# Visual Basic in 12 Easy Lessons Table of Contents

# Visual Basic in 12 Easy Lessons



## Acknowledgments

Mega-thanks go from me to the editorial staff at Sams Publishing. I especially want to thank two editors who are my friends more than my editors: Rosemarie Graham and Dean Miller. Their excellence and caring go far beyond the usual editor/author relationship. If readers find good things in this book it is because of them. If readers find not-so-good things, I take full responsibility.

The technical editor, Ricardo Birmele, had the difficult job of turning my words into accurate text and I thank him so much. In addition, I want to thank the fine production staff at Sams Publishing who care more about the final product than any other staff in the world. Two members of Sam's fine editorial staff who worked harder than the author are Fran Hatton and Matt Usher. Somehow they made a book out of my random words. I appreciate your eagle eyes.

As always, my beautiful bride Jayne and my parents Glen and Bettye Perry are my three favorites in the

world.

# About the Author

**Greg Perry** has written more than 30 books and continues to take programming topics and bring them down to the beginner's level. Perry has been a programmer, trainer, and speaker for the past 17 years. He received his first degree in computer science and then a master's degree in corporate finance. Among the other books he has written are *Teach Yourself Object-Oriented Programming With Visual C++*, *QBasic Programming 101*, *Absolute Beginner's Guide to Programming* (all Sams Publishing), and *The Complete Idiot's Guide to Visual Basic* (Alpha Books). In addition, he has published articles in several publications, including *Software Development*, *Access Advisor*, and *PC World*, and *Data Training*. In his spare time, he manages rent houses, drinks gallons of *Snapple*, participates in local political campaigns, and constantly tries to improve his fractured Italian.

# Introduction

The book you hold offers something you may not have encountered before. Whereas other books might teach you Visual Basic, this book includes a working Visual Basic programming system on a disk to accompany the text's tutorial of the language. With this book, there is literally nothing else to buy (except, of course, the computer)! Microsoft's Visual Basic programming system turns your computer into a Visual Basic programming powerhouse with which you can write Windows applications. The disk that comes with this book also includes all the book's programming projects, code listing, as well as answers to all the exercises at the end of each unit.

Despite the great disk included, this book would be worthless if it didn't teach Visual Basic. *Visual Basic Programming in 12 Easy Lessons* begins at the beginning, assuming that you don't know Visual Basic or any of the BASIC-like languages that preceded Visual Basic. You'll be learning how to program, perform input and output, how to work with disk files, and draw graphics through Visual Basic programs.

Visual Basic is one of the most successful Windows programming tools on the market today and for good reason. Whereas other Windows programming languages require steep learning curves, Visual Basic lets you design and write complete Windows applications without a lot of effort and tedium.

## Who Should Use This Book

*Visual Basic Programming in 12 Easy Lessons* is aimed primarily at beginning programmers who either have never programmed or who have never seen a Visual Basic program before. Text, questions, exercises, and numerous program listings are aimed at both beginning programmers and those programmers new to Visual Basic.

If you already program but have never tackled Visual Basic or Windows programming, this book is right

for you because it teaches more than just the language. This book attempts to teach you how to program correctly, concentrating on proper coding techniques in addition to the Visual Basic language.

## This Book's Philosophy

*Visual Basic Programming in 12 Easy Lessons* extends the traditional programming textbook tutorial by offering all the text and language syntax needed for newcomers to Visual Basic. It also offers complete program examples, exercises, questions, tips, warnings, notes, and, of course, a full-featured Visual Basic programming system.

This book focuses on programming correctly by teaching structured programming techniques and proper program design. Emphasis is placed on a program's readability rather than on "tricks of the trade" code examples. In this changing world, programs should be clear, properly structured, and well documented.

The format of this book, 12 lessons with two units per lesson, was planned to give you the most out of each sitting. You'll find that you can master each lesson in one evening taking a short break between the lesson's two units. At the end of the lessons are projects. They contain programs that use the lesson's key points and the also feature a unique line-by-line description of the program's code.

## A Note to the Instructor

If you're an instructor using this book for your class, you'll find that its inclusion of the Visual Basic language lets the entire class participate on the same level, using the same language version for their programs. When you demonstrate the writing, editing, and debugging techniques, you'll know that your students will be using the same language that you use in class.

Each unit offers numerous questions and exercises that provide a foundation for classroom discussions. The answers to all the questions and exercises are on the enclosed disk. In addition, each unit contains one or more "extra credit" programming exercises that you can assign as homework. The answers to the extra credit exercises don't appear on the disk.

The typical semester class is divided into 15 or 16 weeks of study. A useful lesson plan that incorporates this book would spend one week on each lesson, with four exams every four weeks. Each lesson contains two units, and you can easily cover one unit in a single classroom sitting.

Because *Visual Basic Programming in 12 Easy Lessons* becomes a part-time teacher, questioning and guiding the student as he or she reads and learns, you can spend more classroom time looking at complete program examples and exploring the theory of Visual Basic instead of taking the time to cover petty details.

# Overview

Here is an overview of this book, giving you a bird's-eye view of where you're about to head.

### Lesson 1: Welcome to Visual Basic!

This lesson explains what programming is all about and provides a brief history of programming languages. The lesson presents an overview of Visual Basic's advantages over other languages and explains the event-driven nature of Windows programs. By the second unit, you will be working with the Visual Basic environment.

### Lesson 2: Visual Basic Programs

This lesson familiarizes you with the format of Visual Basic programs. Once you master this lesson, you'll be able to distinguish all the various parts of a Visual Basic program that go together to produce the final working application the user eventually sees. In this lesson, you will create your first Visual Basic application.

### Lesson 3: Getting Down to Business

Being a Windows-development system, Visual Basic relies heavily on graphical screen objects that you, the programmer manage as you write programs. This lesson teaches you how to place the graphical objects on the user's window to create controls with which the users of the program will interact.

### Lesson 4: Data Basics

Visual Basic supports all kinds of data. This lesson teaches you about Visual Basic variables, data, and controls. You must understand the various data types possible in Visual Basic before you can work with data. You'll see how Visual Basic supports both numeric and character data as well as learn the mathematical operators that Visual Basic recognizes.

### Lesson 5: Gaining Control

The messages that you place in programs that help you update those programs later are called *remarks*. It's vital that you learn the importance of proper program documentation early. Of course, interacting with users is extremely important also, and message boxes are Visual Basic's primary means for asking the users questions and getting answers. In addition to the remark and message box processing, this

lesson explains the looping elements of the language. By repeating sections of a program, the program can perform multiple calculations on a large series of data values.

### Lesson 6: Combining Code and Controls

Now that you've mastered most of Visual Basic's user interaction controls, this lesson takes you the next step of the way by demonstrating how to create lists of data and controls that greatly improve the power of the programs that you write. Once you introduce lists into a program, the check boxes and option buttons give your users the means by which they can interact with those lists of data.

### Lesson 7: Functions and Dates

Visual Basic supplies a large number of math, character, date, and time functions. These functions eliminate tedious coding on your part that you would normally have to do. By providing pre-written function routines, Visual Basic lets you concentrate on the user-specific portions of the applications that you write.

### Lesson 8: Modular Programming

Once you've mastered Visual Basic's built-in function routines, it's time to learn how to write your own functions that you can add to your own growing library of Visual Basic routines. As you write more and more Visual Basic programs, you will write routines that you can use in more than one program. Thus, as you build your Visual Basic library of programs, you'll be able to reuse parts of programs that you've already written and speed all subsequent program development.

### Lesson 9: Extending Data and Programs

Your computer would be too limiting if you couldn't store data to the disk and load that data back into your programs. Disk files are required by most real-world applications. The units in this lesson describe how Visual Basic processes disk files and teaches you the fundamental principles needed to effectively work with disk files.

### Lesson 10: Making Programs "Real World"

By adding menus to your programs, you give additional control to the users of your applications. Most Windows programs support the use of menus and the programs that you write with Visual Basic support the standard Windows menu system. In addition to adding menus, this lesson also teaches you how to track timer controls so that you can write real-time programs that respond to the passing of time.

### Lesson 11: Spruce Up your Programs

Visual Basic supports the use of printed reports that you can create with the commands that you learn in this lesson. Not only will the addition of the printer add to your program power, but you can also add spice to your programs by drawing graphics on the program screens that you create to offer eye-catching pictures that capture the user's attention.

### Lesson 12: Getting Exotic

This lesson extends your fundamental knowledge of the Visual Basic programming system by teaching you how to work with additional user controls called scroll bars and grid controls. Scroll bars give users visually-moving controls over data values and input. Using the grid, users can see tables of data that you present in a spreadsheet-like format. You'll also learn ways to respond to mouse movements and clicks. By this lesson's end, you will have mastered most of Visual Basic's fundamental programming techniques. Due to the complex nature of programs that you will begin to write by this lesson, the book ends with a discussion of Visual Basic's interactive debug sessions that shows you how to eliminate problems that might creep into the programs that you write.

## This Book's Disks

This book contains the Visual Basic programming system called the *Visual Basic Primer*. Visual Basic is made by Microsoft, a world-famous company known for its Windows operating environment. The Visual Basic programming system comes with an integrated editor, debugger, and program designer with which you can write fully-working Windows programs.

---

**Note:** The second unit of Lesson 1 explains how to install the Visual Basic programming system on your computer.

---

The disks also contains all the code in all of this book's programs. In addition, the disks contains answers to all review questions and exercises at the end of each unit, except for the extra credit problems. The answers are organized by lesson and are in a directory named \ANSWERS.

## Conventions Used in this Book

This book uses the following typographic conventions:

- Code lines, variables, and any text that you see on-screen appear in monospace.
- Placeholders in statement syntax explanations appear in *italic monospace*.

- New terms appear in *italic*.
- Filenames in regular text appear in uppercase, such as MYFILE.DAT.
- Optional parameters in statement syntax explanations are enclosed in flat brackets ([]). You don't type the brackets when you include these parameters.
- Menu commands appear like this: File Open. This command means to select the Open option from the File menu.

The following items also appear throughout this book:

---

*Definition:* Definitions of new terms often appear in the paragraph in which the term first appears.

---

---

**Note:** When further thought is needed on a particular topic, the note icon brings extra information to your attention.

---

**Tip:** A tip shows you an extra shortcut or advantage possible with the command you just learned.

**Warning:** Sometimes you must take extra care when trying a particular command or function. Warnings point out dangers before you encounter the dangers yourself.

---

Sidebar: In addition, you'll find several sidebars with useful information that is related to the topic at hand.

---

***Concept:*** Concepts, located at the beginning of each major section, provide a succinct overview of the material in that section.

***Review:*** Reviews, which appear at the end of each major section, recap the material you learned in that section.

***Stop and Type:*** This icon provides a description of a subsequent program listing.

***Input:*** An input icon marks a program listing that demonstrates the major concepts from the section you just finished.

***Output:*** This icon accompanies a typical output of the program or program screen.

***Analysis:*** A detailed description of the program appears after the output.

# Visual Basic in 12 Easy Lessons



# Lesson 1, Unit 1

## Introducing Visual Basic Programming

### What You'll Learn

- [lb] Why write programs?
- [lb] A brief history of programming
- [lb] The steps needed to create programs
- [lb] Resolving program errors
- [lb] The difference between syntax and logic errors
- [lb] The transition from BASIC to Visual Basic

This book is more than the usual computer disk and text combination. The book that you now hold contains a working copy of Microsoft Visual Basic For Windows, a complete Windows programming development system. Along with Visual Basic, you get a fantastic (if I may say so myself) introduction

to programming within the Visual Basic environment. In addition to Visual Basic and the text, you also get every program listed in this book, so you can spend less time typing and more time learning.

If you have programmed in other languages, hold on to your hats Visual Basic is unlike other programming languages. The primary difference is that Visual Basic is *fun* to work with. The word *fun* simply does not apply to many other programming languages. With Visual Basic, you create most of your programs by clicking and moving your mouse. Instead of programming, you really *build* programs. Visual Basic is one of the few programming tools with which you can efficiently design your program while you create your program.

This unit describes what programming is all about. You will learn a little about the history of programming, especially the movement of BASIC through computer history. (Visual Basic finds its roots in the more traditional BASIC language.) Get ready to learn how Visual Basic streamlines your programming life.

---

**Note:** This unit spends a lot of time describing the background and importance of programming and early programming languages. This background serves two purposes. First, by learning about the history of programming, you will have a better idea where today's programming tools are headed. Second, you will appreciate more fully the incredible power, ease, and versatility that Visual Basic offers over other programming languages. Visual Basic's capabilities were beyond anyone's imagination just a few years ago.

---

## Why Write Programs?

***Concept:*** If you want your computer to do exactly what you want it to do, you must write a program.

A computer does nothing on its own. In fact, a computer is a dumb machine with no intelligence whatsoever. Despite what you might read in science fiction stories, a computer does nothing more than blindly follow instructions supplied by a programmer. Computers cannot think.

---

*Definition:* A *program* is a set of instructions that tells the computer exactly what to do.

---

When someone buys a computer today, the computer sits on the desk doing nothing until he loads a program into the computer's internal memory and starts running the program. Just as a VCR does not record shows on its own without being programmed to do so, a computer requires detailed instructions found only in programs.

Suppose that you own rental properties and want to use your computer to track your tenant records. Your computer will not help you out in any way until you load and run a rental property program. Where do

you find such a program? There are two ways to obtain programs for computers. You can

Buy one and hope that the program does exactly what you want it to do.

Write your own program.

It's much easier and faster to buy a program that you need. Thousands of programs are on the market today. In fact, there are so many programs out there that you might not see the need for writing your own programs.

If you can find a program that does exactly what you want, you are ahead of the game. If you find a program that meets your exact requirements, you should buy that program because purchasing a program is often less expensive and much quicker than writing the same program yourself or hiring programmers to write it for you.

Think about this for a moment, though: If there are so many programs sold today that do virtually everything, why are programming languages such as Visual Basic continuing to break previous sales records each year? The answer is simple: People buy a computer so that the computer will do jobs that they need done. Firms cannot adapt their business to a computer program. They must find programs, or write their own programs, so that the computer processes information according to the business procedures already in place. The only way to ensure that a program exactly fits the needs of a firm is for the firm to develop its own programs.

Business people are not the only ones who need custom-designed programs. No two people manage their finances exactly the same way; no two scientists need computers for exactly the same kinds of computations; and no two graphic artists need the same kinds of computer drawing tools. Although people buy spreadsheets and word processors for their general-purpose computing needs, many people require specialized programs for specific jobs.

The art of programming computers is rewarding not only from a requirements standpoint, but also on a more personal level. Programming computers is fun! Just as a sculptor looks on a finished work of clay, programmers are often proud of the programs that they write. By the time you finish this book, you will have written programs that were not available before you wrote them. When you want your computer to do something specific and you cannot find a program that does the job exactly the way you want, you will be able to design and write the program yourself.

---

Some Programs are Changeable: There is a third method for getting exactly the program that you need if you want to computerize your company's accounting records. Accounting software firms often sell not only accounting programs but also the *source code* for those programs. The source code is a listing of the program's instructions. By having access to the source code, you can take what the software company wrote and modify the behavior of the program to suit your own requirements.
By starting with a functional program instead of starting from scratch, you save programming time and money. Sadly, most non-accounting software firms do not supply the source code. Most programs sold today have been *compiled*. After compiling, the source code is translated into a locked-in executable program. The bottom line is that you cannot easily change the behavior of compiled programs. For most programs, therefore, you have the choice of buying them or writing them yourself from scratch.

---

*Definition: Code* is another name for program.

There are many ways to write programs for computers. In the next section, you'll learn how the process of entering programs progressed from switches on the front of the computer to clicking and pointing with the mouse today. The majority of programs now in use are supplied in the form of code listings, which often comprise pages of line after line of computer instructions. Visual Basic helps take the drudgery out of coding that is, writing programs. Visual Basic enables you to move elements and place graphical images on the screen with the mouse instead of requiring that you give written and detailed screen instructions as required by languages that came before Visual Basic.

**Tip:** For a complete description of the programming process, a comprehensive overview of programming languages, and a guided tour through the business of programming, check out *Absolute Beginner's Guide to Programming* (Sams Publishing, 1993).

***Review:*** No single program pleases everyone. When a company sells a program, it must be general enough to please most purchasers. Some people need programs to behave in a specific manner in order to fulfill a specific need. They must resort to writing their own programs. Luckily, Visual Basic takes a lot of the pain out of writing programs.

## A Brief History of Textual Programming

***Concept:*** Computers cannot understand just any language. You must learn a language that your computer knows before you can write programs for your computer.

*Definition:* An *application* is yet another name for program.

Many people use computers all day long for word processing, database storage, and spreadsheet analysis, without realizing what is going on behind the scenes. You must always keep in mind that computers cannot think. Your computer does not know how to be a word processor. If you want your computer to do word processing, you must supply detailed instructions in the form of a program. Only by following the detailed instructions of a word processor program that you load can your computer perform word processing.

It would be nice if writing a program is as easy as telling the computer what you want done. Many people can handle ambiguous instructions, but computers are not smart enough to understand vague requirements. Computers can only follow orders given to them, and you must supply those orders in the form of a program. Therefore, you must supply the programs that you write. Writing programs,

especially complex programs, takes time and several procedural steps. Visual Basic speeds the process of creating programs, but even with Visual Basic some programs take time to write and perfect.

---

*Definition:* A *bug* is a program error.

---

Figure 1.1 shows you the typical steps that most programmers go through when writing programs. First, you have an idea for a program. Next, you use a program-development system, such as Visual Basic, to write the program. Errors, or *bugs*, often appear in programs because of the details needed for even the simplest of programs. Therefore, you must test the program thoroughly and fix the errors. Fixing errors is called *debugging*. Once all the errors are out of the program, you have a finished application.

**Figure 1.1. Writing a program involves several steps.**

The second step, writing the source code, is the most tedious part of programming. Remember that the source code is the actual programming instructions that the computer is to follow. You will spend most of your programming time working on the source code so that you end up with a program whose instructions direct the computer in the way that you want.

There are many ways to write the source code for programs. Although today's computers are more powerful than earlier machines in terms of memory capacity and speed, today's computers are no smarter than the very first computer. In the late 1940s, programmers had to write programs for those early computers just as today's programmers must do. The difference lies in the way today's programmers write programs. Today's programming tools Visual Basic is a shining example are far more powerful. They enable you to develop programs more powerful than before and with less effort on your part.

## Where Programmers Began

The very early computers were not programmed through keyboards and mice. As a matter of fact, the first few computers did not even have keyboards! The first computers had to be programmed by routing wires from component to component. Instead of programmers, electrical engineers programmed the early computers.

Those hard-wired programming methods were simply too tedious to be productive. If a different calculation was needed, somebody would have to rewire the computer. Programmers had to have electrical and engineering experience just to make a computer do something. There had to be a way to speed up the process.

Those early computers had memory similar to the way in which today's computers have memory. The difference is that the early memory was minuscule even the largest computers had only a few hundred memory locations for data storage. Despite the short supply of memory, one of the computer experts of the time developed the idea of using the memory to hold both data and the code that instructed the computer on a task. Until that point, the "code" consisted of the hard-wired circuitry.

---

**Tip:** By putting the computer's instructions inside the memory along with data, the computer programs were easier to change because the memory could be changed. Engineers were no longer required to rewire the computer every time the program needed changing.

---

The computer scientists put a panel of switches on those early computers. Figure 1.2 shows a simple representation of those switches. The programmers would flip the switches into a series of unique On and Off states, press the Enter button, and repeat the process until a series of instructions that looked like the following "program" appeared in the program memory:

```
On Off On On Off Off
Off On

Off On On Off On On Off Off

Off Off Off On On Off On Off

On On On On Off Off Off On

Off Off On Off Off On On Off

On On Off On On Off Off On

Off Off On Off On On On Off
```

**Figure 1.2. The switch panel eliminated the hand wiring.**

---

*Definition:* The original On-and-Off programming was called *machine language*.

---

Although this On-and-Off program is virtually indecipherable, each combination of On and Off switches represents a single instruction. By combining several instructions that operated on data located elsewhere in the machine's memory, a complicated thirty-instruction program might not do anything more than multiply two numbers! Despite the difficulties involved, such a lightning-fast calculation would have been unheard of before computers. The military immediately began using computers for trajectories and other calculations.

---

**Tip:** By seeing the short history of programming, you will *really* appreciate what Visual Basic can do

when you start using Visual Basic for your programs.

---

The world of computers began moving forward with the switch panel. More memory was added, and the programs got more powerful. Sometime in the late 1940s, somebody got the bright idea of replacing the switch panel with a typewriter-like keyboard. Instead of assigning On and Off combinations to mean *Add* and *Store,* programmers could actually type the words Add and Store on the keyboard. The computer would analyze the instruction, look up the On and Off combinations needed, and set the memory switches internally.

---

Machine Language is Dead! Long Live Machine Language!: Luckily, you do not have to write programs in the On-and-Off machine language anymore. The computer languages today are much closer to spoken speech than On and Off switches can ever hope to be. Even so, today's most powerful computer, programmed with advanced programming tools such as Visual Basic, *still* recognizes only machine language.
People do not like machine language because it is too difficult to use. Computers do not like anything else. The job of all programming languages is to take the source code that you type in a programming language and to convert it to machine language so that the computer can execute those instructions.

---

## Programming Languages Improved

Once programmers got hold of keyboards, there was no stopping them. The languages grew from the primitive On-and-Off system to higher-level languages that read more like spoken text. Listing 1.1 shows an example of a short FORTRAN program from the early days of these high-level languages. Although this FORTRAN program will be cryptic to you, the more textual approach to programming, as opposed to On and Off switches, opened doors for more people to become programmers. The software industry blossomed in the 1950s, and programs went from simple calculating tools to complete business and scientific applications.

---

**Note:** FORTRAN stands for *FOR*mula *TRAN*slator. It is used primarily in mathematical and scientific programming. Although FORTRAN was one of the earliest high-level programming languages, many computer installations still use FORTRAN programs today. Much of the Visual Basic language was founded in principles of the early FORTRAN language.

---

**Listing 1.1. An early FORTRAN program.**

```
WRITE (6, 10)
```

```
 10 FORMAT('** Payroll Calculations **')
 WRITE (6, 11)
 11 FORMAT('** Enter the employee's ID, hours, and pay rate')
 TAXRAT = 0.25
101 READ(5, 102, END=900) IDEMP, HRSWRK, RATE
 102 FORMAT(I5, 1X, I3, F5.2)
 IF (HRSWRK .GT. 40) GOTO 300
*******COMPUTE REGULAR PAY
 GRSPAY = HRSWRK * RATE
 GOTO 500
 300 OVRHRS = HRSWRK - 40
*******COMPUTE OVERTIME PAY
 OTGRS = OVRHRS * RATE *
1.5
 GRSPAY = 40.0 * RATE + OTGRS
 500 TAXES = GRSPAY * TAXRAT
 PAYNET = GRSPAY - TAXES
 WRITE (6,503) IDEMP, PAYNET
 503 FORMAT('EMP: ', I3, 2X, 'NET PAY:', F6.2)
```

```
 GOTO 101

*******END-OF-JOB PROCESSING

 900 END
```

If you are unfamiliar with FORTRAN programs or with programming in any other language, you probably will not understand much of what you see in Listing 1.1. You might, however, be able to tell from some of the words, that the code performs payroll calculations of some kind. It is true that FORTRAN does not produce extremely readable code, but the commands such as IF, GOTO, and WRITE, improved programmer productivity greatly over what programmers had to do before such languages came along.

With high-level languages such as FORTRAN, the 1950s and 1960s saw an incredible distribution of new software. Programming became more accessible to more people because of the easier-to-use programming languages. The increased number of programs brought a tremendous increase in the number of computers sold. Companies, laboratories, and universities purchased computers. There was no turning back the computer era once so many people had access to so much computing power. As computer companies sold more computers, competition produced less expensive and more powerful machines. And you thought that the 1990s were exciting!

---

*Definition: Syntax* refers to the spelling and grammar of languages.

---

Even though programming languages were easier to learn, there was still much need for easier programming methods. Some Dartmouth College professors decided to write an easier programming language based on FORTRAN but with fewer details than FORTRAN required. Those professors developed BASIC, the language that enabled students to write programs because of its easier format and less restrictive syntax than FORTRAN s.

---

**Note:** BASIC is an acronym for *B*eginner's *A*ll-purpose *S*ymbolic *I*nstruction *C*ode. The acronym's meaning is almost as long as and much more difficult to remember than the BASIC language itself.

---

Listing 1.2 shows the BASIC equivalent of the FORTRAN program that you saw in Listing 1.1. To programming newcomers, this BASIC listing might not be significantly easier to understand than the FORTRAN code. Nevertheless, the BASIC language is a little cleaner and slightly less cryptic than its FORTRAN predecessor. Although programming still took effort, BASIC took some of the rough edges off programming, and it brought more people to the programming craft.

**Listing 1.2. A BASIC program.**

```
10 PRINT "** Payroll Calculations **"

20 PRINT
"** Enter the employee's ID, hours, and pay rate"

30 TAXRAT = .25

40 INPUT IDEMP$, HRSWRK, RATE

50 IF (IDEMP$ = "END") THEN GOTO 160

60 IF (HRSWRK > 40) GOTO 70

70 REM *******COMPUTE REGULAR PAY

80 GRSPAY = HRSWRK * RATE

90
GOTO 110

100 OVRHRS = HRSWRK - 40

110 REM *******COMPUTE OVERTIME PAY

120 OTGRS = OVRHRS * RATE * 1.5

130 GRSPAY = 40 * RATE + OTGRS

140 TAXES = GRSPAY * TAXRAT

150 PAYNET = GRSPAY - TAXES

160 PRINT "EMP: "; IDEMP$; "NET PAY:";
PAYNET

170 GOTO 20
```

```
180 END
```

Well into the 1980s, the rate at which the number of programmers grew remained high. Despite all new programmers, the programming tools themselves really did not advance much. Many people developed new programming languages. Despite their "new and improved" claims, most of the languages retained the textual procedural quality that FORTRAN and BASIC offered.

Large-scale programming tool improvements did not occur until a hardware breakthrough occurred: NASA's space efforts developed the microchip, and microcomputers were born. Now, instead of new and potential programmers, desktop computers opened the door for millions of would-be programmers. These programmers demanded easier programming tools.

---

*Definition: QBasic* is a recent BASIC language clone.

---

Efforts were made to improve the friendliness of programming languages like BASIC. New versions of BASIC, such as QBasic, enabled programmers to write more powerful programs with less effort. Programmers could also write programs that had a less rigid style than those before. Listing 1.3 shows a QBasic version of the payroll calculation shown in the previous two listings. As you can see, the language is getting less strict and more free-form.

**Listing 1.3. A QBasic program.**

```
TaxRate = .25

PRINT "** Payroll Calculations **"

PRINT "** Enter the employee's ID, hours, and pay rate"

INPUT IdOfEmp$, HrsWorked, Rate

DO UNTIL (IdOfEmp$ = "END")

 IF (HrsWorked <= 40) THEN

 '
*******Compute regular pay

 GrossPay = HrsWorked * Rate
```

```
 ELSE
 ' *******Compute overtime pay
 OverTimeHours = HrsWorked - 40
 OverTimeGross = OverTimeHours * Rate * 1.5
 GrossPay = 40 * Rate + OverTimeGross
 END IF
 Taxes = GrossPay * TaxRate
PayNet = GrossPay - Taxes
 PRINT "Emp: "; IdOfEmp$; "Net Pay:"; PayNet
 INPUT IdOfEmp$, HrsWorked, Rate
LOOP
END
```

## You Run Programs to Produce Output

---

*Definition:* Computers generate *output* on screens and printers.

---

Just as a recipe has a result, the cooked meal, instructions in a program produce an important result. The result of the programming effort is the finished program that makes the computer perform a directed task, such as payroll processing. After entering a program or loading a program from the disk, you must tell the computer to run, or execute, the program. The computer then follows the program's instructions and produces the output.

If you entered the code in Listing 1.3 into the QBasic programming language, the following output might result:

```
** Payroll Calculations **

** Enter the employee's ID, hours, and pay rate

? KL823, 40, 9.25

Emp: KL823 Net Pay: 277.5

? PO341, 35, 10


Emp: PO341 Net Pay: 262.5

? LP543, 40, 10

Emp: LP543 Net Pay: 300

? END, 0, 0
```

---

**Note:** The previous QBasic code requires that the user tell the program that there are no more employees to process by typing END as an employee ID and zero amounts for the hours and pay rate.

---

---

**Tip:** This output would be different if the user entered different data, because the program would calculate different payroll results.

---

---

*Definition: Input* is the data that the user enters.

---

As you can see from the execution of this program, the program takes the user's input and calculates net pay amounts. Input can come from sources other than the user. Programs might get input from a disk file, a modem connected to the telephone, or from other programs running along with yours. Users do not

always see the output, either. Programs often perform calculations and data manipulation and send the output of that processing to disk files or to other computers over modem lines.

Figure 1.3 illustrates the program flow from writing the program to generating output, such as the payroll output you see here.

**Figure 1.3. The program's instructions produce the output.**

---

**Warning:** The output you get after typing with a word processor would be the text on paper or on the screen. The output from other programs might be as diverse as payroll reports, calculations, or graphics. The program's instructions determines what the program produces.

---

## Exterminating Bugs Is Not Trivial

---

*Syntax* is the grammar of a language.

---

Bugs are as frustrating in computer programs as they are in computer programmer's houses. Rarely will a program that you write work the first time you run it. There are two kinds of errors: *Syntax errors* sounds like "sin tax" and is worse in many ways and *logic errors*. Since you do not know the Visual Basic programming language yet, this simple phrase demonstrates the two kinds of errors that can appear in your code:

There are two errrors in this sentence.

What are the two errors? I'll wait...

The first error, a syntax error, is easy to spot. errrors is misspelled. The logic error takes more time to find. The logic error is that the entire sentence is logically wrong there is only *one* error, the syntax error errrors.

When you begin to write programs, this example sentence may come back to haunt you a bit. Syntax errors are easy to catch because Visual Basic catches all of them for you. In other words, if you type Tezt when Visual Basic expects you to type Text, Visual Basic immediately displays an error message box in the center of your screen. Visual Basic is extremely stubborn when it comes to syntax errors because it refuses to execute your program until you remove all the syntax errors in the code.

When you make a logic error, however, Visual Basic cannot help you out. For example, if you were writing a payroll calculation routine and accidentally subtracted $50 from everybody's paycheck, Visual Basic has no way of knowing that you should not subtract that amount. If you tell Visual Basic to subtract the amount using correct syntax, of course Visual Basic subtracts the amount. Only until your

employees beat your door down will you know that a logic error took place.

---

**Tip:** Thoroughly test your programs before you distribute them or use them for important work. Run the program using all combinations of input to find and correct as many logic errors as possible.

---

## Graphical User Interfaces Changed Everything

---

*Definition: GUI* stands for *G*raphical *U*ser *I*nterface.

---

A challenging programming problem arose when advanced GUI visual environments, such as Microsoft Windows, appeared in the 1980s. GUI environments require more advanced and difficult programming efforts than text-based computing environments. Although users and programmers wanted easier tools, GUI environments demanded more complex programming tools.

---

*Definition:* An *event* can be a Windows mouse click, a key press, a menu selection, or an internal Windows activity.

---

It is relatively easy to produce the text-based output shown earlier. It is much more difficult to produce Windows-like screens such as the one shown in Figure 1.4. The user does not have to enter the ID, hours worked, and amounts in any preset order. Neither does the user have to press the three command buttons at the right of the screen in any order. In Windows programs, a specific event determines the next course of action. A Windows program is called an event-driven program as opposed to a text-based procedural program. In Windows, the user determines what happens next by triggering an event to which the Visual Basic program responds.

**Figure 1.4. Users randomly control event-driven programs.**

***Review:*** Computer programming has progressed a long way from the wiring panels of the early days. Once keyboards were added, what programmers could do grew by leaps and bounds. Actually, it was not until the development of Visual Basic that another great leap in programming tools arrived. The power of programs that you can create with Visual Basic, given the simplicity of Visual Basic's environment, produces as much computer programming advantage today as the keyboard provided over the switch panel. As Yogi Berra once said, "It ain't bragging if it's true!"

# From BASIC to Visual Basic

***Concept:*** As Windows programming tools are developed, their complexity seems to increase. The Microsoft Windows environment requires much more complicated programming efforts than DOS-based non-Windows programs require. Visual Basic bucks the trend by providing a simple approach to writing Windows programs while retaining all the beginning and advanced programming language instructions found in QBasic.

---

*Definition:* Text-based programs are often called *procedural programs*.

---

Procedural languages such as FORTRAN and QBasic simply don't work well for a GUI environment. The problem actually, the primary advantage for users of GUIs is the event-driven processing explained in the previous section. The user does not always do the same thing in the same order. That is, a user might want to select from a pull-down menu, click a mouse button, type text, or compute a mathematical answer, and he might do those tasks in virtually any order. Text-based procedural programming languages essentially required that the program dictate the order of the user's actions. Windows requires a different approach. The traditional programming languages cannot handle the GUI approach well at all.

It is possible to write event-driven programs in procedural languages. The majority of Windows programs in use today were written in the C language, a procedural language more similar in style to FORTRAN than to Visual Basic, although C and FORTRAN differ greatly in their approach and syntax. Nevertheless, C programmers face great challenges when using C for Windows programs because languages such as C are procedural whereas Windows is event-driven.

Several powerful programming solutions have been devised that aid Windows programmers. A relatively new programming concept called *object-oriented programming* (or *OOP* for short) better lends itself to Windows-like event-driven programming than languages such as FORTRAN, BASIC, or C. Even OOP, however, puts a strain on its programmers that today's busy and backlogged programming departments do not have the resources to handle.

The big problem is that as computers get easier for users by supplying graphical environments such as Windows, the programs that the users use become harder to develop. Therefore, as the demand for these GUI programs increase, so does the backlog of programs that need to be written.

Microsoft introduced a new programming language, Visual Basic, a few years ago. The advantages of Visual Basic became immediately apparent. They are

- [lb] Visual Basic is based on the QBasic programming language, so programming newcomers find Visual Basic friendlier than other languages.
- [lb] Visual Basic is designed from the ground up to be a Windows programming language. Inherent in Visual Basic's fundamental design is the event-driven program concept. As a matter of fact, textual procedural-based programs are difficult to write using Visual Basic.
- [lb] Visual Basic is graphical in its programming approach. You can literally create a complete and working Visual Basic program by moving picture icons on and off the screen without ever writing

one command in Visual Basic's programming language.

The great thing about Visual Basic is that the program looks almost exactly like the output screen that results. In other words, to design and write the simple program that produced Figure 1.3, you would place text, buttons, and lines onto the screen, using the visual tools supplied with Visual Basic, until your "program" looked like what you wanted your finished Windows program to look like. Such placement of visual elements would takes pages and pages of typed instructions using a traditional procedural programming language.

---

**Note:** After the success of Visual Basic for Windows, Microsoft developed Visual Basic for DOS. Visual Basic for DOS found extremely limited success, however, and is now extinct.

