in twips.) A Show method should display the circle on the form and a Move method should add 500 twips to each coordinate of the center of the circle. Initially, the circle should have its center at (500, 500) and radius 500. The form should have a command button captioned "Move and Show Circle" that invokes both methods.

SOLUTION:

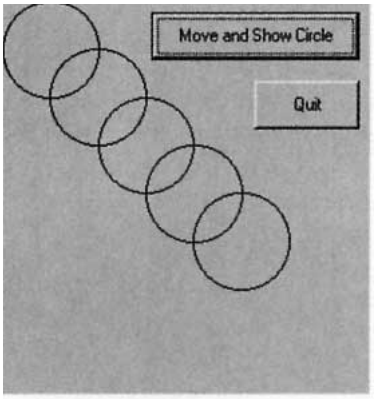


Object	Property	Setting
frmCircles	BorderStyle	0-None
cmdMove	Caption	Move and Show Circle
cmdQuit	Caption	Quit

```
'Class module for CCircle
    Private m_x As Integer  'Dist from center of circle to left side of form
    Private m_y As Integer  'Distance from center of circle to top of form
    Private m_r As Single   'Radius of circle


Private Sub Class_Initialize()
   'Set the initial center of the circle to the upper
   'left corner of the form and set its radius to 500.
   m_x = 0
   m_y = 0
   m_r = 500
End Sub

Public Property Get Xcoord()
   As Integer Xcoord = m_x
End Property

Public Property Let Xcoord(ByVal vNewValue As Integer)
   m_x = vNewValue
End Property

Public Property Get Ycoord() As Integer
   Ycoord = m_y
End Property

Public Property Let Ycoord(ByVal vNewValue As Integer)
   m_y = vNewValue
End Property

   Public Sub Show()
   'Display the circle.
   'See discussion of Circle method in Section 10.4.
   frmCircles.Circle (m_x, m_y), m_r
End Sub

Public Sub Move(Dist)
   'Move the center of the circle Dist twips to the right
   'and Dist twips down.
   m_x = m_x + Dist
   m_y = m_y + Dist
   Call Show
```

```
End Sub

'Form code
Private round As CCircle

Private Sub Form_Load()
   Set round = New CCircle
End Sub

Private Sub cmdMove_Click()
   round.Move (500)
End Sub

Private Sub cmdQuit_Click()
   End
End Sub
```

[Run, and press the command button five times.]



COMMENTS

1. The statement

   ```
   Set objectVar = Nothing
   ```

   dissociates the object variable from the actual object and frees up the memory used by the variable.
2. An object variable declared inside a procedure ceases to exist when the procedure is exited. (We say that the object falls out of scope.) The effect is the same as when the variable is set to Nothing.
3. The counterpart to the Initialize event is the Terminate event procedure which has the template

   ```
   Private Sub Class_Terminate()

   End Sub
   ```

   and is automatically invoked when all references to the object are Set to Nothing or when the object falls out of scope. This procedure is often used to set any objects you may have created inside the class module to Nothing.

4. Methods can be either Function or Sub procedures. A method that returns a value must be a Function procedure; otherwise it can be a Sub procedure.
5. A program with a class module has at least three components: a form, form code, and class module code. Although the View menu gives you access to the form (Alt/V/B) and the form code (Alt/V/C), only the Project Explorer gives access to all three. To view the code for a class module, double-click on the name of the class in the Project Explorer.
6. A class module is saved in a file whose file name has the extension .cls. Therefore, classes have both a name (C*something*) and a file name (*something*.cls).
7. To insert an existing (saved) class module into a program, click on Add Class Module in the Project menu, click on the Existing tab, and enter the filespec for the class module.
8. To delete a class from a program, right click on the class name in the Project Explorer and click on Remove *className*.
9. An object also can be declared with a statement of the form

```
Private pupil As Object
```

   However, objects so declared require more time and effort for Visual Basic to access.
10. The set of properties, methods, and events for a class is called the class **interface**. The classes CStudent and CPFStudent have the same interface, even though they carry out the task of computing a grade differently. The programmer need only be aware of the SemGrade method and needn't be concerned about its implementation. The feature that two classes can have behaviors that are named the same and have essentially the same purpose but different implementations is called **polymorphism**.
11. Sometimes you will have difficulty deciding whether an interface item should be a property or a method. As a rule of thumb, properties should access data and methods should perform operations.
12. The ByVal (which stands for "by value") keyword is automatically inserted before parameters in Property Let statements invoked from the Tools menu. However, this keyword is optional. The default way of passing an argument to a parameter is ByRef (which stands for "by reference"). Usually, passing by value is more efficient than passing by reference.
13. The default parameter name in a Property Let procedure is vNewValue. Although we usually substitute a meaningful name, we retain the convention of beginning the name with the prefix v.
14. We could have preceded our member variables with the keyword Public and allowed direct access to the variables. However, this is considered poor programming practice. By using Property Let procedures to update the data, we can enforce constraints and carry out validation.
15. In a class module, a property is implemented by two procedures, one to set and the other to retrieve the property value. These procedures that access properties are sometimes referred to as **accessor** methods.

12.2 COLLECTIONS AND EVENTS

"An object without an event is like a telephone without a ringer."

Anonymous

A collection is an entity, similar to an array, that is especially well-suited to working with sets of objects. This section discusses collections of objects and user-defined events for classes.