---

Before you can learn to program with Visual Basic, you must get a feel for the tool itself. The second half of this lesson, Unit 2, introduces you to Visual Basic itself. You will learn how to install the Visual Basic disk that comes with this book, start Visual Basic, and manipulate the screens of Visual Basic. Once you master the mechanics of Visual Basic, the second lesson dives right into programming by walking you through your first full-functional Visual Basic application from design to execution.

---

Visual Basic is Event-Driven and Procedural: Don't think that the days of program listings are gone forever. As a matter of fact, one of the most important components of Visual Basic is its procedural BASIC-like programming language underneath the visual environment. When you begin to extend your Visual Basic programming by developing more powerful programs, you will need to combine the visual elements of the language with the QBasic-like routines.

---

---

**Warning:** This unit ends here without describing the details of Visual Basic because Visual Basic's details are the *rest* of this book's job. Now that you have a fundamental grasp of programming and the Windows programming challenges, you are ready to begin taking a rewarding tutorial of Visual Basic for the remainder of this book. In twelve easy lessons, you will be a Visual Basic master!

---

***Review:*** In many ways, writing programs the old-fashioned way that is, writing page after page of text instructions that the computer will eventually follow is somewhat like a long and arduous recipe. The cook starts at the first instruction in the recipe and must be careful not to skip steps. Such listings are vital, and although the recipe might throw in a picture or two, the cook must follow the recipe sequentially if the finished meal is to taste good.

Writing programs with Visual Basic is much more akin to using a VCR than following a step-by-step recipe. In all fairness, though, even the most complicated of VCRs are easier to program than complex Visual Basic applications. Visual Basic, though, is still one of the easiest and fastest tools you can find to develop Windows applications.

---

**Note:** Most of the parts in this book end with a Review as well as a Stop & Type section, which enables you reinforce the section's topics with a hands-on programming exercise. The primary purpose of this first unit has been to teach you the basics of programming from a historical perspective. Therefore, there are no hands-on topics for this unit.

---

# Homework

## General Knowledge

1. What is a program?
2. If you attempt to use a computer that has no running program, what's the result?
3. What are two ways to obtain programs for your computer?
4. What is the advantage of writing your own programs?
5. What is the disadvantage of writing your own programs?
6. True or false: Some vendors sell programs that you can customize to suit your own needs.
7. What is code?
8. What is a bug?
9. What are the two kinds of errors called?
10. If you misspell a Visual Basic command, what kind of error have you violated?
11. What kind of error does Visual Basic find for you?
12. What is the hardest kind of error to find?
13. How were the earliest computers programmed?
14. How did adding switches to the front of early computers improve the programmability of those computers?
15. The earliest programming language, called *machine language*, consisted of combinations of On and Off states produced by the switch panel. What did these On and Off combinations represent?
16. The addition of which piece of hardware made it possible for more people to access computers and program them?
17. What do you call the code entered by the programmer?
18. At the lowest level, what does every high-level programming language become before the computer can understand the language?
19. What was the name of one of the earliest high-level programming languages that is still in use today for scientific and mathematical programming?
20. What language did some Dartmouth professors develop to overcome the fears that some of the

existing programming languages produced?

21. What does BASIC stand for?
22. What kind of programming languages lend themselves well to text-based DOS environments?
23. What kinds of programming languages lend themselves well to Windows-like environments?
24. What does GUI mean?
25. Give two examples of an event.
26. What challenges must an event-driven program overcome?
27. True or false: Visual Basic does not contain any procedural programming tools because none are useful for GUI environments.
28. True or false: A Visual Basic program often looks like its own output.

## Extra Credit

1. A programmer does not sit down in front of a computer and start typing a finished program although Visual Basic makes programming almost that easy. Describe the steps necessary to produce a program.
2. Why should programmers thoroughly test the programs that they write?

---

**Note:** Because this unit was descriptive, it contains no What's the Output?, Find the Bug, or Write Code That... sections.

---

# Visual Basic in 12 Easy Lessons



# Lesson 1, Unit 2

## First Look at Visual Basic

### What You'll Learn

- [lb] The Visual Basic Primer disk
- [lb] Installing the Visual Basic Primer disk
- [lb] Starting and stopping Visual Basic
- [lb] The Visual Basic environment
- [lb] The five windows
- [lb] The top of the screen

This unit helps you install the Visual Basic programming system that comes with this book. You will find that the installation of Visual Basic is easy because Windows does most of the work for you. If you've installed other Windows programs before, you will have no trouble with installing Visual Basic.

---

**Warning:** This book assumes that you have used Windows before. Although you might not be a Windows guru, you should be comfortable with starting Windows, using the mouse, selecting from Windows menus, and so on.

---

---

**Tip:** If you want a good review of Windows fundamentals, get a copy of Alpha Books' *The Complete Idiot's Guide to Windows* (ISBN 1-56761-175-3) and read it before going further with *Visual Basic Programming in 12 Easy Lessons*. You'll be glad you did.

---

Before installing the program, this unit takes a few moments to describe exactly what you get with this book's bundled disk.

## The Visual Basic Primer Disk

***Concept:*** With this book, you get a Visual Basic programming system with which you can learn Visual Basic and create your own Windows programs.

The Microsoft Corporation has released several versions of Visual Basic. When version 1.0 hit the market, Visual Basic changed the way in which people viewed Windows programming. Even in its first release, Visual Basic was a powerful programming tool for introductory Windows programmers. With its visual placement of program and screen elements, you will literally draw programs instead of writing them. With every release of Visual Basic, Microsoft keeps adding more functionality and programming power.

This book contains the Visual Basic version 2.0 programming system. This version of Visual Basic is called the Visual Basic Primer Edition. It includes everything in the original and really expensive Visual Basic version 2.0 Standard Edition except for these three modifications:

- [lb] You can create and run, but not compile, stand-alone Visual Basic programs.
- [lb] You can add a maximum of one form to your application.
- [lb] There is no online help available.

---

*Definition:* A *compiler* creates a stand-alone program.

---

When you buy a copy of Visual Basic and pay a whole lot more money than you did for this book and disk! you get everything you get here plus a compiler. This copy of Visual Basic enables you to create

the same programs as the regular copy, but you must run those programs from within the Visual Basic environment. In other words, when you want to run the program that you write, you have to run Visual Basic first and use the Visual Basic menus to load and run your program.

For most beginning and intermediate programmers, the second modification is not much of a limitation. You can add only one form to a program. One form is probably all you would be using anyway. It sometimes takes a fairly complicated Visual Basic program to require a second form.

The third modification would *really* hinder most people, but not you. This book offers a better tutorial and reference guide than you would ever get with an online reference.

What *can* you expect with the Visual Basic Primer disk? A version of Visual Basic that includes all the tools that you need to learn the in s and out s of Visual Basic. You will add advanced controls to your Visual Basic programs. You will be write programs that behave just like the major Windows applications you use daily. You will learn what programming is all about by mastering the Visual Basic programming language.

***Review:*** The disk that comes with this book contains the Visual Basic Primer programming system, which includes everything you need to create complete Windows applications.

## Installing the Visual Basic Primer Disk

***Concept:*** Install the Visual Basic Primer programming system on your computer's hard disk.

Installing the Visual Basic Primer is simple. The installation described in this unit assumes that you want to install the Visual Basic Primer on your hard disk drive named C and that you are installing from the floppy disk drive named A. Change the drive names accordingly if you want to install to or from different drives.

Follow these steps to install the Visual Basic Primer on your system:

1. Start Windows.
2. When the Program Manager appears, select the Program Manager File Run command (point to the word File on the menu bar, click the mouse, and then click Run). Your screen will look something like the one in Figure 2.1.

**Figure 2.1. Getting ready to install Visual Basic.**

---

**Warning:** Almost every Program Manager displays different program group icons because almost everybody uses a different configuration of software packages. Therefore, your screen might hold a set of icons completely different from what is shown in Figure 2.1. The dialog box shown in the upper-left-hand corner is what matters.

---

3. At the Command Line prompt, type **A:\SETUP** either uppercase or lowercase is acceptable and press Enter. The disk in drive A will spin for a few moments, and the installation program will

display an installation message box on the screen telling you that the installation process has begun.
If you get an Application Execution Error when you type the installation setup command, press Enter and check the Command Line prompt that you typed to make sure that you spelled everything correctly. Make sure that there are no spaces in the command.

4. When you see the Welcome message box, you know that you're on your way to a successful installation. At the Welcome message box, press Continue to tell the installation program that you are ready to complete the process.

---

*Definition: Default* is the value used if you do not type a different value.

---

5. As shown in Figure 2.2, Visual Basic prompts you for the exact location where you want to install the program.
By default, the Visual Basic Primer program installs itself in a directory named VBPRIMER. VBPRIMER is a good name for the directory, so I suggest that you keep this name.

**Figure 2.2. Visual Basic wants to know where you want the program installed.**

6. Click the Continue button to continue the installation.
When Visual Basic notices that the VBPRIMER directory does not exist, Visual Basic warns you that the directory does not exist with a message prompt. This is a silly message box because, of course, the directory does not exist! If the directory existed, you would already have Visual Basic Primer installed on your computer. Therefore, when Visual Basic tells you that the directory does not exist, mutter under your breath, "No kidding," and press the Create Directory button.
7. You will see one more message box that you can ignore. When Visual Basic displays the message box shown in Figure 2.3, press OK and don't worry about what the message box says. The message is a carry-over of the official read *expensive* version 2.0 release of Visual Basic, and this message box does not apply here.
You need about two megabytes of free disk space for Visual Basic and the applications that you create with this book. Assuming there is enough disk space to install Visual Basic, the installation program proceeds to complete the installation. You will see a percentage gauge increase as the installation progresses.

**Figure 2.3. You can completely ignore this message box.**

---

**Warning:** Depending on your installation of Windows, one or two additional message boxes might appear that tell you of particular files that the installation program needs to overwrite. You can click the OK button to accept the information and move on. Also, there is another error message box that might appear that indicates an incorrect number of bytes were counted during installation. As bad as this message sounds, if you click Ignore one of the few times that you can ignore error message boxes the installation continues successfully.

---

8. During the installation, Visual Basic creates a new program group called Visual Basic 2.0 for the

Program Manager's Visual Basic Primer icon. When the installation is complete, you are given the choice to start Visual Basic or to return to Windows. For now, return to Windows so that you can learn the standard startup method.

---

*Definition:* A *control button* appears in the upper-left corner of every window.

---

When you return to the Windows Program Manager, you will see the Visual Basic 2.0 window left open by the installation program. For now, close the window by double-clicking the control button to return the Program Manager to its regular state. (A state of confusion....)

You have now successfully installed the Visual Basic Primer programming system, but there is one last thing you must do to complete the installation of this book's disk. This book's disk also contains all the book's complete working applications. The applications are in a directory called PGMS.

Assuming that you have installed the Visual Basic Primer as described, follow these steps to copy the programs to the VBPRIMER directory so that you can load the programs easily as you work through them:

1. Start the Windows File Manager. You will find the File Manager icon in the Main program group.
2. Click the C: drive to activate C if it is not already the active drive.
3. You'll see the vbprimer directory file folder in the list of directories in the left panel. Click vbprimer to open the file folder.
4. Press F8 to open the Copy dialog box. Type **A:\PGMS*.*** and press Enter to copy all the working applications to the VBPRIMER directory.
5. Close the File Manager by double-clicking the control button in the upper-left hand window. You have now successfully copied all the applications to your hard disk along with Visual Basic.

***Review:*** Your computer now has this book's Visual Basic program installed. You are ready to start the program and master the environment.

## Starting and Stopping Visual Basic

***Concept:*** To write a Windows program with Visual Basic, you have to start Visual Basic. Before exiting Windows, you must exit Visual Basic.

Your Window's Program Manager might have placed the Visual Basic 2.0 icon virtually anywhere on the screen. Look through your icons in the Program Manager until you find the one labeled Visual Basic 2.0. Double-click the icon; the Visual Basic program group appears with a single icon labeled Visual Basic Primer.

Double-click the Visual Basic Primer icon to start Visual Basic. Figure 2.4 shows the Visual Basic opening screen.

**Figure 2.4. The Visual Basic startup screen.**

As you can see from Figure 2.4 and probably from your own screen, the underlying Program Manager icons often peek through to the Visual Basic programming area. Go ahead and exit Visual Basic, and you will learn a way to keep those Program Manager icons from coming through.

Visual Basic is consistent with most Windows programs in that its menus and commands are similar in many ways to Excel, Access, and other programs with which you might be familiar. To exit Visual Basic, select the Exit command from the File menu. When you do this, Visual Basic quits and you are back at the Program Manager. If the Visual Basic program group is still open, double-click its control button to close the open window.

If you want to keep the Program Manager program group icons from showing through to the Visual Basic desktop, click the Program Manager's Options Minimize on Use command. Display the Options menu once more to see that there is now a checkmark next to the second option. The checkmark means that the Program manager shrinks to a small icon when you start any Windows program. The next time you start Visual Basic, the Program Manager icons will no longer get in your way.

---

**Tip:** Here's a great shortcut that even many advanced Windows people do not know: Open the Visual Basic 2.0 icon from the Windows Program Manager. With the Visual Basic Primer icon still highlighted, select File Properties from the Windows menu bar. Press Alt+S to move the text cursor to the Shortcut Key prompt. Press the letter **V** so that the Shortcut Key prompt changes to Ctrl+Alt+V. Press Enter or click the OK button to close the Program Item Properties dialog box. From now on, you will not even have to open the Visual Basic 2.0 program group when you want to start Visual Basic. Simply press Ctrl+Alt+V from the Windows Program Manager; Visual Basic will start immediately.

---

---

**Warning:** Always exit Visual Basic before you shut off your computer or you might lose part or all of the program that you are writing.

---

***Review:*** Unless you add a shortcut keystroke to start Visual Basic, starting Visual Basic requires only that you double-click the Visual Basic Primer icon inside the Visual Basic 2.0 Program Manager group. To exit Visual Basic and return to Windows, you can select File Exit.

## The Visual Basic Environment

***Concept:*** Before you learn how to write programs with Visual Basic, you must learn all about the Visual Basic screen. Some people think that the Visual Basic screen is confusing at first. They are wrong. The Visual Basic screen is confusing not only at first but also the *whole* time you use it! Actually, Visual Basic's screen is more busy than it is confusing. Once you learn how to manage the parts of the screen, however, you will feel much more comfortable using Visual Basic.

Being an effective Visual Basic programmer means knowing how to rearrange the Visual Basic screen

when needed. Visual Basic does not really look like every Windows program. For example, Microsoft Word is one of the most powerful and most used Windows programs. When you use Word, you basically work within one gigantic window. You can open a second document window and resize the two windows to add more screen elements, but most of the time you work within a single document window.

In Visual Basic, you work with several open windows most of the time. There are several windows sometimes called *window panes* in this book open, and you will often need information from each of the windows. Therefore, you should get familiarized with the screen and its components early on.

***Review:*** Learn all you can about the Visual Basic environment now so that you can concentrate on the language and controls later.

## The Five Windows

***Concept:*** The Visual Basic environment contains several windows with which you will work as you build applications.

Figure 2.5 illustrates the major parts of the Visual Basic screen. You might not understand all the components of the screen just yet, but learn the names of the screens now so that you will move right along later when you begin learning how to program Visual Basic.

**Figure 2.5. The elements of the Visual Basic screen.**

---

**Warning:** Actually, Figure 2.5 does not look exactly like your screen probably will look when you first start Visual Basic. You will soon learn how to rearrange the screen so that the screen can look just like the screen in Figure 2.5. Figure 2.5 looks the way it does so that you can see all the major parts of the screen.

---

Table 2.1 describes each of the five primary windows of Visual Basic. Although you cannot understand all of the descriptions at this point, try to familiarize yourself with the descriptions so that the windows will not be so foreign later when you learn how to program in Visual Basic.

**Table 2.1. The primary windows inside the Visual Basic environment.**

| *Window* | *Description* |
|---|---|
| Form | Contains the background for the Windows program that you are writing. You draw and place items on the form that your program's user will eventually see and interact with. If you use a Windows word processor, the form would hold the document that you are editing. Although not every Visual Basic program requires forms, most do because most Visual Basic programs exist to display information for, and retrieve information from, the user. |

| | |
|---|---|
| Toolbox | The Toolbox window contains your tools. That might seem obvious, but you need to know that the tools of Visual Basic are more often called *controls*. The toolbox is where you will find controls that you will place on the Form window as you create a Windows program. For example, when you need to ask the user for text, you will select a text box control from the toolbox and place that text box on the form. |
| Project | A Visual Basic Windows program often contains several different kinds and types of files that all work in unison to form the single running application. The Project window contains the list of all the files used in the current application. Given the common Visual Basic terminology, a Visual Basic application is generally called a *project*. The Project window contains the contents of the project. The Project window is simply a description of the files but the files all reside separately on the disk. |
| Properties | The Properties window describes every individual element in your application. For example, there is a Properties window for a project's form because the form contains properties such as color and size. As you place controls from the Toolbox window onto the Form window, each of those controls has its own properties. Although any one Visual Basic program might have several elements with properties, there is only one Properties window. When you want to see the properties of a different form or control, you change the Properties window to display another set of properties. |
| Code | Unlike most other programming languages, you do not have to write much code as you develop applications in Visual Basic. The more advanced the application needs to be, the more code you will have to write to tie things together. The visual parts of Visual Basic, however, eliminate much of the code that you would have to write if you were still working in a text-based environment. Although you should not expect to understand anything just yet, Figure 2.6 shows a Code window that contains a fairly complex routine. The code in the Code window is the program's source code, which you learned about in the previous unit. When the user runs the program, Visual Basic and your computer interprets that source code and executes the instructions in the source code. |

**Figure 2.6. A Code window with lots of code.**

---

**Note:** Much of the time, the code inside the Code window contains setting and retrieval instructions for form controls. For example, if you need to check whether a user clicked a command button or typed a response, you can use code to check for the click.

---

As with most windows used inside Windows applications, you can move, resize, and close the five windows. Use the mouse to make working with windows simple.

---

*Definition: Maximize* means to increase a window to its largest size.

---

For example, when you first start Visual Basic, the Form window hides the other windows. Usually, the Form window is the largest window because the Form window is the user's background. You can

maximize the Form window by double-clicking the mouse on the Form window's title bar or by clicking the maximize button in the window's upper-right hand corner. Double-click the Form window's title bar now to maximize the window. When you do, there is nothing left on the screen.

Obviously, there is a way to see the menu and the other windows. Press Alt to get the top of the screen back so that you can see the menu bar and toolbar again. Display the Window pull-down menu, and select the Project window to see the Project window. Click the Project window's View Code button to see the Code window.

---

**Tip:** There are two other ways to display the Code window. You can select View Code to see the window. You can also press F7 to display the code.

---

Click anywhere on the white portion of the Form window. Clicking any window activates that window, highlights its title bar, and makes all its commands and menus available. When you activate the Form window, the Project window and Code window hide behind the form, but you can get them back by following the descriptions just offered.

To see the form s Properties window, press F4 or select Window Properties to see the Properties window. Move the mouse cursor over the Properties window and click and hold the mouse button. You can now move the Properties window by dragging the mouse across the screen. When you let up on the mouse button, Visual Basic anchors the Properties window at that point.

Try resizing the Properties window. Move the mouse cursor to any edge or corner of the Properties window. The mouse changes to a double-pointing arrow. By dragging the mouse, you can resize the window.

When you are ready to close a window, the easiest way is to double-click its control button. However, if you click the control button once, you will see the window s control menu, such as the one shown in Figure 2.7. You have probably seen the control menu in other Windows work that you have done. If not, you can use the control menu to move, resize, and close the window with your keyboard. Using the mouse as just described, however, is easier than using the control menu. To close the control menu, you can click the control button once again or press the Esc key twice.

**Figure 2.7. The control menu.**

***Review:*** The five primary windows of Visual Basic supply the locations for controls and work areas that you will use to build Visual Basic applications. The Form window is the most important window for the applications that you write because it is on the Form window where you will draw and place interactive controls for the user to work with. The other windows exist to offer help and tools.

## The Top of the Screen

***Concept:*** As you learn more about Visual Basic's environment, you will find that Visual Basic conforms well to the standard look and feel of standard Windows programs. Many Windows programs contain

menus and toolbars that work much like Visual Basic's.

The top of the screen contains the menu bar and toolbar. The menu bar contains lists of pull-down menus with which you can manage your Visual Basic program. The toolbar supplies quick push-button commands for common tasks.

---

**Warning:** Don't confuse the terms *toolbar* with *toolbox*. The toolbar appears under the menu bar and contains buttons with icons on them. The toolbox is what is typically called the Toolbox window where the controls are located that you will eventually place on the form.

---

## The Menu Bar

If you have worked much with other Windows programs, you are already familiar with the File, Edit, View, Window, and Help menu bar commands because they are similar across many Windows applications. Table 2.2 describes all the Visual Basic menu bar commands with which you will work.

---

**Note:** The menu bar contains additional menus that pull down just like virtually all Windows applications use. These pull-down menus are sometimes called *submenus.* The commands on the submenus perform tasks or produce dialog boxes that require extra information from you before Visual Basic can issue the commands.

---

**Table 2.2. The menu bar commands.**

| *Command* | *Description* |
|---|---|
| File | The File menu contains all file-related commands with which you can load and save Visual Basic applications. It also provides printing access for printed program descriptions as well as the Exit command that you learned about earlier in this unit. |
| Edit | Programmers often use the commands on the Edit menu for copying, cutting, and pasting text and graphical controls among applications. The Edit commands also help you with the creation of your programs by supplying common search and replace actions. |

| | |
|---|---|
| View | The View menu command enables you to control the viewing of your application's Code window, various routines that can appear inside the Code window, as well as the toolbar. By hiding the toolbar, you can gain a little extra screen space. For instance, if you wanted more workspace and did not use the toolbar often, you could hide the toolbar by unselecting View Toolbar (the default is selected so that the toolbar appears). The toolbar will disappear. Selecting View Toolbar once again displays the toolbar. |
| Run | When you complete an application, you can see the results of your work with the Run menu. The Run menu enables you to execute programs, halt the execution, and resume the execution after a halt. |
| Debug | One of the most powerful features of Visual Basic is its debugging capability. With the Debug menu, you can execute a Visual Basic program one statement at a time, looking at data values along the way, and stop the program at any point to analyze what is going on. If a program does not behave the way you think it should, the Debug menu will help you pinpoint the cause of the trouble. |
| Options | You can determine the way in which Visual Basic behaves by modifying values within the Option menu. You can control both environment options the environment is the Visual Basic atmosphere in which you build programs and project options that determine how each particular application behaves. |
| Window | With the Window menu, you can display the Project, Properties, and Toolbox windows as well as auxiliary areas in Visual Basic such as Visual Basic's color selection box (with which you can assign colors to various controls you place on forms), the Menu Design dialog box (which you use for adding menus to your Visual Basic applications), and the Debug window (where you can work while debugging the program). |
| Help | When you select from the Help menu with the Primer edition of Visual Basic, you will not get online help as you would in the full version. If you select Help Contents, for example, the Visual Basic Primer displays the window shown in Figure 2.8, which describes the help system if you were to use the regularly-priced version of Visual Basic. (If you display this screen, select File Exit to get rid of it.) Some of the lower commands on the Help menu produce a message box that tells you that no help is available. You can, however, display the About box to see how much memory is available as well as the copyright notice and version of the Visual Basic Primer system. |

**Figure 2.8. Sorry. You will get no help from the Visual Basic Primer Edition.**

## The Shortcut Access Keys

---

*Definition:* An *access keystroke* is a shortcut method of issuing orders.

---

Many of the menu bar's commands also activate when you press an access keystroke. For example, if you display the File pull-down menu, you will see the menu shown in Figure 2.9. You can activate any command on the File menu by displaying the menu and selecting a commend. You can also issue orders

for four of the commands by pressing an access keystroke.

**Figure 2.9. Access keystrokes make selecting certain commands easier.**

Instead of selecting File Add File..., you can press Ctrl+D. Instead of selecting Save File, you can press Ctrl+S. The access keystrokes are available from within Visual Basic even if you do not first display the menu. For example, you can save the active file by pressing Ctrl+S without having to take the time to display the File menu first.

---

**Tip:** The Ctrl key works just as how the Shift and Alt keys work. Pressing Ctrl+S means press and hold the Ctrl key, and then press the S key while still holding down Ctrl, and then immediately let up on both.

---

Not all of the access keystrokes require that you use the Ctrl key. For example, some menu commands on the Edit pull-down menu do not require a second key such as Ctrl. Also, the Edit Find Previous command requires Shift+F3.

---

**Tip:** As you will see next, many of the toolbar commands provide the same functionality as many of the menu commands.

---

## The Toolbar: Push Button Swiftness

Figure 2.10 shows the toolbar and describes each button on the toolbar. Many of the toolbar buttons represent menu commands. Instead of issuing a menu command by using the mouse or an access key, you can point to a toolbar button to perform the same task.

**Figure 2.10. The toolbar contains quick access to many commands.**

As you progress through this book and learn how to use commands that appear on the toolbar, you will be reminded when you can use a toolbar button. Some people prefer not to use the toolbar. They either want more screen space or do not think the icons are that easy to remember they are *not* easy to remember. Remember that the View Toolbar command hides the toolbar from view if you do not want to see the toolbar.

---

**Warning:** Before you remove the toolbar, be sure that you do not need the measurement indicators that appear to the right of the toolbar. The next section explains what the measurement indicators do.

---

Notice that not all of the toolbar buttons are dark. Some are grayed out, just as some of the pull-down

menu bar commands are grayed out at times. Visual Basic knows that certain commands have to be activated at specific times within the program. If you have not copied text or a control to the clipboard, for example, you cannot use the Edit Paste command.

### The Measurement Indicators

As you draw and resize images on the Form window, you will often look to the two measurement indicators that appear to the right of the toolbar for help. The first indicator describes the upper-left corner measurement of a control, and the second indicator describes the size of the control.

---

*Definition:* A *twip* is 1/1440 of an inch.

---

Each of the measurements appear in twips. For example, Figure 2.11 shows a box placed in the center of the Form window. You know from the measurement indicators that the box's upper-left corner appears exactly 3,000 twips from the left edge of the Form window and exactly 1,560 twips from the top edge of the Form window. Likewise, you know that the box is exactly 1,215 twips wide and 495 twips long.

**Figure 2.11. The measurement indicators enable you to size and place controls on the Form window.**

By using the measurement indicators, you ensure that all screen elements in your user's application are aligned and properly adjusted for the size that you want.

---

**Note:** The Form window's grid of dots that you see in the background helps you align images with one other. The grid is sometimes called a *snap to* grid because controls that you place on the Form window snap to the nearest grid dot location if you place controls between two grid points. If you want to adjust the distance between grid dots, use the Options Environment command. If you want to turn off the grid so that you can place controls between grid points when you want, you can set the Align To Grid option to No from the same menu location.

---

***Review:*** You have now seen a complete description of the Visual Basic screen and its environment. Although you do not know how to use all the elements in the environment, you are at least familiar with the environment and will recognize the names of the screen elements when they appear later in this book.

## Homework

### General Knowledge

1. True or false: You can run but not compile Windows programs using the Visual Basic Primer disk.
2. True or false: You must install the Visual Basic Primer application onto your hard disk.
3. How can you set up a shortcut to run Visual Basic from the Program Manager without first opening the Visual Basic program group?
4. True or false: If your Program Manager contains a set of program groups different from those shown in this unit, you will not be able to install Visual Basic.
5. What command do you type to start the installation of this book's Visual Basic?
6. What does the term *default* mean?
7. What is the default name of the directory where the Visual Basic Primer installs itself?
8. How can you exit Visual Basic?
9. What happens if you turn off the computer before you exit Visual Basic?
10. Name Visual Basic's five primary windows.
11. Which window does Visual Basic use as the application's background?
12. True or false: There is no difference between the toolbox and the toolbar.
13. True or false: You can hide the toolbox from view.
14. What are the access keys used for?
15. What is the toolbar used for?
16. Why is Visual Basic's menu bar familiar to most Windows users?
17. How many access keystrokes are available on the Edit pull-down menu?
18. Why are some of Visual Basic's menu commands and toolbar buttons grayed out at times?
19. What do the toolbar's measurement indicators do?
20. What is a twip?
21. How does the grid help you align controls on the form?

## Extra Credit

Locate each toolbar button's corresponding menu command. In doing so, you will better familiarize yourself with the menu and its contents.

# Visual Basic in 12 Easy Lessons

---



---

# Project 1

## Welcome to Visual Basic!

---

**Note:** This first project is slightly different from the other projects because although you have been exploring the Visual Basic environment, you have not yet written a program. Beginning with the second project, you will see how a complete Visual Basic program works.

---

### Stop & Type

In this lesson, you learned about the history of Visual Basic and the fundamentals its programming environment. Visual Basic is a programming system rich in tools. The five windows of the Visual Basic programming system operate in conjunction with one other to give you the tools that you need to create programs.

In this lesson, you saw the following:

- An introduction to programming
- A brief history of the BASIC language
- How programming for Windows differs greatly from programming for text-based systems, such as DOS
- The kinds of programming errors that you can expect
- Why program errors appear and the steps that you take to correct them

- How to install the Visual Basic Primer disk that comes with this book
- How to start Visual Basic
- The parts of the Visual Basic screen and environment, such as the five fundamental windows of the program
- The Visual Basic menus
- How to stop Visual Basic

## Step 1: Start Visual Basic

Before you write a Visual Basic program, you must start the Visual Basic system. Follow these steps:

1. Turn on your computer.
2. Start Windows.
3. Press Ctrl+Alt+V if you followed the shortcut startup from Unit 2. Otherwise, open the Visual Basic 2.0 program group and double-click the Visual Basic Primer icon. The Visual Basic programming environment appears.

## Step 2: Practice Working with Visual Basic

1. Resize the Form window so that its lower edge reaches the bottom of your screen. Move the mouse cursor to the lower edge of the Form window until the mouse cursor becomes a two-headed arrow. Drag the lower edge downward as far as you can and release the mouse button.
2. When you clicked the Form window's edge, the Form window probably hid the Project and Properties windows from view. Press Alt+W, R to bring the Project window into view. By pressing Alt+W, R, you select the Window Project Window menu command.
3. The Toolbox window is a window that you will use nearly as often as the Form window. You might not like the location of the toolbox, however. Sometimes when you expand the Form window to the size of the entire screen, you will be placing items at the left of the Form window, where the toolbox is now. Therefore, practice moving the Toolbox window.
   Point the mouse cursor to the thin bar at the top of the Toolbox window. This bar to the right of the control button is known as the *title bar* even though there is no title on the Toolbox window's title bar. Click and hold the mouse, and drag the Toolbox window to the right side of the screen.

## Step 3: Modify the Environment

1. If you want to get rid of the toolbar the bar of icon buttons directly below the menu bar press

Alt+V, T. The toolbar disappears to give you more screen room.

2. Visual Basic enables you to turn off the grid dots on the Form window. Press Alt+O, E to select the Environment command from the Options menu. Scroll to the bottom of the list of options by pressing the Page Down key. If you are familiar with scroll bars, you can drag the scroll box down to its farthest point.
3. Click the Show Grid option, which is the second one from the bottom. Press N to change the option to No.
4. Press Enter or click OK to see the results. The Form window no longer has grid dots.
5. Feel optimistic with your Visual Basic skills? Reverse the changes that you made in this section. Turn the toolbar and the grid dots back on.

**Note:** If you want to play more with Visual Basic's environment, go ahead. When you are finished, exit Visual Basic by pressing Alt+F, X. Visual Basic will notice that you've made changes and will display a dialog box that asks whether you want to make changes to the form. Select No to indicate that you do not want this project's changes to remain in effect and that you want to return to Program Manager. Always return to Program Manager when you're finished with Visual Basic.

# Visual Basic in 12 Easy Lessons



# Lesson 2, Unit 3

## Contents of Visual Basic Programs

### What You'll Learn

- [lb] Loading and running a program
- [lb] The label control
- [lb] The text box control
- [lb] Command buttons are fun!
- [lb] Check box controls
- [lb] Option buttons limit the choices
- [lb] You've been framed

- [lb] Drop-down combo lists
- [lb] A simple combo box
- [lb] The list box for selections

This unit explains all about the format of Visual Basic programs. Unlike most programming languages, the Visual Basic language contains text commands as well as graphical controls. The text commands interact with the controls to produce the output of the program and the interactions with the user.

This unit concentrates on the graphical controls that you put in Visual Basic programs. As you read further into this book, you will gain a deep understanding of how the commands work. Some of the Visual Basic commands are easy, and some are complex. This unit concentrates on explaining the contents of programs. It shows you how to load and run Visual Basic programs.

**Note:** This is a hands-on unit. You will use Visual Basic as you progress through this unit to learn how each of Visual Basic's graphical tools work.

## Loading and Running a Program

***Concept:*** Once you start Visual Basic, you can create a program. Once you create a program, you can save that program and load and run it later. This unit gives you practice in loading and running a program that is stored on the disk that comes with this book.

---

*Definition:* Users perform I/O with *controls*.

---

This book's disk comes with a program that demonstrates several of the most important controls in Visual Basic. A control handles much of the user's input and output, often called *I/O*. In other words, when someone (the user) runs the program that you (the programmer) creates, the user must interact with the program by providing answers to questions and selecting options displayed on the screen.

Remember that the Toolbox window is where you get the controls that you place on the Form window. The Form window is the background of the application that you are creating. Figure 3.1 shows where the controls appear on the Toolbox window.

**Figure 3.1. The controls that your program needs are available on the Toolbox window.**

---

The Project File: A Visual Basic program rarely resides in a single file. Most of the time, it takes several files to describe a single Visual Basic application. It often helps to separate the pieces of a Windows program into multiple files.
For example, there is almost always a form file. The form file holds the contents of the Form window that you create as well as all the code that manipulates the objects on the Form window. You might also write additional code that you want to store in a separate file. If your the application requires special controls that do not regularly appear on the Visual Basic Toolbox window, you must add those control

files to the application, too.

---

---

The Project window contains a list of every file in the application. By default, the Visual Basic Primer disk always adds three files to every application's Project window: A form file with the default name FORM1.FRM and two special Visual Basic controls called GRID.VBX (more commonly known as the *grid control*) and OLECLIEN.VBX (called the *OLE control*). For now, don t worry about the special controls.
The Project window's contents are described in yet another file, which is called the *project file*. Every application has a project file, which always ends with a .MAK extension. When you want to load and run an application, you load its project file; Visual Basic ensures that all files in the project are loaded as well.

---

There is a program on this book's disk described in the project file called CONTROLS.MAK. Follow these steps to load CONTROLS.MAK:

1. Start Visual Basic if it is not running.
2. Select File Open Project from the menu bar. Visual Basic displays the standard Windows File Open dialog box.
3. Select the disk and path name of the Visual Basic Primer system if the disk and pathname are different from the location of your Visual Basic Primer system. Select or type the following filename at the File Name prompt: **CONTROLS.MAK**. You can type the name in either uppercase or lowercase letters.
4. Visual Basic loads the project. When Visual Basic finds the CONTROLS.MAK project file, it reads the file and loads all the files related to that particular project. You can see from the Project window that Visual Basic has loaded three files.
5. CONTROLS.MAK does not use two of its three files. The GRID.VBX and OLECLIEN.VBX files are extremely advanced; you can ignore them for now. Visual Basic does not usually display the form when you open a project file. To see the form, press the View Form button on the Project window. Your screen will look like the one in Figure 3.2.

**Figure 3.2. The Form window is often very crowded.**

The Form window works like an artist's easel of paper. Actually, the Form window works more like an artist's easel of one or more transparencies with something differently drawn on each transparency. CONTROLS.MAK contains several controls placed on top of one other. The program looks like a mess.

When the user runs the program, he sees the Form window. What the user sees, however, does not always look like the Form window when the program is first loaded. In other words, you are now looking at all the pieces of the program piled on top of each other. When the user *you*, in this case runs the program, the program ensures that those pieces appear in a logical order and do not overlap.

Before you or anyone else can run a program from within the Visual Basic environment, you must load the program through the File Open command, just as you have done. Once you load a program, you have three ways to run it:

- [lb] Select Run Start from the menu bar.
- [lb] Press F5, the shortcut access key for the Run Start menu command.
- [lb] Press the Start button on the toolbar. (As you saw in the previous unit, the Start button looks like the Play button on a cassette recorder.)

Use one of these three methods to run the program. You will see the more organized and less cluttered screen shown in Figure 3.3. The program's initializing instructions cleaned up the mess that you originally saw in the Form window. As you can see, the Form window of the Visual Basic environment becomes the application's background (minus the grid of dots in the background).

**Figure 3.3. The running program is much cleaner.**

This program demonstrates each of the primary Visual Basic controls so that you can familiarize yourself with using them before you place them in your own applications.

---

**Tip:** Notice that the program contains an Exit command button in the lower-right corner. All of this book's applications offer an Exit command button so that you can terminate the program whenever you want. When you create your own programs, always give your user a way to exit the program.

---

***Review:*** Before you can run a Visual Basic program, you must load it. Generally, you load the project file, which always ends with a .MAK extension. Loading the project causes Visual Basic to load all the files related to the application. Without a mechanism such as a project file, you would have to keep track of all the files related to every application and load each of them individually whenever you wanted to run a Visual Basic program. Once Visual Basic loads all the files related to the project, you can run the program and see what the program does.

## The Label Control

***Concept:*** The label control is one of the simplest controls to work with. With the label control, you can add descriptive text to the form in any location by using different styles and sizes of fonts.

The label control, appearing on the Toolbox window with a capital letter *A*, holds text. The user sees the text on the resulting application. The title of the CONTROLS.MAK application, Have fun with controls! is placed on the form with the label control.

As you will see as you progress through this book, you can set several property values as you place controls. You manage the size of the control and the way the control looks. You can specify a large font size or a small font size, a font style based on any available font style in your Windows system, whether you want the font boldfaced, italicized, or struck through with the strikethrough font style, where the text has a straight line drawn through it.

You can draw a box around the label and shade its background any color that you want. Look again at the running program. You can see a command button labeled Next Control. Click this command button

with the mouse or press Alt+C. Two new labels will appear in the middle of the screen, as shown in Figure 3.4.

**Figure 3.4. Two additional label controls appear in the center of the form.**

---

**Tip:** If a command button contains an underlined character for example, the Next Control command button at the right edge of the program window in Figure 3.4 the underlined character represents the shortcut access keystroke. You can combine the Alt key with the shortcut access key to trigger a press of the command button.

---

---

**Warning:** The user cannot change the text on a label directly. Through code, you can change the label's text in response to a user's action when the need arises. Generally, however, you will set the text within a label when you design the program.

---

***Review:*** The label control is one of the easiest controls to add to your applications. Whenever you need to display text within a title or a description for another control, the label control enables you to display the text with various font sizes and styles.

## The Text Box Control

***Concept:*** Unlike the label control, the user can change values within a text box control. You can get answers from the user by using text box controls.

When you click the Next Control command button on the running program, the labels that you displayed earlier disappear and a text box appears in their place, as shown in Figure 3.5.

**Figure 3.5. The user can change the text within a text box.**

---

**Tip:** If you display a text box that has no initial text within it, the user can type text in the text box control in response to a question that you ask. Sometimes, however, you will display initial text within the text box that the user can use as a default value, changing the text only if the default is not the needed value.

---

Click the mouse cursor anywhere within the Change this! text box. Type new text in the box. Type lots of text in the text box. The text scrolls to the right to accommodate the new text. You can use the arrow keys to move the cursor back and forth within the text box. Also, the Ins and Del keys work just as they do in a word processor. With these two keys, you can insert and delete text from the text box.

The text box control the control with the lowercase ab enclosed in a box enables you to set an initial

value and to control how the text's font style and size appear when the user sees or enters text in the text box. You can control whether the text box contains horizontal and vertical scroll bars so that the user can scroll through the text with scroll bars or with the regular arrow keys.

***Review:*** When the user must enter new text or change existing text, use the text box control to place an area on the screen for the user's keystrokes. The text box control works just like a mini word processor.

## Command Buttons are Fun!

***Concept:*** You have seen command buttons in almost every Windows program, including Visual Basic and the currently-running CONTROLS.MAK application. Command buttons give users push-button access to events that you place within an application.

A command button appears on the screen just as push buttons appear on a VCR or on your keyboard. The CONTROLS.MAK program contains two command buttons that stay on the screen at all times: The Next Control command button and the Exit command button. As you already know, you can trigger a command button's press with a mouse click or with a shortcut access keystroke.

Press the Next Control command button now to see a third command button replace the text box control in the middle of the screen of the running CONTROLS.MAK application. Figure 3.6 shows the screen after the command button is displayed.

**Figure 3.6. A command button controlled by the program.**

Virtually anything can happen when the user clicks a command button. You, the programmer, control exactly what happens. Go ahead and press the CONTROLS.MAK command button. Look and listen to what happens. The computer beeps, and the command button s caption changes from Press Me to Once Again. Press the command button again to hear the beep once more and to restore the command button to its original Press Me state.

You set the command button's caption usually when that you design and write a program. As you can see, you can write the program so that the command button's caption changes when the user clicks the button.

***Review:*** A Windows application without command buttons is like day without night. (Okay, that's exaggerating.) Command buttons supply push button access to events that you want the user to trigger.

## Check Box Controls

***Concept:*** Check box controls offer multiple-choice values from which the user can select. Once the user selects one or more check boxes, your program can analyze the selected check boxes and make decisions based on those responses.

Click the Next Control command button to see the list of three check box values shown in Figure 3.7. The check boxes offer the user a way to select one or more values from a list of values that you display.

**Figure 3.7. Two of the three check boxes are selected.**

With the mouse, click the first and last check boxes on the running CONTROLS.MAK application, as shown in Figure 3.7. When you select a check box, the boxes to the left of the descriptions fills with an X.

Once you click these two controls, the two values are said to be *selected,* and the middle value is *unselected* or *not selected*. You can *deselect* a check box by clicking it a second time. Click the Pears check box to deselect that check box. The X leaves the check box when you deselect the box.

---

**Tip:** You can click not only on the box but anywhere within a text box's description. Therefore, to select or deselect the Pears check box, you can click the mouse cursor over the box next to Pears or click on the word Pears.

---

There is also a way to select check boxes without the mouse. The user can press the Tab key until the check box description highlights. To select it, the user presses the Spacebar.

***Review:*** Check boxes give your users multiple-choice access to choices that they need to make. Users select and deselect check boxes with the mouse or keyboard. Once the user selects all the check boxes needed, a command button keypress can signal to the running application that the user is finished. Your program then can check which values the user selected.

## Option Buttons Limit the Choices

***Concept:*** Unlike check boxes, option buttons give your users a list from which to choose, but they can select exactly one option out of the list.

Often, the option button controls are known as *mutually-exclusive* controls. Unless you group sets of option buttons in *frames* (described later in this unit), the user can select one and only one option button at a time.

Click the Next Control command button to see the CONTROLS.MAK option buttons on your screen like the ones shown in Figure 3.8. Initially, no options are selected.

**Figure 3.8. Option buttons work almost like mutually-exclusive check boxes.**

With the mouse, click one of the option buttons. Click another option button. At once, Visual Basic deselects the first option button and selects the one you just clicked. Select another option button to change the selected option once again. As you can see, Visual Basic ensures that you can select only one option at a time. Instead of check boxes, which permit multiple selections, you would display option buttons for the user whenever he can make only one choice out of several.

---

**Tip:** Think ahead when you use option buttons. For example, you can select an initial option button for

the user, through code or through the Properties window, when you write the program. By selecting an initial option button, you ensure that the user knows the best choice in a given situation, assuming there is a good default value that you can select.

---

***Review:*** The option buttons work like check box controls except that, whereas the check box controls permit multiple selections, the user can select at most one option button at a time.

**You've Been Framed**

***Concept:*** The frame control enables you to group items together on a form. The group works almost like a miniform within the form.

Although the user can select only one option button at a time, you can set up groups of option buttons on the same form. The user can then select one option button at a time from *within each frame*.

---

*Definition:* A *frame* is a box in which you can place control groups.

---

You must always enclose the group within a frame by using the frame control. If you press the Next Control button again, you will see a frame appear in the middle of the form, as shown in Figure 3.9. You can place any controls in a frame, not just option buttons as in the figure.

**Figure 3.9. A frame with a command button and two option button controls.**

If three frames of option button groups appeared on the form, you could select a maximum of three option buttons on the form one in each framed set.

---

**Note:** To keep things simple at this point, only one control is discussed at a time. Only one set of option buttons currently appears on your form. There is no special reason to group them together except to show what a frame is.

---

***Review:*** By framing objects together with the frame control, you can set up groups of controls that work together as if they were each on their own miniform within the larger Form window.

## Dropdown Combo Lists

***Review:*** A dropdown combo list is one of three kinds of lists that you can provide for your user. The dropdown list saves room on the screen by consuming only a single line on the form until the user opens the list to display the rest of the items in it.

The combo box control actually turns into two different kinds of controls on the form, depending on how you set up the combo box. The two kinds of combo boxes are

- [lb] Dropdown combo boxes
- [lb] Simple combo boxes

Press Next Control to see the combo dropdown box on the screen. As so often is the case, a command button appears next to the combo dropdown box. (See how you can resize every control, including command buttons, so that the controls consume exactly as much screen space as needed?)

A list of items is stored in this dropdown combo box. To see the list, click the down arrow at the right of the empty dropdown combo box. You will see the list of electronic gear shown in Figure 3.10. Click the down arrow again. The list folds back up leaving the screen clear of the extra clutter.

**Figure 3.10. After opening the dropdown combo box control.**

---

**Tip:** When screen real estate space is precious, use a dropdown combo box control when you must offer a list of items to the user. The user can display the list when he needs to see what is in the list, and he can restore the list to its normal state when he is finished.

---

The blank space at the top of the list is for the user to add additional items to the dropdown combo box. You will want to add a command button next to the combo box, as in CONTROLS.MAK, so that the user can add new items after typing them. Add two additional items by following these steps:

1. Type **Disc Player**.
2. Click the Add button. The data-entry box where you typed the new value goes blank, but Visual Basic has added Disc Player to the list.
3. Type **Amplifier** and click the Add button.
4. Display the dropdown list by clicking the down arrow. Notice that Visual Basic has added the two new items to the bottom of the list.

---

**Note:** Later, you will learn how Visual Basic can keep the items sorted alphabetically you do not have to write any code to sort the items.

---

The items remain in the dropdown combo box list until you quit the program. Throughout this book, you will learn ways to save the items in the list and to initialize the lists with values using code.

***Review:*** The dropdown combo box control offers a handy way for the user to view and add items to a list. The dropdown combo box gives you the advantage of listing items for the user without taking up screen space that is needed for other things. Your user can display the entire list by clicking the down arrow.

# Simple Combo Box

***Concept:*** The simple combo box control does exactly the same thing as the dropdown combo box except that the simple combo box is always displayed in its *dropdown* form. In other words, if screen space is not a problem, you might want to use a simple combo box to display and collect values by using a list that does not require the user's extra keypress to open the list.

Press the Next Control command button to see what a simple combo box looks like. As you can see, there is no difference between a simple combo box and a dropdown combo box except that the simple combo box is always open.

If the simple combo box is not deep enough to display all the items in the list, Visual Basic adds a vertical scroll bar to the list so that the user can scroll through the list and view the items. As with the dropdown combo box, the user can add items to the simple combo box by typing the new value and clicking a command button that you set up.

Type the value **Rugby** and click the Add command button. Scroll to the bottom of the list you can use the Page Down key to scroll to see the new sport's name.

---

**Note:** There is a third kind of combo box, called a *dropdown list box* control, that I will not cover here. The dropdown list box offers nothing new that you cannot get by using the other combo boxes or list box controls. Discussing it would only add confusion at this point.

---

***Review:*** Simple combo box controls provide an easier way to display and collect list values. The user does not have to understand how to open a simple box control it is always open.

# The List Box for Selections

***Concept:*** List boxes are easy to understand compared to the two combo box controls. If you want users to select from a choice of options that you have supplied and you want to prevent them from adding additional items to the list, use list boxes.

Figure 3.11 shows how your screen looks after you click the Next Control command button again. Later you will learn how to capture the item or items that the user selects from a list box.

**Figure 3.11. Selecting from a list box.**

---

**Warning:** Check boxes and option buttons give users a chance to select from options, but you should not use check boxes or option buttons when there are many options to choose from. The scrolling list box and combo box controls use screen space more efficiently.

---

Click over one of the list box items with the mouse. Visual Basic highlights the item. Through programming, you will know when users selects an item, and you can analyze what they select. Click another item to select another choice.

If the application requires multiple selections, you can set up the list box for multiple selections. The list box control inside the CONTROL.MAK application permits multiple selections. Hold down the Ctrl, Alt, or Shift key and click one or two additional items. Notice that Visual Basic highlights all the items. You can control whether users can select only one or more than one item from a list box. As with combo box lists, you can request that Visual Basic sort the items in the list box alphabetically.

If you want, you can step through all the controls again by clicking the Next Control command button. You have now seen all the controls offered by the CONTROLS.MAK application. Although you will learn about more controls later in this book, this unit has described the primary controls that all Visual Basic programmers should master. Be sure that you know all the names of the controls. Study Figure 3.1 so that you know where each control is located on the Toolbox window.

---

**Warning:** Not every Windows program that you write will contain all these controls. As a matter of fact, *no* Windows program should contain all these controls. The program would be too busy and would require too many different kinds of responses from the user.

---

***Review:*** List boxes are controls that give the users choices from which they can select. Your application is responsible for initializing list boxes with values. The user cannot add items to a list box.

## Homework

### General Knowledge

1. Before you can run a program from within the Visual Basic environment, what must you do?
2. What kind of file holds the description for the entire application?
3. What is the filename extension for project files?
4. Where can you look to learn the contents of a project file?
5. What are controls for?
6. What window do you get controls from as you build applications?
7. Why is it a good reason to add an Exit command button to applications?
8. True or false: The user can change text in a label control.
9. Which control is good to use when you need text from the user?
10. What kinds of things can you do with text displayed in a label or text box control?
11. How do text controls act like mini word processors?
12. What is the control called that gives users access to push buttons?

13. True or false: The user can select at most one check box from a list of check box controls.
14. True or false: The user can select at most one option button from a list of option button controls.
15. What happens if the user selects a check box a second time?
16. What control enables you to add several groups of option buttons to a form?
17. True or false: Users can add items to lists displayed in dropdown combo box controls.
18. True or false: Users can add items to lists displayed in simple combo box controls.
19. True or false: Users can add items to lists displayed in list boxes.
20. True or false: Users can select more than one item in a list box.
21. What serves as the user's application background, holding all the controls for the user to work with?
22. True or false: Visual Basic gives you, the programmer, a chance to add list box and combo box controls that display lists alphabetically.
23. Name at least two ways in which the user can trigger a command button press.
24. Why is a command button often next to a combo box control?

## Find the Bug

1. Pamela is writing a Visual Basic application that displays customer order information. Because of the large amount of information for each customer, Pamela finds that the screen is getting far too crowded. Pamela adds the list of goods that the customer bought to a simple combo box control. That way, the clerk can scroll through the list of items, and purchases are added to the list they are rung up. The simple combo box control takes up too much screen space. What recommendation would you give Pamela?

## Extra Credit

Suppose that you are writing an application that offers users a list of three values from which they can select at most one of the values. Which control is the most appropriate?

# Visual Basic in 12 Easy Lessons



# Lesson 2, Unit 4

## Bare-Bones Programs

### What You'll Learn

- [lb] Event-driven environments
- [lb] Control properties
- [lb] Naming conventions
- [lb] AUTOLOAD.MAK and CONSTANT.TXT

In this unit, you will create your own program with Visual Basic from scratch You will write a fully-working Windows program that displays a resizable window, a command button, and a control button with a control button menu.

Before you create your first application, however, you must understand how Visual Basic programs interact with the Windows environment.

# Event-Driven Environments

***Concept:*** A Windows program behaves differently from a DOS-based program. Instead of the program controlling the user, the user controls the program. When the user responds to a menu or control, Windows generates an event that describes the particular action.

Picture yourself driving down the road. All kinds of controls are at your fingertips steering wheel, blinkers, headlights, breaks, gas pedal, gear shift, radio knobs, rearview mirror, and air conditioning and heater controls. At any time, you might press the brake, turn left, turn on the radio, adjust the mirror, or speed up. The driving conditions, not the physical order of the controls, determine what you do next.

When you press the gas pedal, does your foot alone make the car go faster? The answer is no. You would have to have a very powerful foot to be able to speed up a car already going 50 miles per hour. The foot pedal causes an event to happen. That event is that more gas is fed to the car for fuel to burn and the car goes faster. Your car performs its job and makes adjustments based on the events that you trigger.

---

*Definition:* An *event* can be a mouse move, a mouse click, a keystroke, or a control response.

---

Using a Windows program is like driving a car in the following respect: You do not always perform the same actions, and the actions that you perform cause certain events to occur. In Windows terminology, an *event* is the action that the user takes. Whenever the user clicks the mouse, presses a key, responds to a control, or selects from a menu, an event happens. Windows constantly monitors the running Visual Basic program, looking for events.

---

**Note:** Too many things can happen in a GUI-based program for it to follow a straight, sequential pattern. If you were to run a DOS-based accounting program, the program more than likely would present you with a menu of limited choices. Only after you select from the menu, does the program take you through the next step. In a Windows program, you are offered a selection of controls, and you can respond to any control in any order. The program must be able to sense when an event takes place and to handle it accordingly.

---

Windows and Visual Basic constantly monitor running programs. When the user clicks a command button or performs any other kind of event, Windows intercepts the event and sends it to Visual Basic.

Visual Basic does not respond to some special system events, such as when the user presses Alt+Tab to switch to another application running in memory. That is why Windows must interpret all events and pass the ones handled by Visual Basic to Visual Basic. Figure 4.1 shows the relationship between events and event procedures. Notice that Windows always intercepts the events and passes the appropriate events to the Visual Basic program, where Visual Basic then does something in response to them if event procedures are available.

**Figure 4.1. Windows intercepts events before it passes them to Visual Basic.**

---

*Definition:* An *event procedure* responds to an event.

---

When Visual Basic gets an event from Windows, it checks whether the programmer wrote an *event procedure* for the event. If an event procedure exists, Visual Basic executes the event. In the CONTROLS.MAK program that you ran in the previous unit, clicking the Next Control command button caused the next control in the program's repertoire of controls to appear. The only way that the program could respond to the Next Control command button was for an event procedure to be written for that particular event namely, the command button keypress.

A program can contain controls and still not respond to all events. In CONTROLS.MAK, for example, when you select the check boxes, nothing happens except that the boxes are selected or deselected. The check box controls handle the checking and unchecking of the boxes so that no event procedure is needed to do that. Once you select check boxes, though, the program does absolutely nothing with the selected check boxes because there is no event procedure inside CONTROLS.MAK that does any work when the user selects a check box. The check boxes exist in that program only to illustrate how they operate.

---

**Warning:** Most, but not all events, are user-triggered. Some internal Windows events can take place that trigger actions. Likewise, you can direct a program to respond to timed intervals, such as ticking a clock forward every second. The interval of time would be an event in that case.

---

Is all this talk about events and event procedures getting technical? Actually, you will see that the implementation of event capturing and event procedures is extremely easy. The Visual Basic environment is set up to create event procedures for you when you request them. The bottom line is this: When you want your Windows program to respond to an event, make sure that you write an event procedure for it. If you want to ignore certain events, don't write event procedures for them.

---

**Tip:** Most of the events that you want to handle are obvious. For example, if you add a command button control to a form, you want to do something when the user clicks the command button. However, if the user tries to drag the command button with the mouse, you probably want to ignore that event because the user should rarely be allowed to move controls during the execution of the program.

---

***Review:*** Almost anything that can happen during the execution of a Windows program can be an event. Once you design a Windows program form and place controls on it, you must write code that responds to events. That code is made up of event procedures. In a way, each event procedure is like a miniature program that you write for each control whose event should cause an action to take place.

# Control Properties

***Concept:*** All controls are different and work differently. You use different controls for different things. Even among controls of the same type, however, differences exist. Those differences reside in the control properties.

---

*Definition:* A *property* determines how a control differs from other controls.

---

Consider the three controls shown in Figure 4.2. All three controls are command buttons even though they all look different. The user can click any of them, but each has a different set of properties.

**Figure 4.2. Three command buttons with different properties.**

The top command button has the size seen most often in Windows programs. Visual Basic automatically assigns this size when you add command buttons to your applications unless you specify something different. The second command button is much wider than the top one, and no access keystroke is available. Its caption is not displayed in boldfaced letters, and the font is not standard. The third command button is very small, and its caption is italicized. Because small italics on command buttons are not always easy to read, be careful about using them with small controls.

---

**Tip:** Less is usually better.

---

Adding too much of anything is usually worse than better. The application often requires that command buttons differ from their default size, but try to stay as consistent as possible. It is rarely a good idea to put more than one font on more than one command button that appears on the same form.

One of the most important steps you take when you write Visual Basic programs is setting control properties. When this book teaches you how to add a new control to an application, you will learn about virtually all of its properties. The next unit, for instance, shows you how to place command buttons. Before you learn about placement, though, you will read about the command button's 25 properties.

---

**Note:** Rarely will you have to change all the properties of a control when you place the control on the form. Nevertheless, if you see every property that is available when you learn a new control, you will know what you can and cannot do it.

---

---

Even Forms Have Properties: Visual Basic programmers use the generic term *object* for controls that they place on the form. Actually, even the form is an occurrence of an object. The word *object* means

different things in different computer languages. Although faintly related, a Visual Basic object has little to do with the objects that you find in object-oriented programming languages such as C++.
Every object in a Visual Basic program has properties. Even the forms have properties. As you can see from Figure 4.2, you can add descriptive titles to forms. The title of a form is one of its property settings. The background color is another property that you can set. Although you should stick to forms with white backgrounds for the sake of consistency it is used for most Windows programs you, the programmer, can change it.

---

---

**Tip:** Where do you add properties for controls?
In the Properties window. (Good name, huh?)

---

***Review:*** Each control has a different set of property values. You often set initial property values when you place controls on the form. During the execution of the program, your code also often changes property values. The CONTROLS.MAK application, for example, changes a command button's caption property from Press Me to Once Again and back to Press Me. I added the original Press Me caption when I added the command button to the form. Then I used code to change the caption during the program's execution. What enables Visual Basic to know when to change the caption? When the user presses the command button, a command button click event occurs. Then the event procedure written for that particular event changes the caption. Read on, true Visual Basic believer, and you will see that event procedures are neither as difficult, nor as complicated, as they might first sound.

## Naming Conventions

***Concept:*** Although you will learn all about control properties throughout this book, one control property is worth learning about early on the Name property. All controls have a Name property. The Name property labels each particular control. Without a unique name, you could not distinguish one control from another inside the Visual Basic code. Although Visual Basic assigns default names to all controls, get in the habit of changing those names to something more descriptive so that you can remember them more easily as you add to the program.

---

*Definition:* A *naming convention* is a set of naming rules.

---

The first property that you should always set is the Name property. This unit does not tell you *how* to set the Name property; that is covered in the next lesson. You must know in advance, however, that Visual Basic programmers do not arbitrarily assign names to controls. Programmers who want to make their programming lives less stressful follow prescribed naming conventions when they choose names for their controls.

A convention is not just a group of people gathered for the weekend. A convention is a standard set of

rules that you follow. You are already familiar with naming conventions for example, the file-naming conventions used in Windows and DOS-based computers (up to eight characters for the name and up to three characters for the extension). In Visual Basic, the naming convention for controls is simple. When you decide on a name for your control, be sure to stay within these boundaries:

- [lb] Names can be as short as one character or as long as 40 characters.
- [lb] Names must begin with a letter of the alphabet and can be in either uppercase or lowercase letters.
- [lb] After the initial letter, names can contain letters, numbers, or underscores in names.

---

*Definition:* A *reserved word* is a Visual Basic command.

---

1. [lb] Names cannot be the same as a reserved word. Appendix B lists all the reserved words in Visual Basic.
- [lb] Names should make sense. For instance, although you could name an exiting command button Rose, cmdExit is better; it is self-documenting and easier to remember.

The following are valid names:

cmdExit

ListBoxJan

Nov95Combo

TitleScreen

and these are not:

Select

723

cmdExit&Leave

Select is a Visual Basic command. 723 does not begin with a letter of the alphabet. cmdExit&Leave contains an invalid character, the ampersand (&).

---

**Warning:** Even though the underscore is a valid character, many programmers prefer not to use the underscore because it sometimes can look like a minus sign.
Many programmers use *hump* notation mixing uppercase and lowercase letters in names that contain several words. The term *hump* comes from a camel's back, which the uppercase letters in the name sort of resemble. For example, cmd_Add_Sales is a valid name, but cmdAddSales is just as readable and it does not contain the underscore characters that sometimes lead to confusion.

---

Learn these naming conventions well. The naming conventions apply not only to Name properties but

also to other aspects of Visual Basic, such as *variables* and *procedures*.

---

Write Once, Maintain Often: Rarely are you finished with a program after you write it. You usually have to update the program to reflect changes in the environment in which you use it. When you update a program already written, you are doing what is called *program maintenance*.
For example, if you wrote an accounting program for a small company that merged with a second firm, the accounting department might have to keep both companies separate until the current fiscal year ends. You must modify the program so that the program distinguishes between the two companies and keeps two sets of data. The more time you spend giving meaningful names to controls, the less time you will spend later trying to figure out what each control is for. It is obvious that a control named cmdComputeProfit triggers the computation of a profit calculation.

---

I strongly suggest that you adopt is the standards for naming control prefixes listed in Table 4.1. Table 4.1 shows the three-letter prefix that you should add to the front of a name when you name a control. The prefix describes what the control is. Therefore, from the name itself, you know what kind of control you are working with. Of course, if you placed the control on the form and named the control to begin with, you would know what kind of control you are adding. When you add code to event procedures later, however, the three-letter prefix helps you keep the kinds of controls straight and it prevents you from trying to write an incorrect event procedure for a control that does not produce that particular event.

---

**Note:** Table 4.1 lists all the naming conventions for controls and forms, including those controls that you do not know yet.

---

**Table 4.1. Standards for naming control prefixes.**

| *Prefix* | *Control* |
|---|---|
| cbo | Combo box |
| chk | Check box |
| cmd | Command button |
| dir | Directory list box |
| drv | Drive list box |
| fil | File list box |
| fra | Frame |
| frm | Form |
| grd | Grid |
| hsb | Horizontal scroll bar |
| img | Image |

| | |
|---|---|
| lbl | Label |
| lin | Line |
| lst | List box |
| mnu | Menu |
| ole | OLE client |
| opt | Option button |
| pic | Picture box |
| shp | Shape |
| tmr | Timer |
| txt | Text box |
| vsb | Vertical scroll bar |

Here are some control names that use the three-letter prefixes from Table 4.1:

frmOpening

lstSelections

chkBooksInPrint

There is a wider blanket of conventions that cover the look and behavior of all Windows programs. The book *The Windows Interface: An Application Design Guide*, by Microsoft, discusses all the standards. It suggests how you can design your Windows programs so that they behave like other Windows programs.

When users move from DOS to Windows, they often complain that Windows is different and hard to learn and use. They are correct that Windows is different, but there is there is much disagreement on how difficult and hard-to-use Windows is.

Even if you accept that Windows is hard to learn and use, you have to learn the Windows interface only once. That means that after you master a Windows program, such as the Visual Basic programming environment or Microsoft Excel, every other Windows program behaves in almost exactly the same manner. Almost every Windows program contains an Exit command on a File pull-down menu. Almost every Windows program displays the same File Open dialog box. Almost every Windows program displays a white background. Software developers do not have to follow these standards, but users are more likely to learn the programs if they do. The application design guide helps them stay consistent with the standards.

***Review:*** When you name controls, don't assign arbitrary names or even stick to the name that Visual Basic assigns you won't like Visual Basic's suggestions. Select a name that indicates the purpose of the control, and use one of the three-letter prefixes described in Table 4.1. By following such standards, you will ensure that your program is easier to maintain down the road.

# More Consistency: AUTOLOAD.MAK and CONSTANT.TXT

***Concept:*** With the AUTOLOAD.MAK and CONSTANT.TXT files, you can add even more consistency to your Visual Basic programs.

Over time, you incorporate many common elements in your Visual Basic programs. Perhaps you have written a set of routines in the Visual Basic language that you want to make available to several Visual Basic programs. As you already know, by adding files to the project window, you, in effect, add those files to the resulting application.

AUTOLOAD.MAK is a special project file that you can load and look at. If you load the file, you won t see anything that looks very special. The Form window is empty, and the Project window contains the same two files, GRID.VBX and OLECLIEN.VBX, that you saw earlier when you started Visual Basic and loaded and ran an application.

The purpose of AUTOLOAD.MAK is to create a base application that you add new features to and build on to create another application. In other words, whenever you start creating a new application using the File New Project command, Visual Basic looks at the contents of AUTOLOAD.MAK and creates a new base application that looks just like AUTOLOAD.MAK. Therefore, if you want to change the default behavior of your new applications, load and change AUTOLOAD.MAK. After you change AUTOLOAD.MAK, all future projects that you create will hold those changes.

---

**Tip:** Think of AUTOLOAD.MAK as working as how the DOS AUTOEXEC.BAT file works. Whenever you start your computer, DOS looks at the AUTOEXEC.BAT file and starts the computer according to those instructions. Every time you start a new application, Visual Basic looks at AUTOLOAD.MAK and creates a new application with the same features as AUTOLOAD.MAK.

---

Now take a look at how changing AUTOLOAD.MAK changes all your new projects. Load AUTOLOAD.MAK if it is not loaded, and look at the Project window. Because you won t need the GRID.VBX or OLECLIEN.VBX files for a while, there is no reason to load them along with all the others every time you create a new application. Once you load AUTOLOAD.MAK, follow these steps to remove the files:

1. Display the Project window if you cannot see the Project window.
2. Highlight the GRID.VBX file.
3. Select the File Remove File command. As soon as you do, Visual Basic removes GRID.VBX from the Project window. The last file in the project, OLECLIEN.VBX, is still highlighted.
4. Select File Remove File again. Visual Basic removes OLECLIENsx.VBX from the project window.

---

**Note:** By the way, your toolbox now contains two fewer controls because the two .VBX files that you

just removed contained special controlling tools called *custom controls*.

---

If you now saved AUTOLOAD.MAK to the disk, all future projects that you create would look just like the AUTOLOAD.MAK that you now see. Before you save AUTOLOAD.MAK, however, read a little further to learn about another file that you should add to the Project window.

---

**Tip:** Add the CONSTANT.TXT file to lighten the rest of your programming burden.

---

---

*Definition:* A *named constant* is a constant value with a name that is easy to remember.

---

There will be several times when you have to set various controls to specific values from a list of possibilities. For example, a label control can have a boxed border around it if you set the value of a certain property to 1 a *fixed single-line border* and no border if you set it to 0. Instead of setting properties with those hard-coded values, you can use descriptive named constants such as NONE and FIXED_SINGLE.

The designers of Visual Basic took every possible control property value and assigned names to them so that you can use either the names, which are usually easier to remember than actual numbers, or the values themselves. Once you get used to using named constants, you will rarely use the actual values.

---

**Warning:** Don't fret. The use of named constants will make a lot more sense when you begin building your own applications.

---

All the named constants are stored in a file named CONSTANT.TXT. You can add CONSTANT.TXT to your AUTOLOAD.MAK's Project file so that all the applications you eventually create automatically contain that file as well. Follow these steps to add CONSTANT.TXT to the Project window of AUTOLOAD.MAK:

1. Select File Add File from the menu bar. Visual Basic displays a File Open dialog box. Visual Basic assumes that you want to add either a form (.FRM) file, a Visual Basic language (.BAS) file, or a custom control description (.VBK) file. Therefore, Visual Basic displays only those kinds of files. Instead of adding one of those files, you want to add a text file, which typically has a .TXT filename extension, as in CONSTANT.TXT. Therefore, you must override the suggested filenames in the File Name prompt.
2. Type CONSTANT.TXT for the filename, and press Enter or click the OK command button. Immediately, Visual Basic adds CONSTANT.TXT to the project window, as shown in Figure 4.3.
3. Save the AUTOLOAD.MAK file so that the changes you have made using File Save Project are reflected in all future projects that you create.

**Figure 4.3. The CONSTANT.TXT file is part of the Project window.**

***Review:*** The AUTOLOAD.MAK file determines what all subsequent projects that you create will initially look like. If you remove the custom control files that you find in AUTOLOAD.MAK's Project window and add the CONSTANT.TXT file, all new projects that you create will initially contain CONSTANT.TXT.

## Quick to the Draw!

***Concept:*** Finally! You will now use Visual Basic to create your very own fully working Windows program. If you don't think that you know enough about Visual Basic to write programs, hang on because you will see how easy creating a program can be. Follow these steps to create your first Visual Basic application:

1. Select File New Project to open a new application with a Project window that looks like AUTOLOAD.MAK's that you just saved. Visual Basic opens a new project and a blank Form window on which you will place controls.
2. Double-click the label control. Remember that the label control is the uppercase *A* on the Toolbox window. A blank label control (with the *terrible* default Name property of Label1) appears in the center of the Form window.
3. Press F4 to bring the Properties window into view.
4. Scroll the Properties window to the Caption property. The Caption property, by default, contains the name of the label. Because the Caption property holds the label's text, you should change the text. If you highlighted the Caption property, type **My First!** and press Enter. As Figure 4.4 shows, the label immediately displays the new caption.

**Figure 4.4. After changing the label's caption.**

5. Click and hold the mouse cursor over the label that you just added, and drag the label towards the top of the Form window. Leave about an inch between the top edge of the label and the top edge of the window. Center the label under the title of the form. By default, the title is the name of the form, Form1. (You will change control names from their default names in the next unit.)
6. Double-click the command button control on the Toolbox window. A command button appears in the center of the Form window.
7. Type the following exactly as you see it: **E&xit**. The ampersand (&) causes the x to be underlined, as you can see when you look at the resulting command button's caption. The Caption property is the first property that changes because the *last* property that you changed when you worked with the label control was its Caption property.
8. Center the command button by dragging the command button a little to the left. Once you center the command button, double-click it. Immediately, Visual Basic opens a Code window as shown in Figure 4.5.

**Figure 4.5. The Code window opens when you double-click a control.**

9. Visual Basic knows that you want to write an event procedure that executes whenever the user clicks the command button. You can tell that Visual Basic opened an event procedure for a command button keypress because the name of the procedure is Command1_Click (). (Don't worry

about the parentheses right now.) An event procedure always takes the following format:
ControlName_EventName ()
The default name for a new command button (until you change the Name property) is Command1, and the event that triggers when the user clicks a command button is Click. Therefore, the code that handles this command button's click is Command1_Click ().

10. The two lines that Visual Basic adds to the event procedure are called *wrapper lines* or *wrapper code* because they wrap around the code you add to the event procedure. For this event, simply press Tab and add **End**. The full event procedure should look like this:
Sub Command1_Click ()
End
End Sub
11. That's it. To see your handiwork, close the Code window and press F5 to run the application you just created. The application's form appears on the screen with its two controls, as shown in Figure 4.6.
To terminate your application and return to Visual Basic, click the Exit command button. As soon as you click the command button, the event procedure that you added executes. The End statement that you added to the event procedure also executes. The sole purpose of End is to terminate the application.

**Figure 4.6. Your first application works like a charm!**

---

Other command button events: Other events are possible with command buttons. Open the Code window and look at the other event names. Click the down arrow at the right of the Code window's Proc: dropdown list. Scroll through the list. You will see that there is a DragDrop event for when the user drags the command button with the mouse and a KeyDown button for when the user presses the command button's shortcut access key, among others.
Other kinds of controls, such as the list box or label control, might have events that are identical to those of the command button control, but the other controls also have different events. The Code window's Object: dropdown list always contains a list of the application's current objects such as the form and any controls you have added and the Proc: dropdown combo list contains a list of events for each of the controls that you highlight in the Object: list.

---

If you want, you can save this application. However, the disk that comes with this book contains the full application; it is called MYFIRST.MAK. Therefore, if you save your work, save it under a different filename.

When you save your applications, Visual Basic wants you to name both the form and the entire project. Usually, especially for the one-form applications that you will write in this book, you name the form the same name as the project, but both have different filename extensions. Suppose that you want to save the project under the name FIRST. To save the project, follow these steps:

1. Select File Save Project. Visual Basic opens the Save As dialog box for the form. Type **FIRST** and press Enter. Visual Basic adds the .FRM extension.
2. Visual Basic now displays the Save Project As dialog box. Type **FIRST** and press Enter to save

the project. Visual Basic adds the .MAK extension.

3. You can now exit Visual Basic and take a deserved rest.

***Review:*** With a few keystrokes and mouse clicks, you can create a fully working Visual Basic application. If you rerun the MYFIRST.MAK application, you will see that you can maximize, minimize, and resize the application's window. Visual Basic automatically adds a control button in the upper-left hand corner of the window with which you can control the application from the Windows system level.

# Homework

## General Knowledge

1. What is an event?
2. Why are GUI-based programming environments such as Windows event-driven?
3. True or false: Windows passes all events that happen in your program to Visual Basic.
4. What are properties?
5. What must you write to handle events?
6. Name two properties for any control that you can think of after reading this unit.
7. True or false: Forms have properties.
8. True or false: Forms are objects.
9. Which window do you use for changing property values?
10. What is program maintenance?
11. What advantage does using a three-letter prefix offer?
12. Why should you write Windows programs that look and work in a manner that is consistent with other Windows programs?
13. What is the name of the file that describes all new projects?
14. What is the name of the file that holds named constants?
15. What are named constants?
16. True or false: Visual Basic assigns well-named control names by default.
17. How can you open a Code window for a control's primary event property?
18. How are event procedures named?
19. What are the first and last statements in an event procedure called?
20. What Visual Basic command terminates a running program?
21. What does the Code window's Object: dropdown combo list contain?
22. How does the filename of the form differ from the name of the project?
23. What object does frmStartUp describe?
24. What object does cboNameChoice96 describe?

25. What object does cmdPrintIt describe?

## Find the Bug

1. Victor the Visual Basic programmer just got a new CD-ROM with tons of great fonts. Victor decides to use a different font for every word on his form. Can you help show Victor the light?
2. Describe what is wrong with each of these Name properties:
   . End
   B. 96JanSalesList
   C. cmdStar$
   D. July-List

## Extra Credit

Create a new project that contains two control buttons. Add a Beep statement to the wrapper of the first button s Click event procedure, and change the Caption property to Ring a Bell. (Beep is a command that rings the PC's bell.) Change the second command button's Caption property to Exit, and type End for the Click event procedure. Run the program and test your results.

# Visual Basic in 12 Easy Lessons

# Appendix B

## Reserved Words

Visual Basic contains a long list of commands, functions, and methods. These are known as *reserved words* or *keywords*. You must be careful not to assign procedures, controls, or variables any of these names or Visual Basic will generate an error.

**Table B.1. The reserved words.**

| |
|---|
| Abs |
| Access |
| AddItem |
| AddNew |
| Alias |
| And |
| Any |
| App |
| AppActivate |
| Append |
| AppendChunk |
| Arrange |
| As |
| Asc |
| Atn |
| Base |
| Beep |
| BeginTrans |
| Binary |
| ByVal |

| |
|---|
| Call |
| Case |
| CCur |
| CDbl |
| ChDir |
| ChDrive |
| Chr |
| Chr$ |
| CInt |
| Circle |
| Clear |
| Clipboard |
| CLng |
| Close |
| Cls |
| Command |
| Command$ |
| CommitTrans |
| Compare |
| Const |
| Control |
| Controls |
| Cos |
| CreateDynaset |
| CSng |
| CStr |
| CurDir$ |
| Currency |
| CVar |
| CVDate |
| Data |
| Date |
| Date$ |
| DateSerial |
| DateValue |
| Day |
| Debug |
| Declare |

| |
|---|
| DefCur |
| CefDbl |
| DefInt |
| DefLng |
| DefSng |
| DefStr |
| DefVar |
| Delete |
| Dim |
| Dir |
| Dir$ |
| Do |
| DoEvents |
| Double |
| Drag |
| Dynaset |
| Edit |
| Else |
| ElseIf |
| End |
| EndDoc |
| EndIf |
| Environ$ |
| EOF |
| Eqv |
| Erase |
| Erl |
| Err |
| Error |
| Error$ |
| ExecuteSQL |
| Exit |
| Exp |
| Explicit |
| False |
| FieldSize |
| FileAttr |
| FileCopy |

| |
|---|
| FileDateTime |
| FileLen |
| Fix |
| For |
| Form |
| Format |
| Format$ |
| Forms |
| FreeFile |
| Function |
| Get |
| GetAttr |
| GetChunk |
| GetData |
| DetFormat |
| GetText |
| Global |
| GoSub |
| GoTo |
| Hex |
| Hex$ |
| Hide |
| Hour |
| If |
| Imp |
| Input |
| Input$ |
| InputBox |
| InputBox$ |
| InStr |
| Int |
| Integer |
| Is |
| IsDate |
| IsEmpty |
| IsNull |
| IsNumeric |
| Kill |

| |
|---|
| LBound |
| LCase |
| LCase$ |
| Left |
| Left$ |
| Len |
| Let |
| Lib |
| Like |
| Line |
| LinkExecute |
| LinkPoke |
| LinkRequest |
| LinkSend |
| Load |
| LoadPicture |
| Loc |
| Local |
| Lock |
| LOF |
| Log |
| Long |
| Loop |
| LSet |
| LTrim |
| LTrim$ |
| Me |
| Mid |
| Mid$ |
| Minute |
| MkDir |
| Mod |
| Month |
| Move |
| MoveFirst |
| MoveLast |
| MoveNext |
| MovePrevious |

| |
|---|
| MoveRelative |
| MsgBox |
| Name |
| New |
| NewPage |
| Next |
| NextBlock |
| Not |
| Nothing |
| Now |
| Null |
| Oct |
| Oct$ |
| On |
| Open |
| OpenDataBase |
| Option |
| Or |
| Output |
| Point |
| Preserve |
| Print |
| Printer |
| PrintForm |
| Private |
| PSet |
| Put |
| QBColor |
| Random |
| Randomize |
| Read |
| ReDim |
| Refresh |
| RegisterDataBase |
| Rem |
| RemoveItem |
| Reset |
| Restore |

| |
|---|
| Resume |
| Return |
| RGB |
| Right |
| Right$ |
| RmDir |
| Rnd |
| Rollback |
| RSet |
| RTrim |
| RTrim$ |
| SavePicture |
| Scale |
| Second |
| Seek |
| Select |
| SendKeys |
| Set |
| SetAttr |
| SetData |
| SetFocus |
| SetText |
| Sgn |
| Shared |
| Shell |
| Show |
| Sin |
| Single |
| Space |
| Space$ |
| Spc |
| Sqr |
| Static |
| Step |
| Stop |
| Str |
| Str$ |
| StrComp |

| |
|---|
| String |
| String$ |
| Sub |
| System |
| Tab |
| Tan |
| Text |
| TextHeight |
| TextWidth |
| Then |
| Time |
| Time$ |
| Timer |
| TimeSerial |
| TimeValue |
| To |
| Trim |
| Trim$ |
| True |
| Type |
| TypeOf |
| UBound |
| UCase |
| UCase$ |
| Unload |
| Unlock |
| Until |
| Update |
| Using |
| Val |
| Variant |
| VarType |
| Weekday |
| Wend |
| While |
| Width |
| Write |
| Xor |

| |
|---|
| Year |
| ZOrder |

# Visual Basic in 12 Easy Lessons



# Project 2

## Visual Basic Programs

### Stop & Type

This lesson discussed the controls and properties of Visual Basic programs. The properties determine the look and behavior of the controls. The controls let the user interact with the program. When you write Visual Basic programs, you should follow as many standards as you can so that you can minimize any subsequent maintenance. You saw that the AUTOLOAD.MAK file determines how new projects will behave. You also learned

- To use good judgment when you design applications so that you do not use several controls that would only make the application look too busy.
- To use standard three-letter abbreviations when you name controls so that you know from its name exactly what kind of control you are working with.
- To follow Windows programming standards so that your application looks and behaves like other Windows applications. That way, the user will be familiar with your programs and will be more likely to learn and use the programs that you write.

The projects at the end of each lesson in this book show the creation of a Visual Basic program. They demonstrate the concepts taught in each unit. You have only scratched the surface so far. Beginning with the next lesson, you will start learning the details of controls and properties. So far, you have seen only how to place and activate the label and command button controls. There is much more to learn about these and the other controls. For this project, focus on following the instructions to gain more insight and practice in using Visual Basic.

#### The Program's Description

Figure P2.1 shows the first screen of the application described in this project. The application is simple and easy to implement if you follow the instructions that come next.

**Figure P2.1. Getting ready to view different properties.**

The purpose of this program is to show you the various properties that are possible for labels. Lesson 3 explores the label control's properties in detail, but you are already versed enough in Visual Basic to understand the properties demonstrated by this program.

## The Program's Action

The program first displays a label in the center of the screen. The label's border shows that the label is wide. The text in the center of the label is small, and the label's background color is white. Click the Next Property command to see a different set of properties.

The property changed by your first command button click is the alignment. A label can be left-justified, centered, or right-justified inside its placed width. The label began as a centered label, but you now see a left-justified label.

Click Next Property again. The label takes on the right-justified alignment property.

Clicking the Next Property button once more returns the label to the centered alignment and changes the property's font size to a big font. As you click the Next Property button, you will see that the label updates to show you the changed property.

To see the label's background color change, click Next Property again. The background of the label turns bright green. (Wear your sunglasses.)

Click the Next Property command button once more to see the final label property change. As Figure P2.2 shows, the width of the label shrinks to a smaller size. When you click the Next Property command button again, the label returns to its original size, font, and color property values.

**Figure P2.2. The width property got smaller.**

---

**Note:** This project's application contains a command button event procedure that changes all the properties whenever the user clicks the Next Property command button. This project does not list the code for that event procedure because the code would make absolutely no sense at this point in the book. If you really want to see the code, scroll through the Code window after you start the program.

---

## That's It for Now

You can now exit the program by clicking the Exit command button. The application quits running, and you will return to the Visual Basic development environment.

# Visual Basic in 12 Easy Lessons



# Lesson 3, Unit 5

## Using Labels, Buttons, and Text Boxes

### What You'll Learn

- [lb] The focus and controls
- [lb] The command button
- [lb] Label Properties
- [lb] Text box controls
- [lb] Setting properties

This unit explores labels, command buttons, and text boxes in detail. It explains all the properties associated with those controls. The important thing to remember as you wade through the tables of properties is that you need to master these properties if you are going to master Visual Basic. Unless you understand how to set up each property to respond in the manner your application requires, you cannot use Visual Basic effectively or create the outstanding applications that Visual Basic is capable of creating. Once you learn about the properties available for these controls, the next unit will explain all

the event procedures that are possible with the command button, label, and text box controls.

---

**Note:** From this unit through the end of the book, each lesson's project contains a complete application that uses the concepts taught in the two units. Each unit is descriptive, and the project takes what you learned in the two units and shows you what a real-world application of the material looks like. You get the best of both worlds of teaching: The theory comes in the units, and the practical applications appear in the end-of-lesson projects.

---

## Focus In on Controls

***Concept:*** As you learn about controls, you often hear the term *focus*. Learning about focus now saves you a lot of trouble later.

---

*Definition: Focus* refers to the control that is currently highlighted.

---

The control with the focus is the next control that will accept the user's response. Most Windows users instinctively understand focus even though very few have thought much about focus. The control with the focus is always the control that is highlighted. The focus often moves from control to control as the user works. The focus determines where the next action will take place.

---

**Tip:** In a way, a control's focus is much like a word processor's text cursor. The focus enables the user to know where the next action will take place unless he changes the focus, just as a text cursor tells the user where the next character will appear unless he moves the text cursor to a different location.

---

Only one control can have the focus at any one time, and not every control can get the focus. Figure 5.1 shows a form with four command buttons. The Third command button has the focus. You can tell this because the Third command button is highlighted, whereas the other command buttons are not. Not only is the button highlighted, but there is a dotted line around its caption.

**Figure 5.1. The Third command button has the focus.**

---

*Definition:* The *focus order* determines the control next in line for the focus.

---

Every form has a focus order that determines the next control that will receive the focus. In Figure 5.1, if

the user presses Enter, the figure's Third command button will depress only because it has the focus. If the user pressed Tab before pressing Enter, the next control in sequence to receive the focus would be highlighted. If the Second command button were next in sequence to gain the focus, the user's Tab press would highlight the Second command button.

When you see a dialog box such as a File Open dialog box, the OK button is almost always the command button that has the focus. You can press Enter to select the OK command button, click a different command button (such as a Cancel command button), or press Tab until the command button that you want has the focus.

Through property settings, you can determine the focus order and whether a control can receive the focus. Your form might have some controls that you don't want the user to be able to highlight; therefore, you prevent them from getting the focus.

***Review:*** The focus determines the control that triggers next. If a command button has the focus, it is highlighted and is the next button chosen if the user presses Enter. If a text box has the focus, it will receive the next keystroke's character even if there are five other text box controls on the form at the same time. If the user needs to enter text in a certain text box that does not have the focus, he must press Tab until the text box gets the focus, or he must click the text box that he wants with the mouse cursor.

## The Command Button

***Concept:*** The command button is the cornerstone of most Windows applications. With the command button, your user can respond to events, signal when something is ready for printing, and tell the program when to terminate. Although all command buttons work in the same fundamental way they visually depress when the user clicks them, and they trigger events such as Click events numerous properties for command buttons uniquely define each one and its behavior.

You are already familiar with several command button properties. You have seen that command buttons vary in size and location. Table 5.1 describes all the command button properties. You might want to open a new project, add a command button to the Form window by double-clicking the command button on the toolbox, and press F4 to scroll through the Properties window.

Several of the property settings can accept a limited range of numeric values. Most of these values are named in CONSTANT.TXT, and they are mentioned in this book as well. Some property settings accept either True or False values, indicating a yes or no condition. For example, if the Cancel property is set to True, that command button is the cancel command button and all other command buttons must contain a False Cancel property because only one command button on a form can have a True Cancel property.

---

**Tip:** As you read through the property tables in this book, familiarize yourself not only with the purpose of each property but also with its name. When you write code, you need to refer to these properties by their full and exact names.

---

**Table 5.1. Command button properties.**

| *Property* | *Description* |
|---|---|
| BackColor | The command button is one of the few controls for which the background color property means very little. When you change the background color, only the dotted line around the command button's caption changes color. |
| Cancel | If True, Visual Basic automatically clicks this command button when the user presses Esc. Only one command button can have a True Cancel property value at a time. All command buttons initially have their Cancel property set to False. |
| Caption | The text that appears on the command button. If you precede any character in the text with an ampersand (&),it acts as the access key. Therefore, the access key for a command button with a Caption property set to E&xit is Alt+X. The default Caption value is the command button's Name value. |
| Default | The command button with the initial focus when the form first activates has a Default property setting of True. All command buttons initially have False Default property values until you change one of them. |

*Definition:* An *icon* is a picture on the screen.

| | |
|---|---|
| DragIcon | The icon that appears when the user drags the command button around on the form. |

Because you only rarely enable the to user move a command button, you won t use the Drag... property settings very much.

| | |
|---|---|
| DragMode | Contains either 1 for manual mouse dragging requirements the user can press and hold the mouse button while dragging the control or 0 (the default) for automatic mouse dragging the user cannot drag the command button, but through code you can initiate the dragging if needed. |

| | |
|---|---|
| Enabled | If True (the default), the command button can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | If True (the default), the Caption displays in boldfaced characters. |
| FontItalic | If True (the default), the caption displays in italicized characters. |
| FontName | The name of the style of the command button caption. You typically use the name of a Windows True Type font. |

(margin)*Definition:* A *point* is 1/72 of one inch.

| | |
|---|---|
| FontSize | The size, in points, of the font used for the command button's caption. |

| | |
|---|---|
| FontStrikethru | If True (the default), the caption displays in strikethrough letters. In other words, characters have a line drawn through them. |
| FontUnderline | If True (the default), the caption displays in underlined letters. |
| Height | The height of the command button in twips. |
| HelpContextID | If you add advanced context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| Index | If the command button is part of a control array, the Index property provides the numeric subscript for each particular command button. (See Lesson 6.) |
| Left | The number of twips from the left edge of the Form window to the left edge of the command button. |
| MousePointer | The shape to which the mouse cursor changes if the user moves the mouse cursor over the command button. The possible values range from 0 to 12 and represent the different shapes that the mouse cursor can take on. (See Lesson 12.) |
| Name | The name of the control. By default, Visual Basic generates the names Command1, Command2, and so on, as you add subsequent command buttons to the form. |
| TabIndex | The focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the TabIndex values of the controls. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, the user can press Tab to move the focus to this command button. If False, the command button cannot receive the focus. |
| Tag | Not used by Visual Basic. The programmer can use it for an identifying comments applied to the command button. |
| Top | The number of twips from the top edge of a command button to the top of the form. |
| Visible | True or False, indicating whether the user can see and, therefore, use the command button. |
| Width | The width of the command button in twips. |

**Warning:** Table 5.1 lists only those command button properties that you can initialize and change in the Properties window. Other command button properties that you can change using Visual Basic code are available only at runtime.

***Review:*** There are many command button properties. As you can see from Table 5.1, learning which properties are available tells you a lot about what you can do with command buttons later.

# Label Properties

***Concept:*** The label holds text on the form. Although there are several ways to display text, the label control enables you to post messages on the form that you can change by means of Visual Basic code. The user, however, cannot change the value of a text control.

Label controls are vital to Visual Basic applications because you are always putting text on the form for the user to read. Here are some of the uses for the label control:

Titles (boxed and unboxed)

Data descriptions

Color warning messages

Graphic descriptions

Instructions

Labels are extremely easy to place and initialize. If you don't want a message to appear in a label when the user first starts the application, be sure to delete all the text from the label's Caption property. Table 5.2 lists the properties available for labels within the Properties window.

---

**Tip:** Notice that many of the properties in Table 5.2 match those in Table 5.1. Many properties, such as Width and Height, apply to several different controls. Other properties, such as AutoSize, apply to only a few controls.

---

**Table 5.2. label properties.**

| *Property* | *Description* |
|---|---|
| Alignment | Set to 0 for left-justification (the default), 1 for right-justification, or 2 for centering the Caption within the label. Putting a border around the label or shading the label a color often makes the justification stand out. |
| AutoSize | If True, the control automatically adjusts its own size to shrinkwrap around the contents of its caption. If False (the default), the control clips off the right of the text if the label is not large enough to hold the entire caption. |

---

*Definition:* A *hexadecimal number* is a base-16 number.

---

| | |
|---|---|
| BackColor | The background color of the label. It is a hexadecimal number that represents one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you are ready to set the BackColor property. The default background color is the same as the form's default background color. |

| | |
|---|---|
| BackStyle | If set to 0, meaning transparent, the form s background color comes through the label s background. If set to 1 (the default), the label's background color hides the form behind the label. |
| BorderStyle | Either 0 (the default) for no border or 1 for a fixed single-line border. |
| Caption | The text that appears on a label. |
| DragIcon | The icon that appears when the user drags the label control around on the form. |

Because you only rarely enable the to user move a label control, you won t use the Drag... property settings very much.

| | |
|---|---|
| DragMode | Contains either 1 for manual mouse dragging requirements the user can press and hold the mouse button while dragging the control or 0 (the default) for automatic mouse dragging the user cannot drag the label control, but through code you can initiate the dragging if needed. |

| | |
|---|---|
| Enabled | If True (the default), the label control can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | If True (the default), the Caption displays in boldfaced characters. |
| FontItalic | If True (the default), the caption displays in italicized character. |
| FontName | The name of the label control s style. You typically use the name of a Windows True Type font. |
| FontSize | The size, in points, of the font used for the label control's caption. |
| FontStrikethru | If True (the default), the caption displays in strikethrough letters. In other words, the characters have a line drawn through them. |
| FontUnderline | If True (the default), the caption displays in underlined letters. |
| ForeColor | The color of the text inside the caption. |
| Height | The height of the label control in twips. |
| Index | If the label control is part of a control array, the Index property provides the numeric subscript for each particular label control. (See Lesson 6.) |
| Left | The number of twips from the left edge of the Form window to the left edge of the label control. |

*Definition: DDE* stands for *D*ynamic *D*ata *E*xchange.

| | |
|---|---|
| LinkItem | Contains the data to be passed to an advanced DDE application. |

| | |
|---|---|
| LinkMode | Set to 0 (the default) for no DDE allowance, 1 for automatic DDE allowance, 2 for a code-based DDE, or 3 for a code-based notify requirement. |
| LinkTimeout | The count of tenths of seconds that a sent DDE message is to wait for a response. |
| LinkTopic | Specifies the source application and topic for a DDE application. |
| MousePointer | The shape to which the mouse cursor changes if the user moves the mouse cursor over the label control. The possible values range from 0 to 12 and represent the different shapes that the mouse cursor can take on. (See Lesson 12.) |
| Name | The name of the control. By default, Visual Basic generates the names Label1, Label2, and so on, as you add subsequent label controls to the form. |
| TabIndex | The focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing value of the TabIndex of the control. No two controls on the same form can have the same TabIndex value. |
| Tag | Not used by Visual Basic. The programmer can use it for identifying comments applied to the label control. |
| Top | The number of twips from the top edge of a label control to the top of the form. |
| Visible | True or False, indicating whether the user can see and, therefore, use the label control. |
| Width | The width of the label control in twips. |
| WordWrap | If True, the text wraps to hold the entire caption. If False (the default), the text does not wrap but is truncated to fit the caption. |

**Warning:** Table 5.2 lists only those label properties that you can initialize and change in the Properties window. Other label properties that you can change using Visual Basic code are available only at runtime.

Rarely, and more probably never, will you have to specify every property value when you create a new control. Most of the default values work well without any modification.

***Review:*** The label properties display text on the form that the user can read.

## Text Box Controls

***Concept:*** Text box controls display default values and accept user input. Text box controls enable you to determine how the user enters data and responds to questions and controls that you display.

When you display a text box on a form, you give the user a chance to accept a default value the text box's

initial Text property or to change it to something else. The user can enter text of any data type numbers, letters, and special characters. He can scroll left and right by using the arrow keys, and he can use the Ins and Del keys to insert and delete text within the text box control.

Most of the text box's properties work like the label control's properties. Unlike the label, however, the text box properties describe data-entry properties so that the control can deal with user input instead of simple text display. Table 5.3 describes the property values for the text box control.

**Table 5.3. Text box properties.**

| *Property* | *Description* |
|---|---|
| Alignment | Set to 0 for left-justification (the default), 1 for right-justification, or 2 for centering the Caption within the text box. If MultiLine contains False, Visual Basic ignores the Alignment setting. |
| BackColor | The background color of the text box. It is a hexadecimal number that represents one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you are ready to set the BackColor property. The default background color is the same as the form's default background color. |
| BorderStyle | Either 0 (the default) for no border or 1 for a fixed single-line border. |
| DragIcon | The icon that appears when the user drags the text box around on the form. |

Because you only rarely enable the user to move a text box, you won t use the Drag... property settings very much.

| | |
|---|---|
| DragMode | Contains either 1 for manual mouse dragging requirements the user can press and hold the mouse button while dragging the control or 0 (the default) for automatic mouse dragging the user cannot drag the text box, but through code you can initiate the dragging if needed. |

| | |
|---|---|
| Enabled | If True (the default), the text box can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | If True (the default), the Text displays in boldfaced characters. |
| FontItalic | If True (the default), the Text displays in italicized characters. |
| FontName | The name of the text box's style. You typically use the name of a Windows True Type font. |
| FontSize | The size, in points, of the font used for the text box control's Text value. |
| FontStrikethru | If True (the default), the text value displays in strikethrough letters. In other words, the characters have a line drawn through them. |
| FontUnderline | If True (the default), the text value displays in underlined letters. |
| ForeColor | The color of the text inside the Text property. |

| | |
|---|---|
| Height | The height of the text box in twips. |
| HelpContextID | If you add advanced context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| HideSelection | Keeps text highlighted even when the text box loses its focus. |
| Index | If the text box is part of a control array, the Index property provides the numeric subscript for each particular text box. (See Lesson 6.) |
| Left | The number of twips from the left edge of the Form window to the left edge of the text box. |
| LinkItem | Contains the data to be passed to an advanced DDE application. |
| LinkMode | Set to 0 (the default) for no DDE allowance, 1 for automatic DDE allowance, 2 for a code-based DDE, or 3 for a code-based notify requirement. |
| LinkTimeout | The count of tenths of seconds that a sent DDE message is to wait for a response. |
| LinkTopic | Specifies the source application and topic for a DDE application. |
| MaxLength | If set to 0 (the default), the limit of the Text value can be as great as approximately 32,000 characters. Otherwise, the MaxLength specifies how many characters the user can enter in the text box. |
| MousePointer | The shape to which the mouse cursor changes if the user moves the mouse cursor over the text box. The possible values range from 0 to 12 and represent the different shapes that the mouse cursor can take on. (See Lesson 12.) |

---

*Definition:* A *carriage return* character sends the text cursor to the next line.

---

| | |
|---|---|
| MultiLine | If True, the text box can display more than one line of text. If False (the default), the text box contains a single, and often long, line of text. The text can contain a carriage return. |

| | |
|---|---|
| Name | The name of the control. By default, Visual Basic generates the names Text1, Text2, and so on, as you add subsequent text boxes to the form. |
| PasswordChar | If you enter a character, such as an asterisk (*) for the PasswordChar, Visual Basic does not display the user's text but instead displays the PasswordChar as the user types the text. Use text boxes with a PasswordChar set when the user needs to enter a password and you don't want others looking over his shoulder to peek at the password. Figure 5.2 shows a password-entry form that uses an asterisk for the password character. Even though the text box receives the user's actual typed characters, the screen displays only the PasswordChar typed. |
| ScrollBars | Set to 0 (the default) for no scroll bars, 1 for a horizontal scroll bar, 2 for a vertical scroll bar, or 3 for both kinds of scroll bars. |
| TabIndex | The focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the value of the TabIndex of the control. No two controls on the same form can have the same TabIndex value. |

| | |
|---|---|
| TabStop | If True, the user can press Tab to move the focus to this label control. If False, the label control cannot receive the focus. |
| Tag | Not used by Visual Basic. The programmer can use it for identifying comments applied to the text box. |
| Text | The initial value that the user sees in the text box. The default value is the name of the control. The value continues to update as the user enters new text at runtime. |
| Top | The number of twips from the top edge of a text box to the top of the form. |
| Visible | True or False, indicating whether the user can see and, therefore, use the text box. |
| Width | The width of the text box in twips. |

**Figure 5.2. The password character appears instead of the characters that the user types.**

---

**Warning:** Table 5.3 lists only those text box properties that you can initialize and change in the Properties window. Other text box properties that you can change using Visual Basic code are available only at runtime.

---

***Review:*** The text box control enables you to display default values and to accept the changes and additions that the user makes to those values when he enters text from the keyboard. Once the user enters text in a text box control, you can check the Text property to access that text.

## Setting Properties

***Concept:*** The Properties window makes its easy for you to set properties when you place controls on the form and build your application.

The Properties window contains a list of every property that you can specify at *design time*. For example, there are 25 command button properties refer to Table 5.1 for the complete list that you can set when you place a command button on a form at design time. You can also read and change many of them when you run the program. Three properties that you can set at runtime through code are not available at design time. There are additional properties that you can set at runtime through code but not set at design time.

The Properties window offers several ways of setting property values. Figure 5.3 shows the Properties window for a label control. Notice that the Properties window contains a scrolling list of properties available at design time for labels, a data entry and dropdown list for setting property values, and a dropdown list for selecting another control's properties.

**Figure 5.3. The Properties window offers you several ways to change property values.**

As you found out in the previous unit, you can change a property value by clicking the property s row in the Properties window and entering a new value. The window s data entry text box receives the new property as you type the name. If you clicked the Name property and typed a new name, you would see the name in the data-entry text box.

The Properties window also offers you multiple-choice selections from which to choose if a property can assume a limited number of values. For example, the MousePointer property can hold the only values 0-Default through 7-Size NS. These are the various shapes in which the mouse can appear. Instead of typing one of these values, simply click the MousePointer property and then the down arrow that opens the data-entry text box's dropdown list, shown in Figure 5.4. You can select one of the values from the list without having to type a value.

**Figure 5.4. Visual Basic offers a choice of property values.**

---

**Note:** You don't have to select a control first to see its properties in the Properties window.

---

Some programmers prefer to place several controls on a form before they set any property values. Once you place several controls on the form, each control has its own set of properties that you can set from the Properties window. Use the window's control dropdown selection box to display all the controls in the application, as shown in Figure 5.5. Select from the list of controls that drops down to see the control's property settings appear in the Properties window.

**Figure 5.5. Select the control whose properties you want changed.**

---

**Tip:** The Properties window s dropdown list of controls shows the type and the name of each control.

---

---

*Definition:* A *palette* is a set of colors from which you choose.

---

When you set a color property, such as a label's BackColor property, you can type the hexadecimal color value, but it is much easier to select the color from a palette of colors that Visual Basic displays for you. If you clicked the BackColor property of a control such as a label, the Properties window data-entry text box changes to ellipses (...) instead of the arrow that normally accompanies a dropdown box. When you click the ellipses, Visual Basic displays a color selection palette like the one shown in Figure 5.6. Click one of the palette's colors to select it for the BackColor property. Visual Basic assigns the appropriate color hexadecimal value to match the color that you select.

**Figure 5.6. Select a color from the palette of colors.**

***Review:*** The Properties window offers all kinds of helpful shortcuts that you can use when you set property values for your form s controls. You can move from control to control and set properties along the way by choosing the control from the Properties window. You also can select a value for properties that is based on a fixed list of possible values.

# Homework

## General Knowledge

1. What is meant by *focus*?
2. What is the focus order?
3. Which property controls the focus order?
4. Which property is common for all controls and names the controls?
5. True or false: You can set or change all properties at program design time.
6. True or false: Only one control can have the focus at any one time.
7. If a property can accept only a limited range of values, which text file holds descriptions for those ranges?
8. What is an icon?
9. What meant by the measurement term *point*?
10. What is a palette?
11. Which of the following can the user enter in text box controls?
    . Numbers
    B. Letters
    C. Special characters
    D. All of the above
12. Which property disables a text box control from triggering event procedures?
13. What is the range of values that the Alignment property can hold?
14. True or false: You often must set most of the property values every time you place a control on a form.
15. What is a carriage return character?
16. What does *DDE* stand for?
17. True or false: A control can appear on the form but not be seen by the user.
18. Which four properties available for most controls describe the two measurement indicators to the right of the toolbar?

## Write Code That...

1. What is the best control for titles and instructions?
2. Which text box property setting would you use for a secret message that the user must enter?

3. Suppose that you were writing a Visual Basic program that needs to perform a specific task when the user presses Esc. Which property do you have to set?

## Find the Bug

1. Julie is unhappy. She just began programming in Visual Basic so she can use some help to overcome her frustration. Julie places all her controls on the form before setting property values for the controls. The only problem that Julie has is that she thinks she must close the Properties window, highlight another control, and display the Properties window once again to set that control's property settings. Show Julie a better way.

## Extra Credit

1. Even though only one control at a time can have the focus, there is a property that can make the user think that a text box still has the focus when the focus is really elsewhere. What is the name of this text box property?

Which text box control property works the most like the Caption property of a command button or label?

# Visual Basic in 12 Easy Lessons



# Lesson 3, Unit 6

## Polishing Forms and Controls

### What You'll Learn

- [lb] Properties of the form
- [lb] Advanced labels
- [lb] Scrolling text boxes
- [lb] Using focus to control text boxes
- [lb] Control events

This unit continues the work begun in Unit 5. It explains more about the form and its property settings. In addition, this unit delves further into label and text box controls by showing you some of the more advanced uses of those controls. Not only will you learn more about the property settings, but you will also learn what events are possible for these controls.

By the time you complete this unit, you will know virtually everything there is to know about forms, command buttons, labels, and text boxes. The project at the end of this lesson demonstrates these controls in action. It will give you a better sense of the ways in which you can implement them.

# Properties of the Form

***Concept:*** The form is yet another Visual Basic object. As such, it has property settings that you can set and change while you design the application and during the program's execution. This section explains all the form's property settings in detail.

Table 6.1 describes the property settings of the form that appear in the Properties window when you click the Form window and press F4. The form has more properties than the command button, label, and text box controls, whose properties you saw in the previous unit. As with all control property values, you never need to worry about all these properties at once. Most of the time, the default values are satisfactory for your applications.

**Table 6.1. Properties of the form.**

| *Property* | *Description* |
|---|---|
| AutoRedraw | If True, Visual Basic automatically redraws graphic images that reside on the form when another window hides the image or when the user resizes the object. If False (the default), Visual Basic does not automatically redraw. |
| BackColor | The background color of the form. You can enter a hexadecimal Windows color value or select from the color palette. |
| BorderStyle | Set to 0 for no border or border elements such as a control menu or minimize and maximize buttons, 1 for a fixed-size border, 2 (the default) for a sizable border, or 3 for a fixed-size border that includes a double-size edge. |
| Caption | The text that appears in the form's title bar. The default Caption is the Name of the form. |
| ClipControls | If True (the default), the Paint event a redrawing event triggered when graphic images are covered and then uncovered redraws the graphics. If False, only newly-exposed areas of the graphics are repainted. |
| ControlBox | If True (the default), the form contains a control button and control menu. If False, the form does not contain a control button and a control menu. |
| DrawMode | Contains 16 advanced settings that interact with drawing properties to produce special drawing effects. (See Lesson 11 for more information on graphics.) |
| DrawStyle | Contains seven advanced settings that determine the appearance of lines that you draw. |

---

*Definition:* A *pixel* is the smallest screen width possible on your monitor.

---

| | |
|---|---|
| DrawWidth | The width, in pixels, of lines drawn on the form. |

| | |
|---|---|
| Enabled | If True (the default), the form can respond to events. Otherwise, Visual Basic halts event processing for the form. |
| FillColor | The color value used to fill shapes drawn on the form. |
| FillStyle | Contains eight styles that determine the appearance of the interior patterns of shapes drawn on the form. |
| FontBold | Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command. |

| | |
|---|---|
| FontItalic | Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command. |
| FontName | Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command. |
| FontSize | Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command. |
| FontStrikethru | Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command. |
| FontTransparent | Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command. |
| FontUnderline | Has no effect on the form's Caption property, but does affect text that you eventually display on the form if you use the Print command. |
| ForeColor | The color of foreground text that you display on the form if you use the Print command. |
| Height | The form s height in twips. |
| HelpContextID | Provides the identifying number for the help text if you add advanced context-sensitive help to your application. |
| Icon | The picture icon that the user sees after minimizing the form. |
| KeyPreview | If False (the default), the control with the focus receives these events: KeyDown, KeyUp, and KeyPress before the form does . If True, the form receives the events before the focused control. |
| Left | The number of twips from the left edge of the screen to the left edge of the form. |
| LinkMode | Set to 0 (the default) for no DDE allowance, 1 for automatic DDE allowance, 2 for a code-based DDE, or 3 for a code-based notify requirement. |
| LinkTopic | Specifies the source application and topic for a DDE application. |
| MaxButton | If True (the default), the maximize button appears on the form at runtime. If False, the user cannot maximize the form window. |

---

*Definition: MDI* stands for *M*ultiple *D*ocument *I*nterface.

---

| | |
|---|---|
| MDIChild | If True, the form is a MDI form that is, a child form within a parent form. If False (the default), the form is not a MDI form. |

| | |
|---|---|
| MinButton | If True (the default), the minimize button appears on the form at runtime. If False, the user cannot minimize the form window. |
| MousePointer | The shape to which the mouse cursor changes if the user moves the mouse cursor over the form. The possible values range from 0 to 12 and represent the different shapes the mouse cursor can take on. (See Lesson 12.) |
| Name | The name of the form. By default, Visual Basic generates the name Form1. |
| Picture | A picture file that displays on the form's background. |
| ScaleHeight | The height of the form. ScaleMode determines the unit of measurement used. |
| ScaleLeft | The distance from the left of the screen to the left edge of the form. ScaleMode determines the unit of measurement used. |

| | |
|---|---|
| ScaleMode | Enables you to determine how to measure coordinates on the form. You can choose from eight values. The default unit of measurement is twips, indicated by 1. The other Scale... properties measure use twips. Table 6.2 describes the possible units of measurement. |
| ScaleTop | The distance from the top of the screen to the top edge of the form. ScaleMode determines the unit of measurement used. |
| ScaleWidth | The width of the form. ScaleMode determines the unit of measurement used. |
| Tag | Not used by Visual Basic. The programmer can use it for identifying comments applied to the form. |
| Top | The number of twips from the top edge of the screen to the top of the form. |
| Visible | True or False, indicating whether the user can see and, therefore, use the form. |
| Width | The width of the form in twips. |
| WindowState | Describes the startup state of the form when the user runs the program. If set to 0 (the default), the form first appears the same size as you designed it. If set to 1, the form first appears minimized. If set to 2, the form first appears maximized. |

ScaleMode enables you to determine how to measure coordinates on the form. You can choose from eight values. The default unit of measurement is twips. Table 6.2 describes the possible units of measurement.

**Table 6.2. The ScaleMode property values.**

| *Value* | *Description* |
|---|---|
| 0 | Customized values |
| 1 | Twips (the default) |
| 2 | Points |
| 3 | Pixels |
| 4 | A standard character that is 120 twips wide and 240 twips high |
| 5 | Inches |
| 6 | Millimeters |
| 7 | Centimeters |

***Review:*** You can customize your form in all kinds of ways. You can make it appear maximized or minimized. You can use colors and various styles. Most of the time, you want just a simple form with a caption that identifies the application; the only property values that you probably will have to modify are the Caption properties.

## Advanced Labels

***Concept:*** In the previous unit, you saw all the property values that you can set with labels. Some of the property values produce interesting effects, which are described in this section.

Suppose that you design a label that contains this long caption:

```
If a label's caption is too lengthy, you will need to adjust the label some way.
```

A label is rarely wide enough or tall enough to hold this caption. If you attempt to type text into a label's Caption property that is longer than what fits the size of the label, one of the following things can take place depending on how you have set up the label:

*Definition: Truncate* means to chop off.

1. The text might not fit inside the label, and Visual Basic truncates the text. Figure 6.1 shows the result.
   Set the AutoSize property to False if you want the label to remain the same size. The application assigns the text, and the label might not hold the entire caption.

**Figure 6.1. The label is not large enough.**

2. The label automatically expands downward to hold the entire caption in a multiline box. Figure 6.2 shows the result.
   Set both the AutoSize and WordWrap properties to True if you want the label to expand vertically to hold the entire caption that you assign at design time and during execution.

**Figure 6.2. The label resizes vertically.**

**Tip:** Set WordWrap to True before you set the AutoSize property to True. If you set AutoSize first, the label expands horizontally before you have a chance to set the WordWrap property.

**Warning:** Be careful about placing too many automatically-resizing labels. The labels might overwrite important information on the form if their captions are too long.

3. The label automatically expands across the screen to hold the entire caption in a long label control. Figure 6.3 shows the result.
   A long label like this is not necessarily incorrect. Depending on the length of the text that you assign to the label during the program's execution, there might be plenty of screen space to display long labels. To automatically expand the label horizontally, set the AutoSize property to True and leave WordWrap set to False. This is the default setting.

**Figure 6.3. The label is not tall enough.**

***Review:*** Putting captions in labels seems easy until you think about the effects that can occur if the label is too large or too small to hold the text. By using the property combinations described here, you can add automatically-adjusting labels for whatever text the labels need to hold.

## Scrolling Text Boxes

***Concept:*** By adding scroll bars to text boxes, you can give the user multiline text box capabilities. That way, the user can enter and edit long lengths of text without running out of room inside the text boxes.

The MultiLine property for text box controls determines whether or not the text box can contain one or more lines of text. The multiline text might be an initial default value that you store in the text box's Text property when you place the form on the control. The multiline text also might consist of the user's input when the program runs.

**Warning:** If you set the MultiLine property to True, you must also set the Scrollbars property to something other that

0-None. The user has to have a way to see the multiple lines of text inside the text box and the scroll bars give the user that ability.

---

Figure 6.4 shows a text box that contains scroll bars and a True value for the MultiLine property. As the user enters text in a box like this, he can press Enter to move to the next line in the box. He does not, however, have to press Enter just because the text happens to scroll to the right; the horizontal scroll bars enables him to scroll left and right. When the user wants to end each line in the text box though, he presses Enter to move the carriage return character to the next line.

**Figure 6.4. The label's scroll bars give the user more data-entry freedom.**

---

**Note:** You cannot enter an initial default value for a multiline text box. You can only initialize a text box with text that spans more than one line in the text box at runtime.

---

***Review:*** By setting the MultiLine and the ScrollBars properties, you can use multiline text boxes in your applications. Multiline text boxes respond to user input by accepting more than one line of text.

## Using Focus to Control Text Boxes

***Concept:*** Although access keys are not available for text boxes, you can use a little-known trick to supply access keystroke shortcuts for text box data entry.

As you begin to build Visual Basic applications, you will use text box controls to capture user input. Don't just throw a text box on a form and expect the user to know what to enter in it. In Figure 6.5, for example, the user does not know what data he should enter in the text box controls.

**Figure 6.5. Text boxes without labels confuse the user.**

You must label the text box with a label control that tells the user what you want. The application shown in Figure 6.6 is identical to the application shown in Figure 6.5, but the labels in front of each text box tell the user what kind of data to enter.

**Figure 6.6. The user now knows what data is expected.**

---

**Tip:** Add access shortcut keystrokes to labels that describe data-entry text boxes.

---

Suppose that the user fills in the six sales figures. He might want to go back and change an entry to fix a typing error. When you put text boxes in an application, think about giving your users a chance to correct their mistakes by providing them an access keystroke to each text box.

As you know, access shortcut keys are the Alt+Keystroke combinations that you can apply to controls such as command buttons. In Figure 6.6, the user can press Tab to the Exit command button, click the Exit command button with the mouse, or press Alt+X the access shortcut key for the command button. The underline indicates which letter provides the access.

Text box controls do not have captions, so you cannot directly add access keystrokes to text boxes. Nevertheless, you can add underlined access keystrokes to label captions. For example, suppose that you changed the first label in Figure 6.6 from January to January. The access keystroke for that label is Alt+J.

But wait, there's a problem. Labels cannot receive the focus! If a label contains an access keystroke and the user presses that access keystroke combination, Visual Basic knows that the focus cannot go to the label. It sends the focus to the next control in the TabIndex sequence. All controls contain a TabIndex property. A different numeric value appears in each control's TabIndex property. As you learned in the previous unit, the TabIndex property determines the focus order. Suppose that you assigned a TabIndex value of 0 to the January label and a TabIndex value of 1 to the text box to the right of January. When the user presses Alt+J, the focus goes to the text box because the label cannot receive a focus.

Therefore, after you place all the controls on a form and set their properties, go back to each control and make sure that each label contains a TabIndex value that is one less than the text box control that the label describes. Make sure, as well, that the overall TabIndex sequence is organized so that it sends the focus from control to control in the order you want as the user presses Tab. Figure 6.7 shows the six-month sales data-entry application in which each label has an access shortcut keystroke. The figure indicates the TabIndex value for each control. Given the TabIndex values, the focus goes directly to the May text box when the user presses Alt+Y.

**Figure 6.7. The TabIndex properties describe the access keystroke order.**

***Review:*** Although you cannot add access keys to text boxes, you can add access keys to the labels that describe text boxes. By doing so, you give your users shortcut access to any text box on the form.

## Introducing Control Events

***Concept:*** You know that when the user clicks command buttons and types text in text boxes, he triggers events that your program can capture. This section discusses the events available for the command button, label, and text box controls. Lesson 4 begins your study of the Visual Basic programming language, so you need to understand which events are possible as you write event procedures that respond to those events.

Here's another section full of tables! Nevertheless, they show you all the events available for the controls you have been learning. In the next lesson, you will begin to write code. The code that you write usually appears inside event procedures. You need to know which events are available, so that you can write the correct event procedures.

---

**Note:** As you learn new controls in subsequent lessons, you will find more tables properties and events. Enjoy.

---

Table 6.3 describes the events related to forms. Perhaps the most important form event is Load, which triggers whenever the user runs an application. Throughout this book, you will use the Load event to put startup code in applications so that the startup code executes immediately after the user runs the application and immediately before the form appears on the screen.

---

**Tip:** Remember that if you want to see what events are possible for a certain control, place the control on the Form window and double-click the control. Visual Basic opens a Code window. Open the Proc: dropdown combo box list to see a list of the events available for that control.

---

**Table 6.3. Form events.**

| *Event* | *Description* |
|---|---|
| Activate | Occurs when a form becomes the active window. In Visual Basic, the Activate event occurs after the Load event displays the form. |
| Click | Occurs when the user clicks the form with the mouse. |
| DblClick | Occurs when the user double-clicks the form with the mouse. |
| Deactivate | Occurs when another form becomes the active window. Not available in the Visual Basic Primer Edition. |
| DragDrop | Occurs when a drag operation over the form completes. |
| DragOver | Occurs during a drag operation over the form. |
| GotFocus | Occurs when the form receives the focus. |
| KeyDown | Occurs when the user presses a key and the KeyPreview property for the controls on the form is set to True. Otherwise, the control gets the KeyDown event. |
| KeyPress | Occurs when the user presses a key over the form. |
| KeyUp | Occurs when the user releases a key. |
| LinkClose | Occurs when a DDE operation terminates. |
| LinkError | Occurs when a DDE operation fails. |
| LinkExecute | Occurs when a DDE operation begins to execute. |
| LinkOpen | Occurs when a DDE operation begins. |
| Load | Occurs when the form loads and before it appears on the screen. |
| LostFocus | Occurs when the form loses the focus. |
| MouseDown | Occurs when the user presses the mouse button over the form. |
| MouseMove | Occurs when the user moves the mouse over the form. |
| MouseUp | Occurs when the user releases the mouse over the form. |
| Paint | Occurs when Visual Basic must redraw a form because another object overwrote part of the form and then the user moved the object and exposed the hidden part of the form. |
| QueryUnload | Occurs immediately before the application terminates. |
| Resize | Occurs when the user resizes the form. |
| Unload | Occurs when the form is unloaded using the Unload statement. |

---

**Warning:** Don't Table 6.3 scare you away from Visual Basic! You will use only a handful of these events in most of your programming work.

---

Notice that all the descriptions in Table 6.3 begin with the word *occurs*. Each of these table entries are events that occur as the result of a user or Windows action. Therefore, if you want to do something when the user clicks the form, you write code that performs the task that you want accomplished and you put that code inside the form's Click event procedure. If the form is named MyForm, the Click event procedure is named MyForm_Click(), as you learned in the Unit 4. You will start writing the code for event procedures in the next lesson.

Table 6.4 describes the events available for the command button controls that you place on forms.

**Table 6.4. Command button events.**

| *Event* | *Description* |
|---|---|
| Click | Occurs when the user clicks the command button with the mouse. |

| | |
|---|---|
| DragDrop | Occurs when a drag operation of the command button completes. |
| DragOver | Occurs during a drag operation of the command button. |
| GotFocus | Occurs when the command button receives the focus. |
| KeyDown | Occurs when the user presses a key and the KeyPreview property for any control on the form is set to False. Otherwise, the form gets the KeyDown event. |
| KeyPress | Occurs when the user presses a key over the command button. |
| KeyUp | Occurs when the user releases a key. |
| LostFocus | Occurs when the command button loses the focus to another control or to the form. |

Table 6.5 describes the events available for the label controls that you place on forms.

**Table 6.5. Label control events.**

| *Event* | *Description* |
|---|---|
| Change | Occurs when the label's Caption property changes. |
| Click | Occurs when the user clicks the label with the mouse. |
| DblClick | Occurs when the user double-clicks the label with the mouse. |
| DragDrop | Occurs when a drag operation of the label completes. |
| DragOver | Occurs during a drag operation of the label. |
| LinkClose | Occurs when a DDE operation terminates. |
| LinkError | Occurs when a DDE operation fails. |
| LinkNotify | Occurs when a DDE operation notifies the label with a changed message. |
| LinkOpen | Occurs when a DDE operation begins. |
| MouseDown | Occurs when the user presses the mouse button over the label. |
| MouseMove | Occurs when the user moves the mouse over the label. |
| MouseUp | Occurs when the user releases the mouse over the label. |

Note that no GotFocus event is associated with labels. This is because a label can never receive the focus.

Table 6.6 describes the events available for the text box controls that you place on forms.

**Table 6.6. Text box control events.**

| *Event* | *Description* |
|---|---|
| Change | Occurs when the text box's Text property changes. |
| DragDrop | Occurs when a drag operation of the text box completes. |
| DragOver | Occurs during a drag operation of the text box. |
| GotFocus | Occurs when the text box receives the focus. |
| KeyDown | Occurs when the user presses a key and the KeyPreview property for the controls on the form is set to True. Otherwise, the form gets the KeyDown event. |
| KeyPress | Occurs when the user presses a key over the text box. |
| KeyUp | Occurs when the user releases a key over the text box. |
| LinkClose | Occurs when a DDE operation terminates. |
| LinkError | Occurs when a DDE operation fails. |
| LinkNotify | Occurs when a DDE operation notifies the text box with a changed message. |

| | |
|---|---|
| LinkOpen | Occurs when a DDE operation begins. |
| LostFocus | Occurs when the text box loses the focus to another object. |

***Review:*** Each control has its own set of properties and command buttons. The tables in this unit complete the discussion of forms, command buttons, labels, and text boxes. You now know enough to work with these four fundamental Visual Basic objects. The next lesson builds on your knowledge by teaching you how to add code to event procedures.

## Homework

### General Knowledge

1. Is the form an object?
2. True or false: Label controls can have more than one line of text as long as you set the MultiLine property.
3. What is a pixel?
4. Which form property determines how the form appears when the user first sees the form?
5. What does *truncate* mean?
6. True or false: Suppose that you want to store a lot of text in a label. You must make the label large enough to hold the entire text if the user is to see all of it.
7. True or false: You can add both horizontal and vertical scroll bars to a label.
8. How can you ensure that a label does not expand to display a long value?
9. True or false: You can enter an initial multiline text value by using the Properties window.
10. True or false: A label can get the focus as long as you supply an access shortcut key for the label.
11. Which property determines the focus order?
12. What is perhaps the most important form event?
13. Which form event occurs first: Load or Activate?
14. What window can you use to see all the events for objects?
15. Why is there no GotFocus event for labels?

### Write Code That...

1. Suppose that you were writing a Change event procedure for a text box label called txtLastName. What you name the event procedure?
2. When the user presses a key, either the form gets the keystroke or a control gets the keystroke. What property determines which object gets the keystroke?
3. Suppose that you need to change a form's title in the form's title bar. Which property do you change the Caption property or the Name property?
4. Describe how you add access keystrokes to text boxes.

### Find the Bug

1. Why should you avoid putting too many self-sizing labels on the form at one time?
2. What happens if you set the AutoSize property to True before you set the WordWrap property to True?

## Extra Credit

What if you wanted to enter Property window settings in inches instead of twips, the default unit of measurement. How do you change the unit measurement to inches?

# Visual Basic in 12 Easy Lessons



# Project 3

## Getting Down to Business

### Stop & Type

In this lesson, you learned about all the properties and events that are possible with the form, label, text box, and command button objects. Visual Basic can handle virtually all the requirements that your application will need.

In this lesson, you saw the following:

- [1b] How to set and move the focus from object to object
- [1b] What property values exist for the application's objects
- [1b] Which events Visual Basic makes possible for the objects
- [1b] Why you sometimes must manage labels and text boxes by using the resizing and scroll bar properties

The project named PROJECT3.MAK contains this lesson's project. You should load PROJECT3.MAK and run the application so that you can follow along with the description of this project.

### The Program's Description

Figure P3.1 shows how the PROJECT3.MAK application looks as soon as you run the program. The project contains the following objects:

Three labels

A text box control that contains a vertical scroll bar

Two command buttons, each with its own access key

Notice that the top label contains an access key, Alt+T, that will send the focus to the text box.

**Figure P3.1. The project's opening screen.**

The purpose of this program is to request a message that you can enter in the text box. You can type a message as long or as short as you want. After you type the message, click the center command button or press Alt+S to send that text box message to the empty label below the center command button.

The label receiving the text box data resizes automatically to adjust to the size of the text. Therefore, if you type only a single word, the label shrinks to fit around that one word. If, instead, you type several lines of text, such as a poem, all of the lines of text appear in the label because the label grows to hold all the text.

## The Program's Action

Now type the words **Visual Basic** in the text box. Press Alt+S to send the contents of the text box to the label at the bottom of the screen. The label shrinks to hold the two words, as shown in Figure P3.2.

**Figure P3.2. Watch how the label resizes.**

Obviously, the command button triggers an event that takes all the text in the text box and sends a copy of it to the label at the bottom of the screen. You have yet to learn Visual Basic coding commands, so I'll save the discussion of event procedure code for the next unit.

The resizing label at the bottom of the screen contains the following Property window settings:

AutoSize: True

WordWrap: True

If the AutoSize property is not set to True, the label does not automatically shrink to fit the small text box value assigned to the label. Likewise, when you enter a larger message in the text box and press Alt+S to send it to the lower label, the label does not grow to accommodate the larger message. The WordWrap value of True ensures that the label grows vertically instead of expanding horizontally as a long, one-line label.

Try a different message. Press Alt+T to send the focus back to the text box. Press the Backspace key to erase the two words currently in the text box. Enter the following beautiful poem written several hundred years ago by a composer of lovely symphonies. (Okay, so *I* wrote the poem, but it is beautiful if you ask me.)

*Visual Basic is like music,*
*ringing in my ears.*
*When I think of the things it does,*
*my eyes fill up with tears.*

Press Alt+S to send the poem to the label. See how fast the label expands vertically to hold the full message. Figure P3.3 shows what your screen should look like.

**Figure P3.3. The label grows as large as necessary.**

******Begin Note******

---

**Note:** If you pressed Enter at the end of each poem line, your screen will match the one shown in Figure P3.3. Otherwise, your poem will be squeezed together a bit more than how it appears in Figure P3.3.

---

******End Note******

******Begin Warning******

**Warning:** Visual Basic is extremely sensitive. If you cringe when you read the author's poetry, Visual Basic will refuse to work for you ever again.

******End Warning******

## Close the Application

Until this lesson, you would not have been able to place a label that resized at runtime to fit automatically whatever text it received. Look again at the screen. You will see another trick taught by this lesson: You can send the focus to the text box by pressing the access key (Alt+T) for the label just above the text box. The only way to send the focus to the text box is to ensure that the TabIndex property for the label is one less than the TabIndex property for the text box. When Visual Basic realizes that the access key sends the focus to a label that cannot receive the focus, Visual Basic sends the focus to the next control as determined by the order of the objects' TabIndex values.

You can now exit the application and exit Visual Basic. Take a rest because you will begin writing your first Visual Basic textual code in the next unit.

# Visual Basic in 12 Easy Lessons



# Lesson 4, Unit 7

## Variables, Controls, and Math

### What You'll Learn

- Data Types
- Storing variables
- Mathematical expressions
- The Val() function

It's time to learn a foreign language: Visual Basic. Until now, you created programs by placing controls on the form and setting property values for them. Visual programming like that is at the heart of Visual Basic, and visual nature of Visual Basic is what separates it from the more traditional text-based programming languages.

As you will see in this lesson, there is more to Visual Basic than just a pretty face! If you learn the

text-based Visual Basic language in addition to how to place and interact with controls, you can write extremely powerful applications. The Visual Basic programming language the code often ties controls together so that your applications can analyze data and compute based on your data.

---

**Warning:** Visual Basic is one foreign language that is really not so foreign. You will see that its vocabulary consists of simple words that you are already familiar with.

---

## Data Types

***Concept:*** The Visual Basic language works with all kinds of data. Before you learn how to manipulate data, you will learn how to distinguish among the various data types.

Visual Basic manipulates and analyzes seven kinds of data. Table 7.1 describes the data types that Visual Basic supports. Some data falls into more than one category. For example, a dollar amount can be considered both a currency data type and a single data type. When you write a program, you need to decide which data type best fits your program's data values.

**Table 7.1. Visual Basic data types.**

| *Data Type* | *Description* |
|---|---|
| integer | Numeric values with no decimal point or fraction. integer values range from  32,768 to 32,767. |
| long | Integer values with a range beyond than that of integer data values. long data values range from  2,147,483,648 to 2,147,483,647. long data values consume more memory storage than integer values, and they are less efficient. The long data type is often called long integer. |
| single | Numeric values that range from  3.402823E+38 to 3.402823E+38. The single data type is often called single-precision. |
| double | single numeric values that range from  1.79769313486232E+308 to 1.79769313486232E+308. The double data type is often called double-precision. |
| currency | Data that holds dollar amounts from  $922,337,203,685,477.5808 to $922,337,203,685,477.5807. |
| string | Data that consists of 0 65,500 characters of alphanumeric data. Alphanumeric means that the data can be both alphabetic and numeric. string data values may also contain special characters such as ^%@. |
| variant | Used for data stored in controls and for date and time values. |

The following data values can take on the integer data type:

0

-9455

32766

You can also store these integer data types as long data types, although doing so wastes storage and time unless you plan to change the values later to extremely large or small integer numbers that require the long data type.

The following data values must be stored in a long data type:

32768

-95445

492848559

The following data values can take on the single data type:

0.01

565.32

-192.3424

9543.5645

6.5440E-24

Of course, you can store these data values in double storage locations as well. Use double data types only when you need to store extra large or small values.

The following data values take on the double data type:

-5968.5765934211133

4.532112E+92

928374.344838273567899990

---

What's that E Doing in the Numbers?: The E stands for *exponent*. Years ago, mathematicians grew weary of writing long numbers. They developed a shortcut notation called *scientific notation*. Visual Basic supports scientific notation. You can use scientific notation for single and double numeric data values. When you use scientific notation, you don't have to write or type, in the case of program writing extremely long numbers.
When a single or double numeric value contains the letter E, followed by a number, that number is written in scientific notation. To convert a number written in scientific notation to its non-abbreviated form, multiply the number to the left of E by 10 raised to the power indicated by the number to the right of E. Therefore, 2.9876E+17 means to multiply 2.9876 by 10 raised to the 17th power, or $10^{17}$. $10^{17}$ is a

very large number it is 10 followed by 16 zeroes. The bottom line is that 2.9876E+17 means the same as 29876 followed by 13 zeros.
If a number uses a negative exponent, you multiply the number to left of E by 10 raised to negative power. Therefore, 2.9876E-17 means the same as 2.9876 multiplied by $10^{-17}$ or 2.9876 divided by $10^{17}$ which turns out to be a small number indeed.

---

---

**Note:** Don t consider yourself a mathematician? That's okay Visual Basic calculates all your math for you when you learn how to write programs with this book. Typically, only scientific and engineering programmers need to use scientific notation, and they feel right at home using it. Table 7.1 used scientific notation because there is no practical way of displaying the low or high range for double data values with 307 zeros in a the number.

---

The following data values can take on the currency data type:

123.45

0.69

63456.75

-1924.57

The currency data type can accept and track numeric values to four decimal places. However, you typically store dollar and cent values in the currency storage locations, and these kinds of values require only two decimal places.

---

**Note:** Visual Basic uses the Windows International settings found in the Control Panel if you want to take the time to look to track values such as dates, times, and currency. Therefore, if you have Windows set up for a country, such as France, that uses a comma instead of the decimal point as it is used in America, Visual Basic will support the comma when it displays your values.

---

Never use a currency data value along with a dollar sign. In other words, the following cannot be a currency value even though it looks like one:

$5,234.56

Visual Basic does not want you to use a dollar sign or commas in numeric data of any kind unless, of course, your country uses the comma for the fractional portion of numbers. If your data contains anything other than numbers, a plus sign, a minus sign, or an exponent, Visual Basic cannot treat the data as if it were numeric; therefore, it cannot perform mathematical calculations with the data. Instead, Visual Basic treats the data as string data.

The following data values take on the string data type:

"London Bridge"

"1932 Sycamore Street"

"^%#@#$%3939$%^&^&"

Notice the quotation marks around the three string values. string values require the quotation marks, which tell Visual Basic where the string begins and ends. If you wanted to embed spaces at the beginning or end of a string, you indicate those spaces by including them inside the quotation marks. For example, here are three different strings, each with a different set of embedded spaces: " house", "house ", " house ". You cannot embed quotation marks directly inside a string; Visual Basic thinks the string ends as soon as it comes to the second quotation mark, even if it is not the actual end of the string.

---

**Note:** You just caught me in a lie. Actually, there is a way to embed quotation marks in string data, but doing so is messy and requires the use of a built-in function that you will learn about in Lesson 7.

---

The following data values can take on the variant data type:

"^%#@#$%3939$%^&^&"

123.45

4.532112E+92

21

03-Mar-1996

Do these values look familiar? The variant data type can hold data of *any other data type*. Think about the kind of data stored in label controls. You often want to display numbers, dollar amounts, times, and dates in labels on a form. You can always store variant data in these controls. The data that comes *from* these controls is also variant.

Visual Basic offers you several ways to convert data from one type to another. For example, "34" is a string value because of the quotation marks. You cannot perform math on string data. However, if you store a string that contains a valid number and decide that you need to perform math on the value, Visual Basic provides a built-in data-conversion routine that enables you to convert the string to a number so that you can do math with its value. The data type differences exist so that you can adequately store and describe certain data values.

---

**Note:** Visual Basic uses the variant data type for date and time values, as you will learn in Lesson 7, "Functions and Dates."

---

---

*Definition:* A *constant* is a data value that does not change.

---

The data values that you saw in this section are all constants. A constant does not change. For example, the number 7 is always 7, and never means anything except 7. There are times when you must explicitly state exactly what kind of constant a data value is so that Visual Basic works properly with the data. For example, 7 is a constant, and neither you nor Visual Basic can tell whether 7 should be stored in an integer or a long integer data location.

variant data is the only kind of data that Visual Basic changes, when needed, to suit the situation at hand. In other words, if you store a long integer value in a variant data area, you can perform math with the variant value. If you store a string in a variant data area, you'll be able to perform string operations on the data.

If you run into a situation where a particular data type constant is required, you don t have to use of the default variant data type. You can explicitly state what data type a certain constant is. By adding one of the data type suffix characters listed in Table 7.2 to your constant data, you supply the data type when you use the constant.

**Table 7.2. Data type suffix characters.**

| *Suffix* | *Data Type* | *Example* |
|---|---|---|
| & | long | 14& |
| ! | single | 14! |
| # | double | 14# |
| @ | currency | 14@ |

You don t need the suffix characters very often. Most of the time, you can get by with assigning constants to controls and data areas without needing to add a suffix character to the end of the name. Remember that Visual Basic assumes the data constant is a variant data type unless you use a suffix character to override that assumption. Because variant data can convert to any other data type, you almost always can use the default variant data type.

Your programs are made up of the data that changes and constant data, which remains the same. Many times you must specify constant values, such as the number of months in a year, the number of hours in a day, or a company name. Other times, you work with values that change over time, such as an employee's salary or an inventory valuation. The next section discusses how to set up and use data values that change over time.

---

**Tip:** Is your head spinning yet? If you have never programmed before and this book assumes that you never have this talk about data types has jumped right into hard material that seems technical and frustrating. You really haven't seen a reason to use these data types yet. The next section helps explain why these data types are so important. So don't give up just yet not that you were going to, but I just wanted to be sure!

---

***Review:*** All Visual Basic data falls in one of seven data types. The kind of data you work with and the range of the data values that you use determine which data type a value will take. The various data types enable you to categorize data and to perform specific functions on certain kinds of data.

# Variable Storage

***Concept:*** As you know, your computer has memory. You can store data values in memory locations that your Visual Basic program then operates on. When you define variables, you reserve places in memory for data variables, name those variables, and decide what data type those variables will hold.

---

*Definition:* A *variable* is a named storage location that holds data that changes.

---

Throughout a program, you need to store data in memory locations while you compute with that data. Those memory locations are called *variables*. A variable, unlike a constant, can change. In other words, you can store a number in a variable early in the program and then change that number later in the program. Some variables might not change such as when the application does not need the variable to change but often the contents of variables do change.

A variable is like a box in memory that holds a data value. The two characteristics of variables are

- Every variable has a name.
- Every variable can hold only one kind of data.

The following sections explain how you can request variables from Visual Basic so that you have places to put data values that you need to store.

## Defining Variables

---

*Definition:* To *define* a variable means to create and name a variable.

---

A program can have as many variables as you need it to have. Before you can use a variable, you must request that Visual Basic create the variable by defining the variable first. When you define a variable, you tell Visual Basic these two things:

- The name of the variable
- The data type of the variable

Once you define a variable, that variable always retains its original data type. Therefore, a single-precision variable can hold only single-precision values. If you stored an integer in a

single-precision variable, Visual Basic would convert the integer to a single-precision number before it stored the number in the variable. Such conversions are common, and they typically do not cause many problems.

The Dim statement defines variables. Using Dim, you tell Visual Basic

- that you need a variable
- what to name the variable
- what kind of variable you want

Dim short for *dimension* is a Visual Basic statement that you write in an application s Code window. Whenever you learn a new statement, you need to learn the format for that statement. Here is the format of the Dim statement:

```
Dim VarName AS DataType
```

*VarName* is a name that you supply. When Visual Basic executes the Dim statement at runtime, it creates a variable in memory and assigns it the name you give in the *VarName* location of the statement. *DataType* is one of the seven Visual Basic data types that you learned about in Table 7.1.

---

**Tip:** Think of a variable as an internal label control that the user cannot see. The variable has a name and holds data just like the label control has a Name property and holds a caption. Variables are more specific than controls, however, because of their data type requirements.

---

Variable names must follow the same naming rules as controls. In Unit 4, you learned the naming conventions for a control s Name property for example, the name must be from 1 to 40 characters long, cannot be a reserved word, and so on.

Always use a Dim statement to define variables before you use variables. If the Options Environment Require Variable Definition option is set to Yes as it is by default Visual Basic will issue an error message whenever you use a variable that you have not defined. This option prevents you from inadvertently misspelling variable names and helps you avoid errors in your program.

Suppose that you give a partial program to another Visual Basic programmer to work on. If you want to require that all variables be defined but are unsure of the other programmer's Options Environment Require Variable Definition setting, select (general) from the Code window's Object dropdown list and add the following special statement:

```
Option Explicit
```

No matter how the Options Environment setting is set, the programmer cannot slip variables into your program without defining them properly in Dim statements. Again, requiring variable definitions helps

eliminate bugs down the road, so you are wise to get in the habit of putting Option Explicit in the (general) section of every program s Code window.

The following statement defines a variable named ProductTotal:

```
Dim ProductTotal As Currency
```

From the Dim statement, you know that the variable holds currency data and that its name is ProductTotal. All Visual Basic commands and built-in routines require an initial capital letter. Although you don't have to name your variables with an initial capital letter, most Visual Basic programmers do for the sake of consistency. Additional caps help distinguish individual words inside a name. (Remember that you cannot use spaces in the name of variable.)

---

**Note:** The Static and Global statements also define variables, as you will see in Lessons 6 and 8.

---

The following statements define integer, single-precision, and double-precision variables:

```
Dim Length As Integer

Dim Price As Single

Dim Structure As Double
```

---

**Warning:** Except for rare occasions that you will learn about in Lesson 8, "Modular Programming," never define two variables with the same name. Visual Basic won t know which one you are referring to when you try to use one of them.

---

---

*Definition:* A *variable-length string* holds strings of any length.

---

If you want to write a program that stores the user's text box entry for the first name, you would define a string like this:

```
Dim
FirstName As String
```

You can get fancy when you define strings. This FirstName string can hold any string from 0 to 65,500 characters long. You will learn in the next section how to store data in a string. FirstName can hold data of virtually any size. You could store a small string in FirstName such as "Joe" and then store a longer string in FirstName such as "Mercedes". FirstName is a variable-length string.

---

*Definition:* A *fixed-length string* holds strings of a preset maximum fixed size.

---

Sometimes you want to limit the amount of text that a string holds. For example, you might need to define a string variable to hold a name that you read from the disk file. Later, you will display the contents of the string in a label on the form. The form's label has a fixed length, however assuming that the AutoSize property is set to True. Therefore, you want to keep the string variable to a reasonable length. The following Dim statement demonstrates how you can add the * *StringLength* option when you want to define fixed-length strings:

```
Dim Title As String * 20
```

Title is the name of a string variable that can hold a string from 0 to 20 characters long. If the program attempts to store a string value that is longer than 20 characters in Title, Visual Basic truncates the string and stores only the first 20 characters.

Here's a shortcut: You can omit the As Variant descriptor when you define variant variables. This Dim statement:

```
Dim Value As Variant
```

does exactly the same thing as

```
Dim Value
```

A good rule of thumb is to make your code as explicit as possible. In other words, As Variant requires extra typing at the keyboard, but using the As Variant makes your request for a variant variable clearer. When you later maintain the program, you will know that you *meant* to define a variant variable and that you did not accidentally omit a different data type.

As you will see throughout this book, you use variables to hold intermediate mathematical results, as well as data values typed by the user in controls such as text boxes. Variables work well with the controls on the form. So far, you have learned how to define variables but not store data in them. The next section

explains how to store and retrieve values from the variables that you define.

---

**Warning:** If you begin calling a variable one name, you must stay with that name for the entire program. Sale is not the same variable name as Sales. If you define a variable using a name and later change only the case of certain letters, Visual Basic will convert the original case to the new case. In other words, if you define a variable with the name Profit and use that name for a while in the program, but later in the program you call the variable PROFIT, Visual Basic changes all previous occurrences of Profit to PROFIT.

---

Visual Basic supports a shortcut when you need to define several variables. Instead of listing each variable definition on separate lines like this:

```
Dim A As Integer

Dim B As Double

Dim C As Integer

Dim D As String

Dim E As String
```

you can combine the variables of the same data type on one line. For example,

```
Dim A, C As Integer

Dim B As Double

Dim D, E As String
```

Some programmers even code all variable definitions on the same line. As you can see, though, too many variable definitions makes Dim hard to manage. For example,

```
Dim A, C As Integer, B As Double, Dim D, E As String
```

---

Reducing Dim with Variable Suffix Characters: There is a set of suffix characters for variable names that you can use to specify a variable's data type without having to define the variable first. Table 7.2 listed the suffix characters for constant values. Variables use the more complete version shown here:

| Suffix | Data Type | Example |
|---|---|---|
| % | integer | Age% |
| & | long | Amount& |
| ! | single | Distance! |
| # | double | KelvinTemp# |
| @ | currency | Pay@ |
| $ | string | LastName$ |

The statement Option Explicit cannot appear in the (general) section of the Code window because the suffix characters are not enough to define variables.
The variable LastName$ is a string variable, and Visual Basic knows that the variable is a string variable even if no Dim statement defines the variable as a string. You won t use the variable suffixes much because you will write clearer programs if you use Dim. Nevertheless, you might run into code written by others that uses the suffix characters on variables, and you should be aware of what the suffix characters mean.

---

## Assigning Values to Variables

---

*Definition:* A *null string* is a zero-length empty string that is often represented as "".

---

When you first define variables, Visual Basic stores zeroes in the numeric variables and null strings in the string variables. Use the *assignment statement* when you want to put other data values into variables. Variables hold data of specific data types, and many lines inside a Visual Basic program's Code window consist of assignment statements that assign data to variables. Here is the format of the assignment statement:

```
[Let]VarName = Expression
```

The Let command name is optional; it is rarely used these days. The *VarName* is a variable name that you have defined using the Dim statement. *Expression* can be a constant, another variable, or a

mathematical expression.

Suppose that you need to store a minimum age value of 18 in an integer variable named MinAge. The following assignment statements do that:

```
Let MinAge = 18
```

and

```
MinAge = 18
```

---

**Note:** The rest of this book omits Let from assignment statements because the word is superfluous.

---

To store a temperature in a single-precision variable named TodayTemp, you could do this:

```
TodayTemp = 42.1
```

The data type of *Expression* must match the data type of the variable to which you are assigning. In other words, the following statement is invalid. It would produce an error in Visual Basic programs if you tried to use it.

```
TodayTemp = "Forty-Two point One"
```

If TodayTemp were a single-precision variable, you could not assign a string to it. However, Visual Basic often makes a quick conversion for you when the conversion is trivial. For example, it is possible to perform the following assignment even if you have defined Measure to be a double-precision variable:

```
measurement
= 921.23
```

At first glance, it appears that 921.23 is a single-precision number because of its size. 921.23 is actually a variant data value. Recall that Visual Basic assumes all data constants are variant unless you explicitly add a suffix character to the constant to make the constant a different data type. Visual Basic can easily and safely convert the variant value to double-precision. That's just what Visual Basic does, so the assignment works fine.

In addition to constants, you can assign other variables to variables. Consider the following code:

```
Dim Sales As Single, NewSales As Single

Sales = 3945.42

NewSales = Sales
```

When the third statement finishes, both Sales and NewSales have the value 3945.42.

Feel free to assign variables to controls and controls to variables. Suppose, for example, that the user types the value 18.34 in a text box's Text property. If the text box's Name property is txtFactor, the following statement stores the value of the text box in a variable named FactorVal:

```
FactorVal = txtFactor.Text
```

Suppose that you defined Title to be a string variable with a fixed length of 10, but a user types Mondays Always Feel Blue in a text box's Text property that you want to assign to Title. Visual Basic stores only the first ten characters of the control to Title and truncates the rest of the title. Therefore, Title holds only the string "Mondays Al".

***Stop & Type:*** This is the first of several program code reviews that you will find throughout the rest of the book. Listing 7.1 contains a short event procedure that assigns a new command button caption.

***Review:*** You can instantly make data appear on the form by assigning the Text property of text boxes or the Caption property of labels and command buttons. Suppose you put a command button named cmdJoke on a form. The event procedure shown in Listing 7.1 changes the command button's caption when the user clicks the command button.

**Listing 7.1. An event procedure that assigns a new command button caption.**

```
1: Sub cmdJoke_Click ()

2: cmdJoke.Caption = "Bird Dogs Fly"

3: End Sub
```

---

**Warning:** Throughout this book, you will often see code listings with numbers down the left side. The numbers are used to refer to individual lines. Don't actually type the numbers or the colons that follow them when you enter the program code.

***Analysis:*** No matter what the command button's Caption property is set to at the start of the program, when the user clicks the command button, this event procedure executes and the command button's caption changes to Bird Dogs Fly (line 2).

---

**Tip:** You will see a more comprehensive program that assigns data values to and from form controls in this lesson's project.

---

Provided that you have added the CONSTANT.TXT file to AUTOLOAD.MAK, you can assign the named constants in CONSTANT.TXT to controls inside event procedures. The following assignment statement adds a single line border around the label named lblSinger:

```
lblSinger.BorderStyle = FIXED_SINGLE
```

Of course, you must know the names of the named constants inside CONSTANT.TXT before you can assign them. Throughout this book, I will point out when you can use named constants for setting properties.

---

**Note:** When you are not using CONSTANT.TXT, assign only the number when a control's property can accept a limited range of values. For example, the possible values that you can select for a label's BorderStyle in the Properties window are 0-None and 1-Fixed Single. To assign either border style directly without using a named constant, assign just 0 or 1. Don't spell out the entire property. For example,
lblSinger.BorderStyle = 1

---

***Review:*** Using the assignment statement, you can assign constants, named constants, control values, and variables to other variables and controls. The assignment statement requires that your variables first be defined with Dim if your program's (general) procedure contains an Option Explicit 1 statement, as I have suggested.

## Mathematical Expressions

***Concept:*** Data values and controls are not the only kinds of assignments that you can make. With the Visual Basic math operators, you can calculate and assign expression results to variables when you code assignment statements that contain expressions.

---

*Definition:* An *operator* is a word or symbol that does math and data manipulation.

It is easy to make Visual Basic do your math. Table 7.3 describes Visual Basic's primary math operators. There are other operators, but the ones in Table 7.3 will suffice for most of the programs that you write. Look over the operators. You are already familiar with most of them because they look and act just like their real-world counterparts.

**Table 7.3. The primary math operators.**

| *Operator* | *Example* | *Description* |
|---|---|---|
| + | Net + Disc | Adds two values |
| - | Price - 4.00 | Subtracts one value from another value |
| * | Total * Fact | Multiplies two values |
| / | Tax / Adjust | Divides one value by another value |
| ^ | Adjust ^ 3 | Raises a value to a power |
| & or + | Name1 & Name2 | Concatenates two strings |

Suppose that you wanted to store the difference between the annual sales (stored in a variable named AnnualSales) and cost of sales (stored in a variable named CostOfSales) in a variable named NetSales. Assuming that all three variables have been defined and initialized, the following assignment statement computes the correct value for NetSales:

```
NetSales = AnnualSales - CostOfSales
```

This assignment tells Visual Basic to compute the value of the expression and to store the result in the variable named NetSales. Of course, you can store the results of this expression in Caption or Text properties, too.

If you want to raise a value by a power which means to multiply the value by itself a certain number of times you can do so. The following code assigns 10000 to Value because ten raised to the fourth power 10x10x10x10 is 10,000:

```
Years = 4

Value = 10 ^ Years
```

No matter how complex the expression is, Visual Basic computes the entire result before it stores that result in the variable at the left of the equals sign. The following assignment statement, for example, is rather lengthy, but Visual Basic computes the result and stores the value in the variable named Ans:

```
Ans = 8 * Factor - Pi + 12 * MonthlyAmts
```

Combining expressions often produces unintended results because Visual Basic computes mathematical results in a predetermined order. Visual Basic always calculates exponentiation first if one or more ^ operators appear in the expression. Visual Basic then computes all multiplication and division before any addition and subtraction.

Visual Basic assigns 13 to Result in the following assignment:

```
Result = 3 + 5 * 2
```

At first, you might think that Visual Basic would assign 16 to Result because 3 + 5 is 8 and 8 * 2 is 16. However, the rules state that Visual Basic always computes multiplication and division if division exists in the expression before addition. Therefore, Visual Basic first computes the value of 5 * 2, or 10, and next adds 3 to 10 to get 13. Only then does it assign the 13 to Result.

If both multiplication and division appear in the same expression, Visual Basic calculates the intermediate results from left to right. For example, Visual Basic assigns 20 to the following expression:

```
Result = 8
/ 2 + 4 + 3 * 4
```

Figure 7.1 shows how Visual Basic computes this expression. Visual Basic computes the division first because the division appears to the left of the multiplication. If the multiplication appeared to the left of the division, Visual Basic would have multiplied first. After Visual Basic calculates the intermediate answers for the division and the multiplication, it performs the addition and stores the final answer of 20 in Result.

**Figure 7.1. Visual Basic computes expressions in a predetermined order.**

---

**Note:** The order of computation has many names. Programmers usually use one of these terms: o*rder of operators*, *operator precedence*, or *math hierarchy*.

---

It is possible to override the operator precedence by using parentheses. Visual Basic always computes the values inside any pair of parentheses before anything else in the expression, even if it means ignoring operator precedence. The following assignment statement stores 16 in Result because the parentheses force Visual Basic to compute the addition before the multiplication:

```
Result = (3 + 5) * 2
```

---

**Tip:** Appendix B contains the complete Visual Basic operator precedence table. The table contains several operators that you have yet to learn about, so you might not understand the full table at this time.

---

The following expression stores the fifth root of 125 in the variable named root5:

```
root5 = 125 ^ (1/5)
```

As you can see from this expression, Visual Basic supports fractional exponents.

---

*Definition: Concatenation* means the joining together of two or more strings.

---

One of Visual Basic's primary operators has nothing to do with math. The concatenation operator joins one string to the end of another. Suppose that the user entered his first name in a label control named lblFirst and his last name in a label control named lblLast. The following concatenation expression stores the full name in the string variable named FullName:

```
FullName = lblFirst & lblLast
```

There is a problem here, though, that might not be readily apparent there is no space between the two names. The & operator does not automatically insert a space because you don t always want spaces inserted when you concatenate two strings. Therefore, you might have to concatenate a third string between the other two, as in

```
FullName = lblFirst & " " & lblLast
```

Visual Basic actually supports a synonym operator, the plus sign, for concatenation. In other words, the following assignment statement is identical to the previous one:

```
FullName = lblFirst + " " + lblLast
```

Even Microsoft, the maker of Visual Basic, recommends that you use & for string concatenation and reserve + for mathematical numeric additions. Using the same operator for two different kinds of operations can lead to confusion.

***Review:*** The math operators enable you to perform all kinds of calculations and assignments. Visual Basic means never having to use a calculator again! When you use expressions, however, remember operator precedence. When in doubt, use parentheses to indicate exactly which operations in an

expression you want Visual Basic to calculate first.

## The *Val()* Function

***Concept:*** Visual Basic treats all data stored in controls on the form as variant values. There is a quirk in Visual Basic that makes Visual Basic think these controls are strings when you want to add two control values together. The Val() function makes sure that Visual Basic treats control values as numbers when you need numbers for expressions.

Figure 7.2 shows a simple form with two text boxes and a command button. As soon as the user types last year's sales into a text box named txtLast and this year's sales into a text box named txtThis, the command button triggers a Click event procedure that does the following:

```
lblAvg = (txtLast.Text +
txtThis.Text) / 2
```

**Figure 7.2. Computes the average sales value for two years.**

Despite the fact that everything is correct about this expression, Visual Basic does not store the average of the two text boxes in the lblAvg label. Visual Basic incorrectly thinks that you want to concatenate the two text box values instead of adding them.

---

**Note:** See the problems that can occur when a language uses the same operator, +, for two different purposes!

---

Instead of using this relatively simple expression, you must add a little mess by using the Val() function. The word Val() looks like the name of an event procedure because of the parentheses. However, Visual Basic has several built-in routines called *functions.* Function names end with parentheses. You will learn all about Visual Basic functions in Lesson 7. For now, just use the functions they're easy and don t worry about how they work.

Here's how you use Val(): Put a string or something that Visual Basic might think is a string, such as a control inside the parentheses. As long as the parenthetical value can be interpreted as a number, even if the value is stored as a string, Visual Basic converts the value to a number temporarily so that you can calculate a mathematical result. The previous assignment requires that you use Val() twice. For example, instead of

```
lblAvg = (txtLast.Text + txtThis.Text) / 2
```

you must code

```
lblAvg =
(Val(txtLast.Text) + Val(txtThis.Text)) / 2
```

Val() ensures that Visual Basic treats the text boxes as numeric quantities and not as strings.

By the way, if the user does not enter numbers in both text boxes, Visual Basic uses zero for the converted number if no numeric digits appear in the text boxes. If any number or set of digits appears in the text boxes, Val() uses those digits for the converted number, often producing funny results. For example, if txtLast.Text contains 18 years old (why the user would type *that* for a sales figure is beyond me!), Val() strips off the 18 and uses 18 for the numeric value in the average calculation.

---

**Note:** You can find this simple sales averaging program under the name SALAVG.MAK on the disk that comes with this book. The program is extremely simple. It does not format the average to dollars and cents, so you might have more or fewer than two decimal places. Nevertheless, you can load and study the program's cmdAvg_Click() procedure to get a feel for the calculation being performed and for how Val() helps the calculation.

---

***Review:*** When you calculate using the form's data, you might have to convert control values to numbers temporarily by using Val() before you use the values in numeric expressions. The controls retain their variant data types except for the locations where Val() converts the label data types to numbers.

## Homework

### General Knowledge

1. What is a variable?
2. What statement defines variables?
3. What is a data type?
4. Name the seven Visual Basic data types.
5. Which data type holds the smallest range of values?
6. Which data type holds the largest range of values?
7. Which data types always have decimal places?
8. What does the E mean when it appears inside numbers?
9. What does scientific notation do?
10. True or false: Constants can change throughout a program.
11. True or false: Variables can stay the same throughout a program.

12. Why is it helpful to define all variables before you use them?
13. What is the difference between a variable-length string and a fixed-length string?
14. True or false: You can store control property values in variables.
15. True or false: You can store variables in control property values.
16. Given the Visual Basic order of precedence, what value is assigned to the left-hand variables in each of the following assignment statements?
    . ResultA = 1 + 2 * 3
    B. ResultB = (1 + 2) * 3
    C. ResultC = 2 ^ 4
    D. Average = 10 + 20 + 30 / 3
    E. Num = 10 - 3 * 2 + 4 / 2
    F. Ans = 10 * 2 ^ 2
17. What does it mean to concatenate two strings?
18. What is the difference between + and &?

## Write Code That...

1. Define a variable that holds the exact square footage measurement of your home. Use a variable that can accept a decimal point, but don't exaggerate and define a variable that is much larger than you really need.
2. Rewrite the following assignment statements so that they are shorter:
   Let SalesPrice = Price / Discount
   Let Tax = TaxRate * SalesPrice

## Find the Bug

1. What is wrong with the following variable definition?
   Dim MyAge As Integer
   Dim Height As Single
   Dim Weight As Double
   Dim Name As String
2. Judy is having problems with her Visual Basic program. She declared three variables that she wants to use for three totals:
   Dim DivTotal As Single
   Dim DivTotal As Single

Dim DivTotal As Single
Judy's goal is to store the inventory totals for each of her company's divisions in the three variables. Visual Basic does not like what she is doing. Help Judy out because her boss is getting snippy.

3. Larry Derryberry gets an error when he assigns the following currency variable the value. Tell Larry what he is doing wrong.
Salary = $47,533.90

## Extra Credit

Add a third label to the SALAVG.MAK averaging program shown in Figure 7.2 so that the program averages three sales values.

# Visual Basic in 12 Easy Lessons



# Lesson 4, Unit 8

## Data Comparisons

### What You'll Learn

- [lb] Relational operators
- [lb] If statements
- [lb] Handling false conditions
- [lb] Logical operators
- [lb] Multiple choice with Select Case
- [lb] Two additional Select Case formats

Computers cannot think on their own, but with your help they can be taught to make decisions based on values contained in controls and variables. Visual Basic s decision-making capability enables it to calculate sales figures based on certain conditions, to print exception reports, and to check user responses

by means of the form s controls.

You have learned only a few statements so far:

- [lb] The End statement, which terminates programs
- [lb] The Dim statement, which defines program variables
- [lb] The assignment statement, which stores data values in controls and variables.

In addition, you know about the fundamental math operators. In this unit, you will learn some new programming statements and operators that you can use along with the ones you already know to write programs that make data-based decisions.

## The Relational Operators

***Concept:*** Visual Basic supports the use of six operators that produce true or false results based on data values. Once you learn the relational operators, you can combine them operators with the If statement to add power to your programs.

---

*Definition: Relational operators* compare data values to one another.

---

Table 8.1 describes the six relational operators that Visual Basic supports. You use the relational operators to compare data values. They are easy to use. If you take any two numbers, one number is always be greater than, equal to, or less than the other.

---

**Note:** The mathematical operators that you learned in the previous unit produced numeric answers. The relational operators produce only true and false answers. In other words, one data value is either more than another a true result or the data value is *not* more than the other a false result.

---

**Table 8.1. The relational operators.**

| *Operator* | *Usage* | *Description* |
|---|---|---|
| > | Sales > Goal | The *greater than* operator. Returns true if the value on the left side of > is numerically or alphabetically greater than the value on the right. Otherwise, false. |
| < | Pay < 2000.00 | The *less than* operator. Returns true if the value on the left side of < is numerically or alphabetically less than the value on the right. Otherwise, false. |

| | | |
|---|---|---|
| = | Age = Limit | The *equal to* operator (sometimes called the *equal* operator). Returns true if the values on both sides of = are equal to each other. Otherwise, false. |
| >= | FirstName >= "Mike" | The *greater than or equal to* operator. Returns true if the value on the left side of >= is numerically or alphabetically greater than or equal to the value on the right. Otherwise, false. |
| <= | Num <= lblAmt.Caption | The *less than or equal to* operator. Returns true if the value on the left side of <= is numerically or alphabetically less than or equal to the value on the right. Otherwise, false. |
| <> | txtAns.Text <> "Yes" | The *not equal to* operator. Returns true if the value on the left side of <> is numerically or alphabetically unequal to the value on the right. Otherwise, false. |

---

*Definition:* The *ASCII table* is a list of characters with corresponding numeric representations.

---

All the relational operators work on both numeric and alphabetic values. You can compare any kind of number against another number, or any kind of string against another string. When you compare strings, Visual Basic uses the ASCII table, included in Appendix A, to determine how to compare the characters. For example, the ASCII table says that the uppercase letter *A* whose ASCII numeric value is 65 is less than the uppercase letter *B* whose ASCII numeric value is 66. Notice that all uppercase letters are less than lowercase letters. Therefore, the abbreviation *ST* is less than *St*.

---

**Tip:** Pronounce *ASCII* as *ask-ee*.

---

To understand how relational operators work, you must understand how to use their true or false results. The If statement, introduced in the next section, explains how you can take use true and false results to make decisions in your program. Before you read the next section, make sure that you understand how these operators compare values. For a quick self-test, make sure that you understand the *Result* column of Table 8.2 before you go any further.

**Table 8.2. Relational results.**

| *Relation* | *Result* |
|---|---|
| 10 > 5 | True |
| 5 > 10 | False |
| 5 < 10 | True |
| "Apple" <= "Orange" | True |
| "Mac Donald" < "Mc Donald" | True |
| 0 >= 0 | True |

| | |
|---|---|
| 0 <= 0 | True |
| 1 <> 2 | True |
| 2 >= 3 | False |

Keep Each Side a Consistent Data Type: The expressions on both sides of a relational operator must have the same data type or at least compatible data types. In other words, you cannot compare a string to a numeric data type. If you try, you will get a Type mismatch error because the data types don't match. You can compare any numeric data type against any other numeric data type most of the time. In other words, you can test whether a single-precision value is less than or greater than an integer value. Be careful, however, when you compare non-integer numeric data for equality. Precision numbers are difficult to represent internally. For example, if you assigned 8.3221 to a single-precision variable and assigned 8.3221 to another single-precision variable, Visual Basic might return a false result if you compare the values for equality. Internally, one of the variables might actually hold 8.322100001 because of rounding errors that occur in normally insignificant decimal places. You can safely compare two currency values for equality, however, because Visual Basic maintains their accuracy to two decimal places.

**Note:** The relational operators are sometimes called the *conditional operators* because they test conditions that are either true or false.

***Review:*** The relational operators compare values against one another. You can compare for equality, inequality, and size differences. The relational operators work for both string data and numeric data. By themselves, the relational operators would not be worth much. However, you can use them to compare data by using the If statement, which you learn about in the next section.

## The *If* Makes Decisions

***Concept:*** The If statement uses the relational operators to test data values. It perform one of two possible code actions, depending on the result of the test. In the previous unit, you saw how Visual Basic executes the Dim and assignment statements in the order in which you type them in the program. With If statements, Visual Basic tests whether to execute blocks of code. In other words, an If statement uses the relational operators to test data and *might* execute one or more lines of subsequent code, depending on the results of the test.

The If statement makes decisions. If a relational test is true, the body of the If statement executes. In fact, the previous sentence is almost identical to Visual Basic's If statement. Here is one format of If:

```
If relationalTest Then

 One or
more Visual Basic statements

End If
```

The End If statement informs Visual Basic where the body of the If statement ends. Suppose that the user enters a sales figure into a text box control named txtSales. The following If statement computes a bonus amount based on the sales:

```
If (txtSales.Text > 5000.00) Then

 Bonus = Val(txtSales.Text) * .12

End If
```

Remember that Visual Basic stores zero in all variables that you don't first initialize. Therefore, Bonus has a zero before the If statement executes. Once the If executes, the code changes the Bonus variable only if the value of the txtSales.Text property is more than 5000.00. The Val() function converts the text box's variant data to a numeric value for the computation. In a way, the If reads like this:

*If the sales are more than $5,000.00, then compute a bonus based on that sales value.*

The body of an If can have more than one statement. The following If calculates a bonus, the cost of sales, and a reorder amount based on the value of the txtSales text box entry:

```
If (txtSales.Text > 5000.00) Then

 Bonus = Val(txtSales.Text) * .12

 CostOfSales = Val(txtSales.Text) *
.41

 ReorderCost = Val(txtSales.Text) * .24

End If
```

The three statements that make up the body of the If execute only if the condition txtSales.Text >

5000.00 is true. Suppose that this code contains another assignment statement immediately after End If. That assignment statement is outside the body of the If,so the true or false result of the condition affects only the body of the If. Therefore, the tax computation in the following routine executes regardless of whether the sales are more than or less than $5,000.00:

```
If (txtSales.Text > 5000.00) Then

 Bonus = Val(txtSales.Text) * .12

 CostOfSales = Val(txtSales.Text) * .41

 ReorderCost = Val(txtSales.Text) * .24

End If

Tax = .12 *
Val(txtSales.Text)
```

---

**Tip:** The parentheses are not required around the relational test in an If statement, but they help separate the test from the rest of the code.

---

Can you see how the program makes decisions using If? The body of the If executes only if the relational test is true. Otherwise, the rest of the program continues as usual.

There is a shortcut form of If that you might run across. The *single-line If statement* has a format that looks like this:

```
If relationalTest Then VBStatement
```

The single-line If does not require an End If statement because relational test and the body of the If reside on the same line. Single-line If statements do not provide for easy program maintenance. If you decide that you want to add to the body of the If, you must convert the single-line If to a multiple-line If,and you might forget to then add End If. Therefore, even if the body of an If statement takes only one line, code the If as a multiple-line If-End If statement.

***Review:*** The If statement determines whether code executes. The If checks the true or false condition of the relational test. If the data relationally tests true, Visual Basic executes the body of the If. If the data relationally tests false, Visual Basic skips over the body of the If statement. No matter what happens, the

code that follows the End If statement executes as usual.

## Handling False Conditions

***Concept:*** Whereas If executes code based on the relational test's true condition, the Else statement executes code based on the relational test's false condition. Else is actually part of the If statement. This section explains the full If-Else statement. It shows you how you can execute one section of code or another, depending on the relational test.

The Else statement, part of an extended If statement, specifies the code that executes if the relational test is false. Here is the complete format of the If statement with Else:

```
If relationalTest Then

 One or more Visual Basic statements

Else

 One or more Visual Basic
statements

End If
```

Typically, programmers call this full-blown If statement the If-Else statement. The If-Else statement is sometimes called a *mutually exclusive* statement. The term *mutually exclusive* simply means that one set or of code or the other executes, but not both. The If-Else statement contains two sets of code that is, two bodies of one or more Visual Basic statements and only one set executes, depending on the result of the If. An If statement is either true or false. Therefore, either the first or the second body of code in an If-Else executes.

***Stop & Type:*** Suppose that a salesman receives a bonus if sales are high (over $5,000.00) or suffers a pay cut if sales are low (below $5,000.00). The If-Else in Listing 8.1 contains the code necessary to reward or punish the salesman. The code body of the If computes the bonus as done in the previous section. The code body of the Else subtracts $25 from the saleman s pay, which is stored in the variable named PayAmt, if the sales quota is not met.

Listing 8.1 contains a *code fragment* sometimes called a *code snippet* because the listing does not show variable definitions or any previous code that initialized PayAmt with the salesman s pay. Listing 8.1 contains only the If-Else code needed to reward or punish the salesman s effort.

***Review:*** The If handles the true result of a conditional test and the Else handles the false result. By using an If-Else, you can increase the power of Visual Basic s data-testing and decision-making capabilities.

**Listing 8.1. Determining a sales bonus or penalty.**

```
1: If (Val(txtSales.Text) > 5000.00) Then

2: Bonus = .05 * Val(txtSales.Text)

3: Else

4: PayAmt = PayAmt - 25.00

5: End If

6: Taxes = PayAmt * .42
```

***Analysis:*** Notice that line 6 computes a tax amount. No matter what the outcome of the If-Else s relational test is, line 6 always executes. Line 6 is not part of either the If body or the Else body of code.

Line 1 tests whether the salesman's sales value stored in the text box named txtSales is more than $5,000.00. If the relational test is true, line 2 computes a bonus. If the test is false, Else executes (line 4).

As Figure 8.1 illustrates, either the If's body of code executes or the Else's body of code executes, but never both. Visual Basic decides in line 1 which assignment to make.

**Figure 8.1. Either the If body or the Else body executes, but not both.**

Line 4 introduces an assignment statement that might surprise you at first. Line 4 appears to make the statement that the pay is equal to the pay minus 25. You know that *nothing* can be equal to itself minus 25. In math, the equal sign acts as a balance for the two sides of the equation. In Visual Basic, however, when the equal sign is not used inside an If's relational test, it is an assignment it that takes everything to the right of the equal sign and stores that value in the variable to the left of the equal sign. Line 4 first subtracts the 25 from PayAmt and then assigns that result back to PayAmt. In effect, it lowers the value of PayAmt by 25.

---

**Note:** When a variable appears on both sides of an assignment's equal sign, the variable is being *updated* in some way.

---

# Logical Operators

***Concept:*** Three additional operators, And, Or, and Not, look more like commands than operators. And, Or, and Not are called *logical operators*. Logical operators enable you to add to the power of relational operators by extending the tests that your If statements can make. They enable you to combine two or more relational tests.

Table 8.3 describes the logical operators, which work just like their spoken counterparts.

**Table 8.3. The logical operators.**

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

As you can see from Table 8.3, the And and Or logical operators enable you to combine more than one relational test in a single If statement. The Not negates a relational test. You can often turn a Not condition around. Not can produce difficult relational tests, and you should use it cautiously. The last If in Table 8.3, for instance, could easily be changed to If (Ans <> "Yes") to eliminate the Not.

Your code often must perform an assignment, print a message, or display a label if two or more conditions are true. The logical operators make the combined condition easy to code. Suppose that you want to reward the salesman if sales total more than $5,000 and if the he sells more than 10,000 units of a particular product. Without And, you have to embed an If statement in the body of another If statement. For example,

```
If (Sales > 5000.00) Then

 If (UnitsSold > 10000) Then

 Bonus = 50.00

 End If
```

```
End If
```

Here is the same code rewritten as single If. It is easier to read and to change later if you need to update the program.

```
If (Sales > 5000.00) And (UnitsSold > 10000) Then

 Bonus = 50.00

End If
```

How can you rewrite this If to pay the bonus if the salesperson sells *either* more than $5,000 in sales *or* if he sells more than 10,000 units? Here is the code:

```
If (Sales > 5000.00) Or (UnitsSold > 10000) Then

Bonus = 50.00

End If
```

***Stop and Type:*** Listing 8.2 contains an If-Else that tests data from two divisions of a company and calculates values from the data.

***Review:*** The logical operators enable you to combine two or more conditional tests. Without logical operators, you must code a longer series of nested If statements.

**Listing 8.2. Calculating sales figures for a company s divisions.**

```
1: If (DivNum = 3) Or (DivNum = 4) Then

2: DivTotal = DivSales3 + DivSales4

3: GrandDivCosts = (DivCost3 * 1.2) + (DivCost4 * 1.4)

4: Else

5:
DovTotal = DivSales1 + DivSales2
```

```
6: GrandDivCosts = (DivCost1 * 1.1) + (DivCost5 * 1.9)

7: End If
```

***Analysis:*** Assume that the users of the code in Listing 8.2 own a company with four divisions. The east coast divisions are numbered 3 and 4, and the west coast divisions are numbered 1 and 2. Listing 8.2 calculates aggregate sales and costs totals for one of the coasts, depending on the DivNum value. You must assume that all the variables have been defined and initialized with proper values.

If DivNum contains either a 3 or a 4, the user is requesting figures for the east coast, and the code in lines 2 3 executes to produce an east coast pair of values. If DivNum does not contain a 3 or a 4, the program assumes that DivNum contains a 1 or a 2,and the west coast pair of values is calculated in lines 5 6.

## Multiple Choice with *Select Case*

***Concept:*** The If statement is great for data comparisons in cases where one or two relational tests must be made. When you must test against more than two conditions, however, the If becomes difficult to maintain. The logical operators help in only certain kinds of conditions. At other times, you must nest several If-Else statements inside one other.

Consider the If statement shown in Listing 8.3. Although the logic of the If statement is simple, the coding is extremely difficult to follow.

**Listing 8.3. Nested If-Else statements get complex quickly.**

```
If (Age = 5) Then

 lblTitle.Text = "Kindergarten"

Else

 If (Age = 6) Then

 lblTitle.Text = "1st Grade"

 Else

 If (Age 7) Then
```

```
 lblTitle.Text = "2nd Grade"

 Else

If (Age = 8) Then

 lblTitle.Text = "3rd Grade"

 Else

 If (Age = 9) Then

 lblTitle.Text = "4th Grade"

 Else

 If (Age = 10) Then

 lblTitle.Text = "5th Grade"

 Else

 If (Age = 11) Then

 lblTitle.Text = "6th
Grade"

 Else

 lblTitle.Text = "Advanced"

 End If

 End If

 End If
```

```
 End If

 End If

 End If

End If
```

Visual Basic supports a statement, called Select Case, that handles such multiple-choice conditions better than If-Else. Here is the format of the Select Case statement:

```
Select Case Expression

Case value

 One or more Visual Basic statements

Case value

 One or more Visual Basic statements


[Case value

 One or more Visual Basic statements]

[Case Else:

 One or more Visual Basic statements]

End Select
```

The format of Select Case makes the statement look as difficult as a complex nested If-Else, but you will soon see that Select Case statements are actually easier to code and to maintain than their If-Else counterparts.

*Expression* can be any Visual Basic expression such as a calculation, a string value, or a numeric value

provided that it results in an integer or string value. *value*s must be integer or string values that match *Expression* s data type.

The Select Case statement is useful when you must make several choices based on data values. Select Case can have two or more Case *value* sections. The code that executes depends on which *value* matches *Expression*. If none of the *value*s match *Expression*, the Case Else body of code executes if you code the Case Else. Otherwise, nothing happens and control continues with the statement that follows End Select.

---

**Warning:** Don't use Select Case when a simply If or a simple If-Else will suffice. Test logic is often so straightforward that a Select Case would be overkill and even less clear than an If. Unless you need to compare against more than a couple of values, stick with the If and If-Else statements because of their simplicity.

---

***Stop and Type:*** The fastest way to learn Select Case is to see an example of it. Listing 8.4 contains a Select Case version of the child grade assignments shown in Listing 8.3. Select Case organizes the multiple-choice selections into a more manageable format.

***Review:*** The Select Case statement is a good substitute for long, nested If-Else conditions when one of several choices are possible. You set up your Visual Basic program to execute one set of Visual Basic statements from a list of statements inside Select Case.

**Listing 8.4. Using Select Case to simplify complex nested If-Else statements.**

```
1: Select Case Age

2: Case 5: lblTitle.Text = "Kindergarten"

3: Case 6: lblTitle.Text =
"1st Grade"

4: Case 7: lblTitle.Text = "2nd Grade"

5: Case 8: lblTitle.Text = "3rd Grade"

6: Case 9: lblTitle.Text = "4th Grade"

7: Case 10: lblTitle.Text = "5th Grade"

8: Case 11: lblTitle.Text =
```

```
"6th Grade"

9: Case Else: lblTitle.Text = "Advanced"

10: End Select
```

***Analysis:*** If the Age variable holds the value 5, the label is assigned "Kindergarten" in line 2. If the Age variable holds the value 6, the label is assigned "1st Grade" in line 3. The logic continues through line 9. If Age holds a value that does not fall within the range of 5 through 11, line 9 assigns "Advanced" to the label.

The body of each Case can consist of more than one statement, just as the body of an If or If-Else can consist of more than one statement. Visual Basic executes all the statements for any given Case match until the next Case is reached. Once Visual Basic executes a matching Case value, it skips the remaining Case statements and continues with the code that follows the End Select statement.

---

**Note:** Programmers often trigger the execution of complete procedures, such as event procedures, from within a Case statement. As you will learn in Lesson 8, instead of putting several statements in the body of an If-Else or a Case, you can execute a procedure that contains all the statements that execute when a given condition is true.

---

## Two Additional *Select Case* Formats

***Concept:*** Visual Basic s Select Case is one of the most powerful selection statements in any programming language. Pascal, C, and C++ all popular programming languages each contain statements that act like Visual Basic s Select Case, but Select Case offers two additional powerful formats that enable you to modify the way Case matches are made.

The two additional formats differ only slightly from the standard Select Case that you learned in the previous lesson. They enable you to extend the power of Select Case so that Visual Basic can make Case matches on both relational tests and on ranges of values. Here is the first additional format:

```
Select Case Expression

Case Is relation:

 One or more Visual Basic statements
```

```
Case Is relation:

 One or more Visual Basic statements


[Case Is relation:

 One or more Visual Basic statements]

[Case Else:

 One or more Visual Basic statements]

End Select
```

*relation* can be whatever relational test you want to perform against *Expression* at the top of the Select Case. The standard Select Case statement, discussed in the previous section, compared the *Expression* value against an exact Case match. When you use the relational Is Select Case option, each Case can be matched on a relational test.

Here is the format of the second additional Select Case format:

```
Select Case Expression

Case expr1 To expr2:

 One or more Visual Basic statements

Case expr1
To expr2:

 One or more Visual Basic statements

[Case expr1 To expr2:

 One or more Visual Basic statements]
```

```
[Case Else:

 One or more Visual Basic statements]

End Select
```

The Case lines require a range, such as 4 To 6. The To Select Case option enables you to match against a range instead of a relation or an exact match.

---

**Tip:** You can combine the extended formats of Select Case with the standard Select Case so that two or more kinds of Case formats appear within the same Select Case statement.

---

[ic:Stop&Type]The code in Listing 8.4 contains a minor logic bug. The code handles all Age values over 11, but it does not handle Age values below 5. Therefore, if Age contains a value of 4, Listing 8.4 would assign "Advanced" to the label. However, if you add a relational test for the first Case, as shown in Listing 8.5, the code can handle any Age value.

***Review:*** The additional Select Case options extend the power of Select Case beyond that of any selection statement in other languages. Using the two Case options that you learned in this section, you can code multiple-choice selections based on three kinds of matches:

- [lb] An exact Case match to Select Case's *Expression*
- [lb] A relational Case match to Select Case's *Expression*
- [lb] A range of Case matches to Select Case's Expression

**Listing 8.5. Using Select Case to simplify complex nested If-Else statements.**

```
1: Select Case Age

2: Case Is <5: lblTitle.Text = "Too young"

3: Case 5: lblTitle.Text = "Kindergarten"

4: Case 6: lblTitle.Text =
"1st Grade"

5: Case 7: lblTitle.Text = "2nd Grade"

6: Case 8: lblTitle.Text = "3rd Grade"
```

```
7: Case 9: lblTitle.Text = "4th Grade"

8: Case 10: lblTitle.Text = "5th Grade"

9: Case 11: lblTitle.Text =
"6th Grade"

10: Case Else: lblTitle.Text = "Advanced"

11: End Select
```

***Analysis:*** By testing for a value less than 5 in line 1, Listing 8.5 ensures that both younger ages and older ages are covered by the Case selections.

---

**Tip:** Although the Case Else line is optional, be sure to code one unless you know the exact range of values that can match *Expression*.

---

[ic:Stop&Type]Listing 8.6 contains a similar Select Case problem that uses the To option for some of the Case values. Each Case tests against a range of possible values. The ages fall into categories, and the appropriate titles are updated depending on the category matches.

**Listing 8.6. Using Select Case ranges for categorizing multiple matches.**

```
1: Select Case Age

2: Case Is <5: lblTitle.Text = "Too young"

3: Case 5: lblTitle.Text = "Kindergarten"

4: Case 6 To 11: lblTitle.Text = "Elementary"

5: lblSchool.Text = "Lincoln"

6:
Case 12 To 15: lblTitle.Text = "Intermediate"
```

```
7: lblSchool.Text = "Washington"

8: Case 16 To 18: lblTitle.Text = "High School"

9: lblSchool.Text = "Betsy Ross"

10: Case Else: lblTitle.Text = "College"


11: lblSchool.Text = "University"

12: End Select
```

***Analysis:*** Listing 8.6 contains every Select Case option that exists. In the Select Case in line 1, the Age value is compared to its many possibilities throughout the rest of the Select Case. If Age holds a value less than 5, the label is updated in line 2 to reflect the child s age. Nothing else executes. Visual Basic recognizes that the first Case in line 2 is a one-line Case, and the program continues with the line that follows End Select (line 12) as soon as line 2 finishes.

Line 3 compares the Age variable to the value of 5, and assigns "Kindergarten" to the label if Age is equal to 5.

Lines 4 9 compare whether Age falls within three different ranges of values and updates two label values accordingly. If all Case statements fail through line 9, the Case in line 10 takes over and assumes that Age holds a value greater than 18. (If Age holds any value less than 18, an earlier Case statement would have taken control.)

## Homework

### General Knowledge

1. What is a condition?
2. True or false: Relational tests produce one of three results.
3. True or false: Visual Basic requires parentheses around a relational test.
4. What are the six relational operators?
5. Determine the true or false result of the following relational tests:
   . "zz" < "ZZ"
   B. 161 <= 161

C. 3.4 < 4
D. (3 <> 4) And (5 = 5)
E. (3 <> 4) Or (5 = 5)
F. (3 = 3) And (10 < 9)

6. What is the name of the table that determines how string values are compared?
7. What Visual Basic statement makes decisions?
8. What Visual Basic statement prescribes a true *and* a false course of action?
9. True or false: The body of an If can contain more than one statement, but the body of the Else must contain one statement at most.
10. What is a *nested If* statement?
11. What statement helps eliminate tedious and convoluted If-Else statements?
12. How many kinds of Case options are there?
13. What happens if every Case fails and there is no Case Else option for a particular Select Case statement?
14. What happens if every Case fails and there is a Case Else option on a particular Select Case statement?
15. Which Case option tests for ranges of values?
16. Which Case option tests for relational values?

## Write Code That...

1. Rewrite the following nested If statement using a single If with a logical operator:
```
If (M = 3) Then
If (P = 4) Then
TestIt = "Yes"
End If
End If
```
2. Rewrite the following If to eliminate the Not to clarify the code:
```
If Not(d < 3) Or Not(p >= 9) Then
```
3. Rewrite Listing 8.6 so that the message "Age Error" appears in the lblTitle.Text property if Age holds a value less than zero.

## Find the Bug

1. What is wrong with the following If statement?
```
If (A < 1) And (C >= 8) Then
```

```
lblName.Text = "Overdrawn"
Else
lblName.Text = "Underdrawn"
End Else
End If
```

## Extra Credit

Given the following If and matching Select Case, which one is preferable and easier to maintain? Remember that you want to keep things simple and clear.

```
If (grade >= 70) Then
lblLetter.Text = "Passing"
Else
lblLetter.Text = "Failing"
End If
```

or

```
Select Case grade
Case Is >= 70: lblLetter.Text = "Pasing"
Case Else: lblLetter.Text = "Failing"
End Select
```

# Visual Basic in 12 Easy Lessons

# Appendix A

## Operator Precedence

There are three sets of operators, arithmetic, comparison, and logical operators. This appendix's three tables describe each operator set's order of precedence.

**Table A.1. The arithmetic operator precedence.**

| *Level* | *Operator* |
|---|---|
| 1 | Exponentiation (^) |
| 2 | Negation (-) |
| 3 | Multiplication and division (*, /) |
| 4 | Integer division (\) |
| 5 | Modulo division (Mod) |
| 6 | Addition and subtraction (+, -) |
| 7 | String concatenation (&) |

**Table A.2. The comparison precedence.**

| *Level* | *Operator* |
|---|---|
| 1 | Equality (=) |
| 2 | Inequality (<>) |
| 3 | Less than (<) |
| 4 | Greater than (>) |
| 5 | Less than or equal to (<=) |
| 6 | Greater than or equal to (>=) |
| 7 | Like |

**Table A.3. The logical operator precedence.**

| *Level* | *Operator* |
|---|---|
| 1 | Not |
| 2 | And |
| 3 | Or |
| 4 | Xor |
| 5 | Eqv |
| 6 | Imp |
| 7 | Is |

# Visual Basic in 12 Easy Lessons



# Project 4

## Data Basics

### Stop & Type

This lesson taught you how Visual Basic stores data values in several different formats. The different data types enable you to categorize data. For example, when you are working with currency amounts, use the currency data type so that Visual Basic ensures accuracy to two decimal places.

The key to manipulating data in Visual Basic programs is the variable. A variable is a named storage location in memory that you define by using the Dim statement. The Dim statement requests that Visual Basic set aside the memory, attach one data type to the variable, and name the variable. Once you define variables, the assignment statement stores values in the variables. Until you store values in variables, the numeric variables hold zeros and the string variables hold null strings.

Once the variables receive values, the If and Select Case statements determine the appropriate program paths to take. The If and Select Case analyze the data in variables and make runtime decisions based on the values there.

In this lesson, you saw the following:

- How to distinguish the data types
- Why you must define and name variables
- How to use Visual Basic's mathematical operators
- When the operator precedence table becomes crucial

- What to do when you want to override the operator precedence
- How to code If-Else and Select Case statements to make decisions
- Why the various Case options exist

## The Program's Description

Figure P4.1 shows how the PROJECT4.MAK application looks as soon as you load and run the program. The project's form contains several controls. Notice the following about the controls on the form:

The user must enter values in three text boxes that correspond to the number of inventory items sold, the price per item, and a special discount code. The user must type either a 1, 2, 3, or 4 to indicate which discount percentage is desired for the total price.

The Calculate Inventory command button computes the value of the inventory, given the user's entries in the text boxes, and displays the amount of the inventory in the lower label marked Extended Amount.

An Exit command button terminates the program.

**Figure P4.1. The project's opening screen.**

The program acts like a smart but simple cash register that extends the total inventory sold price and adds a discount based on a discount code. There are several ways to support such a discount list in Visual Basic, and this program demonstrates only one method. In future lessons, you will learn how to add option button controls and scrolling list boxes to support such discount percentage codes.

---

**Note:** Assume that the company supports only the four discount codes described. If the user leaves the discount code blank, the program does not compute a discount code. If the user had to type a discount code, it would be too easy for the user for forget the decimal point if you required the user to enter a decimal discount like .15. The user might also enter the percent sign, which could cause program confusion as well. Therefore, for percentage values, the table of discount codes works very well and eliminates much user and programmer frustration.

---

## The Program's Action

Enter the following values in the text box controls:

Units Sold: **20**

Price Per Unit: **2.75**

Discount Code: **3**

Press the Calculate Inventory command button (Alt+C). Visual Basic calculates the full inventory price and displays the answer 46.75 to the right of the Extended Amount label.

Even though this project works with currency data types, the displayed answer does not always appear with two decimal places. For example, entering a Price Per Unit amount of 2.00 produces the Extended Amount of 34 with no decimal places showing. Visual Basic never makes assumptions about how you want numeric data displayed. Even with currency values, Visual Basic will not display two decimal places unless you specifically request two decimal places. Lesson 7 shows you how to format numeric output to look exactly as you want it. Until then, be lenient with your own programs and the programs in this book, because the decimal places will not always work out the way you would prefer. It is more important, until Lesson 7, to concentrate on how the results are produced, not how the results are formatted to look.

## Checking the Discount Code

One of the most powerful features of the PROJECT4.MAK application is its use of the If statement in the discount code's text box LostFocus event procedure. The LostFocus event occurs when the user moves the focus from the discount code text box to another control. Therefore, the txtDisc_LostFocus() event procedure, shown in Listing P4.1, executes immediately after the user enters a value for the discount code.

---

**Note:** Line 5 uses a Visual Basic element called a *method,* which you have not seen yet. A method works almost like a built-in function, such as Val(). However, instead of converting data, a method performs an action for a particular control. You must request a method by specifying a control, followed by a period and the name of the method.

---

**Listing P4.1. Ensure that the user enters a proper discount code.**

```
1: Sub txtDisc_LostFocus ()

2: If (Val(txtDisc.Text) < 0) Or (Val(txtDisc.Text) > 4) Then

3: Beep

4: txtDisc.Text =
""
```

```
5: txtDisc.SetFocus

6: End If

7: End Sub
```

1. The discount text box's Name property contains txtDisc. The name of the LostFocus event procedure is txtDisc_LostFocus().
2. Use Val() to convert the text box value to a number while it tests to make sure that the user entered a number from 0 to 4.
3. The body of the If executes only if the user enters a bad discount code. The Beep statement beeps the computer's speaker to get the user's attention.
4. This line erases whatever value the user entered in the text box.
5. This line returns the focus to the text box so that the user is forced to enter a good value before doing anything else.
6. This line ends the body of the If statement.
7. This line terminates the event procedure.

---

3: Audibly warns the user of an error.

---

## Computing the Inventory

When the user clicks the Calculate Inventory command button, the command button's Click event procedure, shown in Listing P4.2, executes. The event procedure uses a combination of variables and a Select Case statement to compute the proper inventory amount based on the user's text box values at the discount specified.

**Listing P4.2. Computing the inventory amount.**

```
1: Sub cmdInven_Click ()

2: Dim Discount As Single

3: Dim ExtAmount As Currency

4: ExtAmount = Val(txtUnits.Text) *
```

```
Val(txtPrice.Text)

5: Select Case Val(txtDisc.Text)

6: Case 0: Discount = 0

7: Case 1: Discount = .05

8: Case 2: Discount = .1

9: Case 3: Discount = .15

10: Case 4: Discount = .2

11: End Select

12: lblExt.Caption = ExtAmount - (Discount *
ExtAmount)

13: End Sub
```

1. The command button's Name property contains cmdInven, so the name of the Click event procedure is cmdInven_Click().
2. A single-precision variable named Discount is defined to hold an intermediate calculation.
3. A currency variable named ExtAmount is defined to hold an intermediate calculation.
4. The first part of the extended inventory amount is computed by multiplying the user's number of units sold by the price per unit.
5. Select Case makes a decision based on one of four values stored in the txtDisc text box.
6. If the discount code is 0, zero is used for the discount percentage.
7. If the discount code is 1, 5% is used for the discount percentage.
8. If the discount code is 2, 10% is used for the discount percentage.
9. If the discount code is 3, 15% is used for the discount percentage.
10. If the discount code is 4, 20% is used for the discount percentage.
11. The body of the Select Case comes to an end.
12. This line finishes computing the discount amount by applying the discount to the extended price, and it displays the final amount on the form.

---

4: Always convert control values to numbers by using Val() before you calculate with the values.

---

---

11: No Case Else is required because the txtDisc LostFocus() procedure ensures that only valid values appear in txtDisc.

---

## Close the Application

This project helped solidify your understanding of how and when to use variables for data. You can now exit the application and exit Visual Basic. The next lesson adds additional programming skills to your repertoire and increases the power of the programs that you can write and control.

# Visual Basic in 12 Easy Lessons



# Lesson 5, Unit 9

## Remarks and Message Boxes

### What You'll Learn

- The need for remarks
- Remarkable code
- Introduction to message and input boxes
- The MsgBox statement
- The MsgBox() function
- The InputBox() functions

There's more to Visual Basic programs than forms, visual controls, and code. Often, you'll include messages called *remarks* within your programs that Visual Basic, Windows, and your computer completely ignore. The remarks aren't for the computer but they are for programmers.

At other times, you'll need to display messages to users and get answers from users, and no kind of control or Visual Basic object works better than *message boxes* and *input boxes*. This unit teaches you how to display and manage message boxes and input boxes. You'll learn how and when message and input boxes work more effectively than labels and text box controls.

# The Need for Remarks

***Concept:*** Remarks help both you and other programmers who might modify and update your Visual Basic applications in the future. Remarks offer descriptive messages that explain in English (or whatever language you prefer) what's going on in the program's event procedures.

It's said that a program is written once and read many times. That saying is true because of the nature of applications. Often, you'll write a program that helps you or your business compute required calculations and keep track of daily transactions. Over time, requirements change. Businesses buy and sell other businesses, the government changes its reporting and taxing requirements, and people's needs change. You should realize that, after you write and implement a program, you'll make modifications to that program later. If you use the program in a business, you'll almost certainly make many modifications to the program to reflect changing conditions.

---

*Definition:* Program *maintenance* refers to the modification of programs over time.

---

---

**Note:** If you program for someone else, the chances are high that others will modify the programs that you write and that you'll modify programs that other programmers write. Therefore, as you write programs, think about the future maintenance that you and others will make. Write your programs clearly, using ample spacing and indention, and add remarks that explain difficult sections of code.

---

A *remark* is a message that you put inside a program's code. Programmers concerned about maintenance know that ample remarks help clarify code and aid future maintenance. Visual Basic completely ignores any and all remarks because those remarks are for people looking at your program code. Users don't see remarks because users don't see the program's code; rather, users see a program's output.

Programmers often add remarks to their programs for the following purposes:

- To state the programmer's name and the date that the program was written
- To describe in the (general) procedure the overall goal of the program
- To describe at the top of all procedures the overall goal of that procedure
- To explain tricky or difficult statements so that others who modify the program later can understand the lines of code without having to decipher cryptic code

---

**Note:** Even if you write programs for yourself, and if you are the *only* one who will modify your programs, you should *still* add remarks to your program! Weeks or months after you write a program, you'll have forgotten the exact details of the program and remarks that you interspersed throughout the code will simplify your maintenance and will help you find the code that you need to change.

---

---

**Tip:** Add remarks *as you write your programs.* Often, programmers will say to themselves, "I'll finish the program and add remarks later." Trust me the remarks don't get added. It's only later, when programmers need to modify the program, that they notice the lack of remarks and regret it.

---

***Review:*** Add remarks to your program so that you and others can more quickly grasp the nature of the program and can

make modifications to it more easily when needed.

## Remark-able Code

***Concept:*** Visual Basic supports several remark formats. Unlike some other programming languages, Visual Basic remarks are easy to add to your code, and their free-form nature enables you to add remarks whenever and wherever needed.

Visual Basic supports the following two kinds of remarks:

- Remarks that begin with the Rem statement
- Remarks that begin with the apostrophe (')

The Rem statement is more limiting than the apostrophe and isn't as easy to use as apostrophes. Nevertheless, you'll run across programs that use Rem statements, so you should learn how Rem works. Here is the format of the Rem statement:

```
Rem The remark's text
```

You can put anything you want in place of *The remark's text*. Therefore, all of the following are remarks:

```
Rem Programmer: Bob Enyart, Date: Mar-27-1996

Rem This program supports the check-in and check-out

Rem
process for the dry-cleaning business.

Rem This event procedure executes when the user

Rem clicks on the Exit command button. When pressed,

Rem this event procedure closes the program's data

Rem files, prints an exception report, and terminates

Rem
the application
```

The first of these remark sections consists of a one-line remark that tells the programmer's name and the date that the program was last modified. If someone else must modify the program later, that person can find the original programmer if needed to ask questions about the program's code. The second remark describes the overall program's goal by stating with a high-level description the program's purpose. The third remark might appear at the top of a command button's Click event procedure.

As you can see, you can add one or more lines of remarks depending on the amount of description needed at that point in the program. Visual Basic ignores all lines that begin with Rem. When someone looks at the program code later, that person will know who the programmer is, the date that the program was written, the overall purpose of the program, and

the overall description of each procedure that includes a remark section.

Say that you used apostrophes in place of the Rem statement in the previous remarks. The following rewritten remarks demonstrate that the remarks are even more effective because Rem doesn't get in the way of the remark's text:

```
' Programmer: Bob Enyart,
Date: Mar-27-1996

' This program supports the check-in and check-out

' process for the dry-cleaning business.

' This event procedure executes when the user

' clicks on the Exit command button. When pressed,

' this event procedure closes the program's
data

' files, prints an exception report, and terminates

' the application
```

The remarks don't have to go at the beginning of event procedures. You can place remarks between lines of code, as done here:

```
Dim Rec As Integer

Rem Step through each customer record

For Rec = 1 To NumCusts

 ' Test for a high balance

 If custBal(Rec) > 5000 Then

 Call PayReq

 End If

Next Rec
```

---

**Note:** Don't try to understand the details of this code yet. Concentrate now on the remarks. The code contains some advanced features (Visual Basic *arrays* and *subroutine procedures*) that you'll learn about in the last half of this book.

---

The apostrophe remark offers another advantage over using Rem because you can place apostrophe remarks at the end of Visual Basic statements. By placing a remark to the right of certain lines of code, you can clarify the purpose of the code. Consider how the following code section uses remarks to explain specific lines of code:

```
a = 3.14159 * r ^ r ' Calculate a circle's area
```

Perhaps only a mathematician could interpret the formula without the remark. The remark helps even nonmathematicians understand the purpose of the statement. There is no reason that you should have to reexamine code every time you look at code. By reading remarks, you can gleam the code's purpose without taking the time to interpret the Visual Basic code.

The wrong kind of remarks won't help clarify code, though, so don't overdo remarks. As a matter of fact, lots of lines of code need no remarks to explain their purpose. The following remark is redundant and wastes both your programming time and anyone's who may maintain the program later:

```
Dim Sales As Single ' Define a variable named
Sales
```

***Stop and Type:*** Listing 9.1 contains a Select Case routine that you saw in Listing 8.5 of the previous unit. This code contains remarks that help clarify the purpose of the code.

***Review:*** Add remarks throughout a Visual Basic program, using either the Rem statement or the apostrophe, to tell other programmers as well as yourself what the program is doing. Add the remarks as you write the program so that the remarks will be up to date and always reflect your intent.

```
1: Rem The following Select Case to End Select code

2: Rem assigns a student's grade and school name

3: Rem to the label on the form. The
code checks

4: Rem to make sure that the student is not too

5: Rem young to be going to school.

6: Select Case Age

7: ' Check for too young...

8: Case Is <5: lblTitle.Text = "Too young"
```

```
9: ' Five-year olds are next assigned

10: Case 5:
lblTitle.Text = "Kindergarten"

11: ' Six to eleven...

12: Case 6 To 11: lblTitle.Text = "Elementary"

13: lblSchool.Text = "Lincoln"

14: ' Twelve to fifteen...

15: Case 12 To 15: lblTitle.Text = "Intermediate"


16: lblSchool.Text = "Washington"

17: ' Sixteen to eighteen

18: Case 16 To 18: lblTitle.Text = "High School"

19: lblSchool.Text = "Betsy Ross"

20: ' Everyone else must go to college

21: Case Else: lblTitle.Text =
"College"

22: lblSchool.Text = "University"

23: End Select
```

***Analysis:*** Indention was used in Listing 9.1's remarks to put the remarks where they would be most effective. The remarks on lines 7, 9, 11, 14, 17, and 20 don't appear at the end of their respective code lines because the remarks would hang too far out to the right and would not fit in the Code window. Nevertheless, the remarks do clarify each Case in the sequence. Notice that the first few lines, 1 through 5, give an overall descriptive account of the code that follows.

## Introduction to Message and Input Boxes

---

*Definition: Message boxes* perform specialized program output.

There will be many times in programs when you'll need to ask the user questions or display error messages and advice to the user. Often, the controls on the form won't work well for such dialogs. For example, in the Lesson 4's project, the user had to enter 0 through 4 when asked for a discount percentage code. If the user entered an incorrect code, the program beeped and erased the user's bad entry, but the program didn't tell the user why the entry was a mistake. Users don't always know what's expected of them. The user would be helped much more by seeing an error message, such as the one shown in Figure 9.1, before the program cleared the user's incorrect entry.

**Figure 9.1. A message box can provide the user with helpful information.**

**Note:** Message boxes aren't controls. Unlike controls that stay on the form throughout a program's entire execution cycle, a message box pops up on top of the form and disappears when the user responds to the message box, usually by clicking the message box's OK command button.

There are two ways to produce message boxes. You can use the MsgBox statement or the MsgBox() function. You've seen one built-in function before, Val(), which converts a string of digits to a number. Visual Basic includes several built-in functions; you'll learn many of them in Lesson 7, "Functions and Dates." Until Lesson 7, however, you need to learn a few of the functions such as Val() and MsgBox(), because they help you work with the commands that come before Lesson 7. The choice of using the MsgBox statement or the MsgBox() function depends on the response that you need the user to make to the display of the message box.

*Definition: Input boxes* perform specialized program input.

The text box controls that you've seen are great for getting values from the user. Other controls that you'll learn as you progress through this book also accept the user's input from the keyboard or mouse. Nevertheless, Visual Basic's controls just aren't enough to handle all the input that your program will need. Input boxes are great to use when the user must respond to certain kinds of questions. Text boxes and other controls are fine for getting fixed input from the user, such as data values with which the program will compute. Input boxes are great for asking the user questions that arise only under certain conditions. Input boxes always give the user a place to respond with an answer. In Figure 9.2, the input box is asking the user for a title that will go at the top of a printed report listing.

**Figure 9.2. Use input boxes to request information.**

Note that there is more than one way for the user to respond to Figure 9.2's input box. The user can answer the question by typing the title at the bottom of the input box and pressing Enter or clicking the OK command button. The user also can click the Cancel command button whether or not the user entered a title. Therefore, the program must be capable of reading the user's entered answer as well as responding to a Cancel command button press. Responding to message box and input box command buttons is part of the processing that you'll learn about in the remaining sections of this chapter.

***Review:*** Message boxes display output and input boxes get input. The message and input boxes offer ways for your programs to request information that regular controls can't handle. Use controls to display and get data values that are always needed. Use message and input boxes to display messages and get answers that the program needs in special cases, such as for error conditions and exception handling.

## The MsgBox Statement

***Concept:*** The MsgBox statement displays messages for the user. In addition, the MsgBox() function displays messages but also provides a way for your program to display and check for multiple command button clicks on the message box

window. This section explains how to use the MsgBox statement.

All message boxes displayed with the MsgBox statement display an OK command button. The command button gives the user a chance to read the message. When the user has finished reading the message, the user can click OK to get rid of the message box. Your program suspends execution during the reading of the message box. Therefore, the statement following the MsgBox statement executes when the user clicks OK.

Here is the format of the MsgBox statement:

```
MsgBox msg [, [type] [,
title]]
```

The *msg* is a string (either variable or a string constant enclosed in quotation marks) and forms the text of the message displayed in the message box. The *type* is an optional numeric value or expression that describes the options you want in the message box. Figure 9.3 shows which icons you can place in a message box (Visual Basic displays no icon if you don't specify a *type* value). The *title* is an optional string that represents the text in the message box's title bar. If you omit the *title*, Visual Basic uses the project's name for the message box's title bar text.

**Figure 9.3. The icons that you can display in a message box.**

---

*Definition: Modality* determines whether the user or system responds to a message box.

---

The options that you select, using the *type* value in the MsgBox statement, determine whether the message box displays an icon as well as controls the modality of the message box. Table 9.1 contains the code values that you'll use to form the MsgBox *type* value.

**Table 9.1. The MsgBox statement's *type* values.**

| *Value* | *CONSTANT.TXT Value* | *Description* |
|---|---|---|
| 16 | MB_ICONSTOP | Displays the stop sign icon |
| 32 | MB_ICONQUESTION | Displays the question mark icon |
| 48 | MB_ICONEXCLAMATION | Displays the exclamation icon |
| 64 | MB_ICONINFORMATION | Displays the lowercase i (meaning *information*) icon |
| 4096 | MB_SYSTEMMODAL | The user's application is *system modal*, meaning that the message box must be handled before you can switch to any other Windows program |

The modality often causes confusion. If you don't specify the system model *type* value of 4096 (or if you don't use CONSTANT.TXT's MB_SYSTEMMODAL named constant to specify the system's modal mode), the user's application won't continue until the user closes the message box, but the user *can* switch to another Windows program by pressing Alt+Tab or by switching to another program using the application's control menu. If, however, you do specify that the message box is system modal, the user won't able to switch to another Windows program until the user responds to the message box, because the message box will have full control of the system. Reserve the system modal message boxes for serious error messages that the user *must* read and respond to before continuing with the program.

---

**Note:** If you don't specify an icon, Visual Basic doesn't display an icon. If you don't specify the system modality, Visual Basic assumes that you want an application modal message box.

---

***Stop and Type:*** Listing 9.2 ought to answer questions that you have about message boxes. The listing contains a series of message box statements. Each statement displays a different message box and each message box is different.

***Review:*** Use the MsgBox statement to display messages, such as error messages, to the user. The message boxes will contain an OK button that the user clicks to close the message box. Depending on whether you specify a *type* value, the message box might contain an icon as well as be system modal. If you specify a third *title* value, the message box's title bar will contain the text of the title you request.

**Listing 9.2. Displaying different message boxes.**

```
1: ' A simple message box with a message and no icon

2: MsgBox "Just a message"

3: ' A message box with the stop sign icon

4:
MsgBox "Stop in the name of love", MB_ICONSTOP

5: ' A message box with the question mark icon

6: MsgBox "Did you turn on the printer?", MB_ICONQUESTION

7: ' A message box with the exclamation icon

8: MsgBox "Don't forget to
exit!", MB_ICONEXCLAMATION

9: ' A message box with the "i" information icon as

10: ' well as a title that's not the project name

11: MsgBox "A byte is 8 bits", MB_ICONINFORMATION, "A title"

12: ' A message box that is
system modal

13: MsgBox "You cannot switch programs", MB_SYSTEMMODAL

14: ' A message box with a question mark icon, a system modal setting,

15: ' and a title of Title
```

```
16: MsgBox "Info icon, system modal, and a title",
MB_ICONQUESTION + MB_SYSTEMMODAL, "Title"
```

[ic:Output]Listing 9.2 demonstrates several different kinds of message boxes. Assuming that the project name is PROJECT1.MAK, Figure 9.4 shows the simple message that appears from line 2's MsgBox statement.

**Figure 9.4. A simple message box with no icon or given title.**

***Analysis:*** Note that there is no icon because line 2 doesn't contain a *type* value. Also, the message box's title is Project1 because the project's name is PROJECT1.MAK and the MsgBox statement doesn't specify a different title. The message box also is application modal. The user must respond to the message before continuing with the program. The user can switch to another Windows program, however, by pressing Alt+Tab, or Alt+spacebar, or by selecting from the control menu.

Line 4's MsgBox statement displays a stop sign icon to the left of the message. Rather than use the CONSTANT.TXT named constant, you can type 16 for the *type* value.

Line 6's MsgBox statement displays a question mark icon to the left of the message. Line 8's MsgBox statement displays an exclamation point icon to the left of the message.

Line 11 displays a message as well as the information icon. The information icon is a nice touch to add when offering advice to the user. Line 11 also displays a title in the message box's title bar. The title is simple: A title.

Line 13's MsgBox produces a system modal message box. Line 14's MsgBox statement displays a message, an information icon, and a title, a title; this line is a system modal message box. Figure 9.5 shows Line 13's message box output.

---

**Note:** Note that Visual Basic expands or contracts the message boxes to hold the full text that you want to display.

---

**Figure 9.5. A complete message box statement produces a message, icon, and title.**

As you can see from Line 13's *type* value, you can add together an icon's value as well as the system modal value (or their named constants, as done in Listing 9.2) to obtain both an icon as well as a system modal message box.

---

**Tip:** If you want to specify a title in a message box but *not* display an icon or change to system modality, insert two commas before the title string, as follows :
MsgBox "A byte is 8 bits", , "A title"

---

## The MsgBox() Function

***Concept:*** Use the MsgBox() function when you want the user to indicate a response to the message in the message box. By using the MsgBox() function, you can display several different command buttons inside the message box and determine which command button the user pressed so that you'll know how the user responded to the message.

Figure 9.6 shows a message box that looks a little different from the other ones that you've seen so far. Instead of a single OK command button, Figure 9.6's message box contains three command buttons. The MsgBox() function enables you to program message boxes with multiple command buttons and then determine which command button the user pressed to close the message box.

**Figure 9.6. Message boxes can have several command buttons.**

The format of the MsgBox() function is almost identical to that of the MsgBox statement. You must use the MsgBox() function differently from the statement, however. Always assign a MsgBox() function to a variable. Here is the format of the MsgBox() function:

```
anIntVariable = MsgBox( msg [, [type] [, title]])
```

As you can see from the format, the MsgBox() function's format differs from the MsgBox statement in that you assign the MsgBox() function to an integer variable that you've already defined. In addition, the MsgBox() function supports several more *type* values than the MsgBox statement supported. Table 9.2 lists the MsgBox() function's *type* values along with their CONSTANT.TXT named constant equivalents.

**Table 9.2. The MsgBox() function's *type* values.**

| *Value* | *CONSTANT.TXT Value* | *Description* |
|---|---|---|
| 0 | MB_OK | The OK button appears only in the message box |
| 1 | MB_OKCANCEL | The OK and Cancel buttons appear |
| 2 | MB_ABORTRETRYIGNORE | The Abort, Retry, and Cancel buttons appear |
| 3 | MB_YESNOCANCEL | The Yes, No, and Cancel buttons appear |
| 4 | MB_YESNO | The Yes and No buttons appear |
| 5 | MB_RETRYCANCEL | The Retry and Cancel buttons appear |
| 16 | MB_ICONSTOP | Displays the stop sign icon |
| 32 | MB_ICONQUESTION | Displays the question mark icon |
| 48 | MB_ICONEXCLAMATION | Displays the exclamation icon |
| 64 | MB_ICONINFORMATION | Displays the lowercase i (meaning *information*) icon |
| 0 | MB_DEFBUTTON1 | The first button has the initial focus |
| 256 | MB_DEFBUTTON2 | The second button has the initial focus |
| 512 | MB_DEFBUTTON3 | The third button has the initial focus |
| 4096 | MB_SYSTEMMODAL *system* message | The user's application is *modal*, meaning that the box must be handled before you can switch to any other Windows program |

Here is the MsgBox() function that displayed the message box shown in Figure 9.6:

```
BPress = MsgBox("Are you ready for the report?", MB_ICONQUESTION + MB_YESNOCANCEL,
"Report Request")
```

The MB_ICONQUESTION named constant added to the MB_YESNOCANCEL named constant produced both a question mark icon and the three buttons. A title also appeared due to the third value inside the MsgBox() function.

The reason that you assign MsgBox() functions to variables is so that you can tell what button the user pressed. Suppose that the user pressed the Yes button in Figure 9.6. The program could then print the report. If, however, the user pressed the No command button, the program could describe what the user needed to do to get ready for the report (load paper, turn on the printer, and so on). If the user pressed the Cancel command button, the program would know that the user didn't want the report at all. Of course, the application determines what set of command buttons will work best for any given message box.

Table 9.3 lists the possible return values for the MsgBox() function. In other words, the integer variable will contain one of Table 9.3's values after every MsgBox() function completes. A subsequent If statement can then test to see which command button the user pressed.

**Table 9.3. The MsgBox() function's return command button values.**

| *Value* | *CONSTANT.TXT Value* | *Description* |
|---|---|---|
| 1 | IDOK | The user pressed the OK button |
| 2 | IDCANCEL | The user pressed the Cancel button |
| 3 | IDABORT | The user pressed the Abort button |
| 4 | IDRETRY | The user pressed the Retry button |
| 5 | IDIGNORE | The user pressed the Ignore button |
| 6 | IDYES | The user pressed the Yes button |
| 7 | IDNO | The user pressed the No button |

***Stop and Type:*** Listing 9.3 shows how you might handle the previous MsgBox() function that displayed the three-button message box in Figure 9.6. After the user clicks a button, the program can use If statements to determine which button the user pressed.

**Listing 9.2. Displaying different message boxes.**

```
1: BPress = MsgBox("Are you ready for the report?", MB_ICONQUESTION +
MB_YESNOCANCEL, "Report Request")

2: ' Check the button pressed

3: Select BPress

4: Case IDCANCEL: lblPress.Text = "You pressed Cancel"

5: Case IDYES: lblPress.Text = "You pressed Yes"

6: Case IDNO: lblPress.Text =
"You pressed No"

7: End Select
```

***Analysis:*** Line 1 displays the message box and saves the button press in the variable named BPress. Line 3 begins a Select Case that assigns one of three strings to labels on the form that match the user's button press. This example is very simple; normally, you would perform one of several different kinds of routines, depending on which of the three buttons the user pressed.

There is no need to code a Case Else statement because the three-button message box can return only one of the three values tested in Listing 9.2.

# The InputBox() Functions

***Concept:*** You'll find that the InputBox() function is easy because it acts a lot like the MsgBox() function. There are two InputBox() functions. The difference between them lies in the type of data that each returns. The InputBox() function receives answers that are more complete than the MsgBox() function can get. Whereas MsgBox() returns one of seven values that indicate the user's command button press, the InputBox() function returns either a string or a variant data value that holds the answer typed by the user.

There are two InputBox() functions. Here are the formats of the InputBox() functions:

```
aVariantVariable =
InputBox( prompt [, [title] [, default][, xpos, ypos]]])
```

and

```
aStringVariable = InputBox$( prompt [, [title] [, default][, xpos, ypos]]])
```

The difference between the InputBox() functions lies in the return value. The InputBox() function returns a variant data type and the InputBox$() function (notice the dollar sign) returns a string data type. Generally, the return type is not extremely important. You'll almost always receive the InputBox() answer in a string variable, and if you use the InputBox() function, Visual Basic converts variant data types to strings when needed.

The *prompt* works a lot like the *msg* value in a MsgBox() function. The user sees the *prompt* inside the input box displayed on the screen. The *title* is the title inside the input box's title bar. *default* is a default string value that Visual Basic displays for a default answer, and the user can accept the default answer or change the default answer.

The *xpos* and *ypos* positions indicate the exact location where you want the input box to appear on the form. The *xpos* value holds the number of twips from the left edge of the form window to the left edge of the input box. The *ypos* value holds the number of twips from the top edge of the form window to the top edge of the input box. If you omit the *xpos* and *ypos* values, Visual Basic centers the message box on the form.

---

**Note:** Input boxes always contain an OK command button and a Cancel command button. If the user presses OK (or presses Enter, which selects OK by default), the answer in the input box is sent to the variable being assigned the returned value. If the user presses Cancel, a null string, "", returns from the InputBox() function.

---

***Stop and Type:*** The code in Listing 9.3 displays an input box that asks the user for a company name. The user either enters a response to the prompt or presses the Cancel command button to indicate that no answer is coming.

***Review:*** The InputBox() and InputBox$() functions get answers from users. You can offer a default answer that the user can accept or change. The input box functions return the answer into a string or variant variable to which you assign the function.

**Listing 9.3. The input box asks the user questions.**

```
1: Dim CompName As String
```

```
2: CompName = InputBox$("What is the name of the
company?", "Company Request", "XYZ, Inc.")

3: If (CompName = "") Then

4: ' The user pressed Cancel

5: Beep

6: MsgBox "Thanks anyway"

7: Else

8: ' The user entered a company name

9: MsgBox "You
entered " & CompName

10: End If
```

[ic:Output]Figure 9.7 contains the message box displayed from Listing 9.3.

**Figure 9.7. Input boxes ask the users questions and return the answers.**

***Analysis:*** Listing 9.3 might be part of an event procedure that executes when the program is ready for a company name. Line 1 defines a string variable that holds the user's response. Line 2 contains the InputBox$() function that asks for the company name and displays a default answer (automatically highlighted as shown at the bottom of Figure 9.7).

As soon as the user answers the input box request, Line 3 begins an if that checks for one of two results: either the user pressed the Cancel button in response to the input box, or answered the input box by pressing OK. If the user pressed Cancel, the input box function returns a null string, and lines 5 and 6 beep and display a message box thanking the user for trying. Lines 8 and 9 restate the company name entered by the user.

## Homework

### General Knowledge

1. What is a remark?
2. Why are remarks important?
3. True or false: Remarks appear on the user's screen.
4. True or false: Remarks appear in the Code window.
5. How many kinds of remarks does Visual Basic support?
6. Name three uses for remarks.
7. Who are remarks for?
8. True or false: There is no reason to worry about remarks if you're the only programmer who will work on your

program now or later.

9. True or false: Message boxes are controls.
10. How can you control the number of command buttons that appear on message boxes?
11. How can you control the text that appears on message boxes?
12. What does *modal* mean?
13. True or false: Visual Basic includes a message box statement and a message box function.
14. True or false: Visual Basic includes an input box statement and an input box function.
15. Why would you use an input box rather than a message box?
16. What data types can the input box functions return?
17. How can you check to make sure that the user pressed Cancel in response to an input box?
18. Suppose that you used King George, III as a default value in an input box function. What does the user have to do to use that default value?
19. How many kinds of icons can you display in message boxes?
20. How many kinds of icons can you display in input boxes?
21. How can you control the command button focus with message boxes?
22. How many different command buttons can you display in message boxes?
23. How many return values are there for the MsgBox() functions?

## Write Code That...

1. If the code that you write senses that a particular disk file is corrupted, you want to inform the user of the serious error and you want to make sure that the user can't switch to another Windows program before responding to your error message box. What *type* argument would you use in the MsgBox statement?

**Find the Bug**

1. Bob the programmer knows how important program documentation is. Despite his best efforts, Bob gets errors when he tries to add the following remark to the end of a line of code:
   do until (endOfFile) Rem Continue until the end
   Help Bob with his problem.

## Extra Credit

Write a command button Click routine that asks the user for his or her age. If the user presses Cancel, beep and display another message box saying, Thanks anyway. If the user enters an age, use Val() to convert and store the age in a numeric variable and then display the number of years until the user's retirement by subtracting the age from 65. Hint: The Str$() function performs the opposite of Val() and you can append the converted number to another string to display the message. (Lesson 7 covers Str$() in full detail.)

# Visual Basic in 12 Easy Lessons



# Lesson 5, Unit 10

## Looping

### What You'll Learn

- The Do While Loop
- The Do Until Loop
- The Other Do Loops
- The For Loop

Now you're *really* ready to write powerful programs! You've learned some controls, you've defined some variables, and you've written programs that make decisions. It's now time to learn how to write programs that perform repetitive data processing. When a computer does something over and over, the computer is said to be *looping*.

Computers don't get bored. The primary strength of computers is their capability to loop through a series of calculations over and over very quickly. Computers can process every customer balance, calculate sales averages among many divisions, and display data for each company employee.

This unit describes how you can add looping to Visual Basic programs so that the programs can process several data values using looping statements. Loops don't just help when you have large amounts of data to process. Loops also enable you to correct user errors and repeat certain program functions when the user requests a repeat.

# The Do While Loop

***Concept:*** The Do statement supports several different loop formats. The Do While loop is perhaps the most common looping statement that you'll put in Visual Basic programs.

---

*Definition:* A *block* consists of one or more program statements.

---

The Do While statement works with relational expressions just as the If statement does. Therefore, the six relational operators that you learned about in the previous lesson work as expected here. Rather than control the one-time execution of a single block of code, however, the relational expression controls the looping statements.

The code that you've seen so far inside event procedures has been sequential code that executed one statement following another in the order that you typed the statements. Looping changes things a bit. Many lines of your programs will still execute sequentially, but a loop will cause blocks of code to repeat one or more times.

Like the If statement that ends with an End If statement, a loop will always be a multiline statement that includes an obvious beginning and ending of the loop. Here is the format of the Do While loop:

```
Do While (relational test)

 Block of one or more Visual Basic statements

Loop
```

---

*Definition:* An *infinite loop* repeats forever.

---

The block of code continues looping as long as the *relational test* is true. Whether you insert one or several lines of code for the block doesn't matter. It's vital, however, that the block of code somehow change a variable used in the *relational test*. The block of code keeps repeating as long as the Do While loop's *relational test* continues to stay true. Eventually, the *relational test* must become false or your program will enter an infinite loop and the user will have to break the program's execution through an inelegant means, such as pressing the Ctrl+Break key combination.

---

**Warning:** Even if you provide an Exit command button as you've seen used in this book's applications, the program will often ignore the user's Exit command button click if the program enters an infinite loop.

---

Figure 10.1 illustrates how the Do While loop works. As long as the *relational test* is true, the block of code in the body of the loop continues executing. When the *relational test* becomes false, the loop terminates. After the loop terminates, Visual Basic begins program execution at the statement following the Loop statement because Loop signals the end of the loop.

**Figure 10.1. The Do While loop's action continues while the *relational test* is true.**

---

**Note:** As soon as the *relational test* becomes false, the loop terminates and doesn't execute even one more time. The Do While's *relational test* appears at the top of the loop. Therefore, if the *relational test* is false the first time that the loop begins, the body of the loop will never execute.

---

Throughout this book, you'll see indention used for the body of the loop code. By indenting the body of the loop to the right of the loop's introduction and terminating statements, you'll make it easier to spot where the loop begins and ends.

***Stop and Type:*** Listing 10.1 contains a section of an event procedure that contains a Do While loop that asks the user for an age. If the user enters an age less than 10 or more than 99, the program beeps at the error and displays another input box asking for the age. The program continues looping, asking for the age, as long as the user enters an age that's out of range.

***Review:*** The Do While loop continues executing a block of Visual Basic statements as long as a *relational test* is true. As soon as the *relational test* becomes false, the loop terminates.

**Listing 10.1. The Do While loop executes as long as the *relational test* is true.**

```
1: Dim StrAge As String

2: Dim Age As Integer

3: ' Get the age in a string variable

4: StrAge = InputBox$("How old are you?", "Age Ask")

5: ' Check for the Cancel command button

6: If (StrAge = "")
Then

7: End

8: End If

9: ' Cancel was not pressed, so convert Age to integer

10: Age = Val(StrAge)
```

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

12: Do While ((Age < 10) Or (Age > 99))

13: ' The user's age is out of range

14: Beep

15:
MsgBox "Your age must be between 10 and 99", MB_ICONEXCLAMATION, "Error!"

16: StrAge = InputBox("How old are you?", "Age Ask")

17: ' Check for the Cancel command button

18: If (StrAge = "") Then

19: End


20: End If

21: Age = Val(StrAge)

22: Loop
```

***Output:*** Figure 10.2 shows the message box error displayed in line 15 if the user enters an age value that's less than 10 or more than 99.

**Figure 10.2. The user sees this message box as long as the age is out of range.**

***Analysis:*** Lines 1 and 2 define two variables, a string and an integer. The code uses the string variable to capture the return value from the InputBox$() function. Use a string variable so that you can test for the Cancel button because, as you learned in the previous lesson, InputBox$() returns a null string if the user presses Cancel. If the user presses Cancel, the code terminates the entire program with an End statement (lines 7 and 19). If the user enters an age (and did not press Cancel), the code converts the string age to an integer and checks to make sure that the age is within the range of 10 to 99.

Line 12 begins a loop if the age is less than 10 or more than 99. The loop continues executing from line 13 to the end of the loop in line 22. If the age is out of range, the body of the loop executes. Line 14 beeps, thus sending an audible signal to the user that something is wrong. Line 15 displays an error message box (the one shown in Figure 10.2) and after the user presses OK at the message box, the user is once again asked for an age with an InputBox$() function shown on line 16. The loop then checks to make sure that the user didn't press Cancel (lines 17 through 20) and, if not, the code converts the string age to an integer and the loop begins once again. If the age

entered in the previous pass of the loop falls within the valid age range, the program finishes (any code that exists past line 22 executes). Otherwise, the loop begins once again.

The code contains some redundancy. For example, lines 4 and 16 contain almost the same InputBox$() function, and the same check for a Cancel command button press appears twice in the program. There are other looping statements that you'll learn about later in this chapter; those statements can help simplify this code by removing some of the redundancy.

Perhaps the most important thing to note about the Do While loop in Listing 10.1 is that the body of the loop provides a way for the *relational test* to terminate. Line 12's *relational test* uses the Age variable that the body of the loop reassigns each time the loop's block of code executes. Therefore, assuming that the user enters a different value for the age, the loop will test against a different set of relational values in line 12 and, it is hoped, the relational test will fail (which would mean that the age is inside the range) and the program will stop looping. If the loop body did nothing with the *relational test* variable, the loop would continue forever.

## The Do Until Loop

***Concept:*** Whereas the Do While loop continues executing the body of the loop as long as the *relational test* is true, the Do Until loop executes the body of the loop as long as the *relational test* is false. The program's logic at the time of the loop determines which kind of loop works best in a given situation.

The Do Until loop works almost exactly like the Do While except that the Do Until loop continues executing the body of the loop *until* the *relational test* is true. Like the Do While, the Do Until is a multiline looping statement that can execute a block of code that's one or more lines long.

Here is the format of the Do Until:

```
Do Until (relational test)

 Block of one or more Visual Basic statements

Loop
```

Again, keep in mind that the *relational test* must be *false* for the loop to continue. Figure 10.3 illustrates how the Do Until works.

**Figure 10.3. The Do Until loop's action continues while the *relational test* is false.**

***Stop and Type:*** You can use the Do While or the Do Until for almost any loop. Listing 10.2 contains the age-checking event procedure that contains a Do Until loop. The loop ensures that the age falls between two values. As you can see, the *relational test* for the Do Until is the opposite of that used in Listing 10.1's Do While loop.

---

**Note:** Use the loop that makes for the cleanest and clearest *relational test*. Sometimes, the logic makes the Do While clearer, whereas other loops seem to work better when you set them up with the Do Until loop.

---

***Review:*** The Do Until loop continues executing a block of Visual Basic statements as long as a *relational test* is false. As soon as the *relational test* becomes true (the loop is said to *Do a loop until the condition becomes false*), the loop terminates and the program continues on the line that follows the closing Loop statement.

**Listing 10.2. The Do Until loops until the *relational test* becomes true.**

```
1: Dim StrAge As String

2: Dim Age As Integer

3: ' Get the age in a string variable

4: StrAge =
InputBox$("How old are you?", "Age Ask")

5: ' Check for the Cancel command button

6: If (StrAge = "") Then

7: End

8: End If

9: ' Cancel was not pressed, so convert Age to integer

10: Age = Val(StrAge)

11: ' Loop if the
age is not in the correct range

12: Do Until ((Age >= 10) And (Age <= 99))

13: ' The user's age is out of range

14: Beep

15: MsgBox "Your age must be between 10 and 99", MB_ICONEXCLAMATION, "Error!"

16: StrAge =
InputBox("How old are you?", "Age Ask")
```

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

18: If (StrAge = "") Then

19: End

20: End If

21: Age = Val(StrAge)

22: Loop
```

***Analysis:*** Line 12 is the only line that marks the difference between Listing 10.1 and 10.2. The age must now fall within the valid range for the loop to terminate.

---

**Note:** There is really no advantage of using Do While or Do Until. Use whichever one seems to flow the best for any given application.

---

## The Other Do Loops

***Concept:*** There is another pair of Do loops that works almost exactly like the two previous sections' loops. Do-Loop While and the Do-Loop Until look very much like their counterparts that you learned earlier. Nevertheless, these new loop formats check their *relational test*s at the bottom of the loop rather than at the top.

If a loop begins with a single Do statement, the loop ends with either Loop While or Loop Until. Here is the format of the Do-Loop While:

```
Do

 Block of one or more Visual Basic statements

Loop Until (relational test)
```

---

**Tip:** The dash in Do-Loop While serves to remind you that the body of the loop comes before the Loop While statement. The dash in the Do-Loop Until performs the same purpose.

---

That Do looks lonely by itself, doesn't it? Figure 10.4 illustrates the flow of the Do-Loop While loop's execution.

**Figure 10.4. The Do-Loop While loop doesn't check for the *relational test* until the bottom of the loop body.**

Notice that the Do-Loop While loop's *relational test* appears at the bottom of the loop instead of at the top of the loop. You'll use the Do-Loop While loop when you want the body of the loop to execute at *least one time*. Often, by placing the *relational test* at the bottom of the loop, you can eliminate redundant code that otherwise might be required if you used Do While.

To complete the loop statements, Visual Basic also supports a Do-Loop Until statement. Like the Do-Loop While, the Do-Loop Until statement tests the *relational test* at the bottom of the loop. Therefore, the body of the loop executes at least once no matter what the *relational test* turns out to be. The loop continues as long as the *relational test* remains *false*. Figure 10.5 illustrates the action of the Do-Loop Until.

**Figure 10.5. The Do-Loop Until loop checks for a false *relational test* at the bottom of the loop body.**

***Stop and Type:*** Listing 10.3 contains the age-checking event procedure that's much shorter than the previous versions. The *relational test* appears at the bottom of the loop, so the extra InputBox$() function call is not needed.

***Review:*** The *relational test* appears at the bottom of the loop if you use the Do-Loop While loop statement. The body of the loop always executes at least once. The body of the loop executes more than once as long as the *relational test* stays true. There is a corresponding Do-Loop Until statement that checks for a false condition at the bottom of the loop's body.

**Listing 10.3. Use the Do-Loop While to check the relation at the bottom of the loop.**

```
1: Dim StrAge As String

2: Dim Age As Integer

3: Do

4: StrAge = InputBox("How old are you?", "Age Ask")

5: ' Check for the Cancel command button

6: If (StrAge = "") Then

7: End

8: End If

9: Age =
Val(StrAge)

10: If ((Age < 10) Or (Age > 99)) Then
```

```
11: ' The user's age is out of range

12: Beep

13: MsgBox "Your age must be between 10 and 99", MB_ICONEXCLAMATION, "Error!"

14: End If

15: Loop While ((Age < 10) Or (Age
> 99))
```

***Analysis:*** The loop begins almost immediately in line 3. The loop's body will *always* execute at least once, so the InputBox$() appears right inside the loop. By placing the InputBox$() function inside the loop, you eliminate the need to put this function in the code twice (once before the loop and once inside the loop, as was necessary using the previous looping statements in Listings 10.1 and 10.2).

Line 10 must check to make sure that the InputBox$() value is in or out of the age range so that the message box error can be displayed.

---

**Note:** In this simple application of the looping statements that you've seen here, the Do-While loop required less code than the Do While and Do Until loops. By changing line 15's *relational test*, a Do Until would also work. These last two loops will not, in every case, produce less code. The logic of the application determines which loop works best.

---

## The For Loop

***Concept:*** The For loop (sometimes called the For-Next loop) also creates a loop. Unlike the Do loops, however, the For loop repeats for a specified number of times. The format of the For loop looks a little more daunting than the Do loops, but after you master the format, you'll have little trouble implementing For loops when your code needs to repeat a section of code for a specified number of times.

There isn't one correct loop to use in all situations. The For statement provides the mechanism for the fifth Visual Basic loop construction that you'll learn. A For loop always begins with the For statement and ends with the Next statement. Here is the format of the For loop:

```
For CounterVar = StartVal To EndVal [Step IncrementVal]

 Block of one or more
```

```
Visual Basic statements

Next CounterVar
```

The loop in Listing 10.4 computes the total of the numbers from 1 to 10.

**Listing 10.4. Add the numbers from 1 to 10.**

```
1: Sum = 0

2: For Number = 1 To 10

3: Sum = Sum + Number

4: Next Number
```

Number is the *CounterVar* in the For's format (line 2). The *CounterVar* must be a variable. 1 is the *StartVal* (line 2). The *StartVal* can be either a Number, expression, or variable. 10 is the *EndVal* (still in line 2). The *EndVal* can be either a Number, expression, or variable. There is no Step specified here. In the For statement's format, the Step *IncrementVal* is optional. If you don't specify a Step value, Visual Basic assumes a Step value of 1. Therefore, both of the following For statements are exactly the same:

```
For Number = 1 To 10
```

and

```
For Number = 1 To 10 Step 1
```

Listing 10.4's summing For loop initially assigns to the *CounterVar* the *StartVal* in line 2. Therefore, Number is assigned 1 at the top of the loop. Visual Basic then executes the body of the loop using the value of 1 for Number. With Number being equal to 1, line 3 works as follows the first time through the loop:

```
Sum = Sum + 1
```

When Visual Basic executes the Next Number statement, Visual Basic returns to the top of the loop (the For statement), adds the Step value of 1 to Number, and continues the loop again using 2 as Number in the loop's body. Therefore, the second time through the loop, line 3 works as follows:

```
Sum = Sum + 2
```

The loop continues, adding the default Step value of 1 to Number each time that the loop executes. When Number becomes 10 (the *EndVal*), the loop finishes and the statement following the Next statement continues.

---

**Tip:** Remember, the For loop terminates when the *CounterVar* becomes larger than the *EndVal*. There's an exception to this: If you code a negative Step value, the loop terminates when the *CounterVar* becomes smaller than the *EndVal*, as you'll see a little later in this section.

---

You don't need the For statement to sum the values of 1 through 10. You could code one long assignment statement like this:

```
Sum = 1 + 2 + 3 +
4 + 5 + 6 + 7 + 8 + 9 + 10
```

You could also code back-to-back assignment statements like this:

```
Sum = Sum + 1

Sum = Sum + 2

Sum = Sum + 3

Sum = Sum + 4

Sum = Sum + 5

Sum = Sum + 6

Sum = Sum + 7

Sum =
Sum + 8

Sum = Sum + 9

Sum = Sum + 10
```

Neither of these approaches is extremely difficult, but what if you needed to add together the first one hundred integer Numbers? The previous assignments could become tedious indeed, but the For loop to add the first one hundred integers is just as easy to code as for the first 10 integers, as Listing 10.5 demonstrates.

**Listing 10.5. Add the Numbers from 1 to 100.**

```
1:
Sum = 0

2: For Number = 1 To 100 ' Only this line changes

3: Sum = Sum + Number

4: Next Number
```

The following loop displays five message boxes:

```
For c = 1 To 20 Step 4

 MsgBox "This is a message
box"

Next c
```

---

*Definition:* A loop *iteration* is one full loop cycle.

---

The loop counts up from 1 to 20 by 4s, putting each count into the variable named c and printing a message box each time. The Step value changes how Visual Basic updates the *CounterVar* each time that the loop iterates.

If you specify a negative Step value, Visual Basic counts *down*. The following loop rings the PC's speaker five times:

```
For i = 5 To 1 Step -1

 Beep

Next i
```

---

**Warning:** If you specify a negative Step value, the *EndVal* must be less than the *StartVal* or Visual Basic will execute the loop only once.

---

---

You Can Terminate Loops Early: Sometimes, you'll be processing user input or several data values using looping

statements, and an exception occurs in the data that requires an immediate termination of the loop. For example, you may be collecting sales values for a company's ten divisions inside a For loop that iterates ten times. However, the user can enter zero for a division's sales value, indicating that there is no sales data for that division. Rather than complete the loop, your program might need to quit the loop at that point because the full divisional report information can't be gathered at the time.

The Exit Do and the Exit For statements automatically terminate loops. No matter what the Do loop's relational test results in, or no matter how many more iterations are left in a For loop, when Visual Basic encounters an Exit Do or Exit For statement, Visual Basic immediately quits the loop and sends execution down to the statement following the loop.

Typically, an If statement triggers one of the Exit statements like this:

```
For Divisions = 1 To 10
' Code to get a sales value
If (sales = 0) Then
Exit For ' Quit the loop early
End If
' Process the rest of the code
Next Divisions
```

The If ensures that the Exit For executes only under one specific condition (a missing sales value). Without that specific condition triggering the Exit For, the loop cycles normally.

---

***Stop and Type:*** Listing 10.6 contains a fairly comprehensive For loop that computes compound interest for an initial investment of $1,000.00. The code appears inside the Click event procedure for a command button named cmdInt. In case you're not familiar with compound interest, each year the amount of money invested, including interest earned so far, compounds to build more money. Each time period, normally a year, means that another year's interest must be added to the value of the investment. A For loop is perfect for calculating interest. Listing 10.6 uses five compound cycles.

***Review:*** The For loop repeats a block of one or more statements of Visual Basic code. Unlike the Do loops, the For loop iterates for a specified Number of times as controlled by the For statement's control values and variables.

**Listing 10.6. Using a For loop to calculate compound interest.**

```
1: Sub cmdInt_Click ()

2: ' Use a For loop to calculate a final total

3: ' investment using compound interest.

4: ' Num is a loop control variable

5: ' IRate is the annual interest rate

6: ' Term is the Number of years in the
investment
```

```
7: ' InitInv is the investor's initial investment

8: ' Interest is the total interest paid

9: Dim IRate, Interest As Single

10: Dim Term, Num As Integer

11: Dim InitInv As Currency

12:

13: IRate = .08

14: Term = 5

15: InitInv = 1000.00


16: Interest = 1 ' Begin at one for first compound

17:

18: ' Use loop to calculate total compound amount

19: For Num = 1 To Term

20: Interest = Interest * (1 + IRate)

21: Next

22:

23: ' Now we have total interest,

24: ' calculate the total
investment

25: ' at the end of N years
```

```
26: lblFinalInv.Caption = InitInv * Interest

27: End Sub
```

***Analysis:*** This analysis focuses on the loop and not the interest calculation. The most important thing that you can do at this point is to master the For looping statement. Lines 1 through 8 contain fairly extensive remarks. The remarks contain variable descriptions so that anyone looking at the code or changing the code later will know what the variables are for.

After the program defines all the variables in lines 9 through 11, the variables are initialized with start-up values in lines 13 through 16. If you use this event procedure, be sure to add a label named lblFinalInv to a form and add a command button to the form named cmdInt. Line 15 will seem to give you trouble as you type it unless you remember Lesson 2's description of data suffix characters. Visual Basic uses the pound sign, #, to indicate double-precision values, and Visual Basic will assume that 1000.00 is a double-precision value (I don't know why) and will convert the 1000.00 to 1000# right after you press Enter at the end of the line! Don't worry about Visual Basic's pickiness here.

The most important part of this program appears on lines 19 through 21. Line 19 begins a For loop that iterates through each interest rate period (five of them), compounding the interest on top of the investment to date on line 20. Again, don't let the finance worry you. The calculation is less important than understanding the looping process. After the loop finishes, line 26 completes the event procedure by placing the compounded investment in the label. By the way, if you do implement this event procedure in your own application, the investment will appear in the window as a double-precision Number with several decimal places showing. You'll learn how to format the data into dollars and cents in lesson 7.

## Homework

### General Knowledge

1. What is a loop?
2. How many different kinds of looping statements does Visual Basic support?
3. How many different Do statements does Visual Basic support?
4. True or false: A *block* can consist of a single statement.
5. What is an infinite loop?
6. How can you utilize a loop for correcting user errors?
7. How many times does the following loop execute?
   I = 10
   Do While I > 1
   I = I - 1
   Loop
8. How many times does the following loop execute?
   I = 10
   Do While I >= 1
   I = I - 1

Loop

9. How many times does the following loop execute?
   I = 10
   Do Until I > 1
   I = I - 1
   Loop
10. How many times does the following loop execute?
    For I = 1 To 10
    Beep
    Loop
11. What Step value does Visual Basic assume if you don't specify any Step value?
12. Which statement, the Do or For statement, supports a loop that continues for a specified Number of times?
13. Which statement, the Do or For statement, supports a loop that continues according the a relational test?
14. What is the difference between a Do While and a Do Until loop?
15. What is the difference between a Do While and a Do-Loop While loop?
16. What is an *iteration*?
17. How can you force a For loop to count down rather than up?
18. If a For loop's initial starting value is greater than the ending value, what must be true for the increment value?
19. What statements terminate Do and For loops early?

## Write Code That...

1. Write a program that assigns the value of 34 to a variable and then asks the user to guess the Number using an input box. Use a Do loop to check the user's guess and keep asking for additional guesses until the user guesses the Number.

## Find the Bug

1. Larry, a fledgling programmer, wrote the following For loop that seems to be in an infinite loop. Can you spot the problem?
   For i = 1 To 25
   Total = Total * I
   i = i - 1
   Next i
2. Kim wants her For loop to loop for 100 iterations but she's having trouble. Tell Kim what's wrong with the following attempt:
   For I = 100 To 1 Step 1

## Extra Credit

A *nested loop* is a loop within a loop. The outer loop determines how many times the inner loop executes. See whether you can determine how many times the following code beeps the user.

```
For i = 1 To 5 ' The outer loop
For j = 1 To 3 ' The inner loop
Beep
Next j ' Inner loop completes before
Next i ' outer loop iterates again
```

# Visual Basic in 12 Easy Lessons



# Project 5

## Gaining Control

### Stop & Type

This lesson taught you how to document and control programs. As you increase the power of your Visual Basic applications, they become more complex. As programs get more complex, program maintenance becomes more difficult. Remarks help document summary information about the program and provide more specific help for tricky lines of code.

You can now add control to your programs by putting loops in the code, which repeat sections of the program. Computers are excellent tools for repeating calculations as many times as necessary to produce results. Loops also provide ways for you to check user input against good and bad values and to repeat the input prompting so that the user enters data in the required format and range.

In this lesson, you saw the following:

- How remarks help document your code
- When message boxes offer the user more help than label controls
- How to get feedback from input boxes
- What kind of power the loops add to your programs

## The Program's Description

Figure P5.1 shows how the PROJECT5.MAK application looks as soon as you load and run the program. The project's form contains four command buttons. The command buttons perform different tasks that demonstrate the various topics of this lesson.

**Figure P5.1. The project's opening screen.**

The program's three center command buttons perform the following actions:

- The Color Change command button quickly steps the form's background color through a series of color changes.
- The Odd Entry command button keeps requesting an odd number until the user enters an odd number
- The Even Entry command button keeps requesting an even number until the user enters an even number

## The Program's Action

Click the Color Change command button. Watch closely. The form's background changes colors extremely quickly while beeping. Although the colors change so rapidly that you cannot see all the colors, the program changes the form's background color, by means of the form's BackColor property value, sixteen times.

The beeping is accomplished through four Beep statements that do nothing more than ring the PC's bell four times between each color change. The true purpose of the beeping is to slow down the color changes. On most computers, the colors would change so fast without the intermediate beeps, that you would not see anything except a blur. Even with the beeping, the colors change almost faster than the eye can see.

The Odd Entry command button asks the user for an odd number, using the input box shown in Figure P5.2. The command button's Click event procedure keeps looping until the user enters an odd number.

**Figure P5.2. The program requests an odd number.**

The third command button, Even Entry, requests an even number and keeps looping until the user enters an even number.

## The Color Changing Routine

Listing P5.1 contains the event procedure for the Color Change command button's Click procedure.

**Listing P5.1. The color-changing event procedure.**

```
1: Sub cmdColor_Click ()

2: ' Wildly Change the color of the form

3: Dim ColorVal As Integer

4: For ColorVal = 0 To 15 ' Step
through the color values

5: frmControl.BackColor = QBColor(ColorVal)

6: Beep ' These beeps simply add

7: Beep ' to the intensity of the

8: Beep ' color change and slow

9: Beep ' things down some

10: Next ColorVal

11: End Sub
```

## Descriptions

1: The command button is named cmdColor, so the name of the Click event procedure is cmdColor_Click().

2: A remark helps explain the purpose of the event procedure.

3: An integer variable holds a For loop's color-changing value.

4: This line begins a For loop that iterates 16 times, assigning the numbers 0 through 15 to the variable

named ColorVar.

---

5: The built-in QBColor() function lets you easily change color values.

---

5: The built-in function named QBColor() requires a number in its parentheses from 0 to 15. Each number represents a different color value. Using QBColor() is easier than specifying a hexadecimal value.

6: A Beep statement slows down the color change and adds a little spice to the program.

7: A Beep statement slows down the color change and adds a little spice to the program.

8: A Beep statement slows down the color change and adds a little spice to the program.

9: A Beep statement slows down the color change and adds a little spice to the program.

10: This line terminates the For loop.

11: This line terminates the event procedure.

## Getting Odd Values

When the user clicks the Odd Entry button, the program asks for an odd number, using the InputBox$() function taught in this lesson. The InputBox$() function captures a string value. If the user clicks Cancel, the form window returns. Otherwise, the event procedure expects an odd number and keeps requesting that number using a Do loop until the user types an odd number.

**Listing P5.2. The event procedure that requests an odd number.**

```
1: Sub cmdOdd_Click ()

2: ' Request an odd number

3: Dim OddStr As String

4: Dim OddNum As Integer

5: Do

6: OddStr = InputBox$("Enter an odd number", "Get Odd")
```

```
7: If (OddStr
= "") Then ' Quit if pressed Cancel

8: Exit Do ' Quits the Do loop early

9: End If

10: OddNum = Val(OddStr) / 2

11: ' The integer OddNum holds the exact value

12: ' of the Val(OddStr) / 2 if OddNum is even

13: Loop Until (OddNum <>
(Val(OddStr) / 2))

14: End Sub
```

## Descriptions

1: The command button's Name property contains cmdOdd, so the name of the Click event procedure is cmdOdd_Click().

2: A remark explains the purpose of the procedure.

3: This line defines a string variable that will hold the result of the InputBox$() function.

4: This line defines an integer variable that will hold the result of the InputBox$() function.

5: This line begins a loop.

6: Here the program collects the user's response to the request for an odd number, and it displays an appropriate input box title. Use InputBox$() to get strings from the user.

7: If the user pressed the Cancel command button, the program exits the loop.

8: This line terminates the Do loop.

9: This line is the end of the body of the If statement.

10: This line converts the input string to a number, divides the number by two, and stores the result in an

integer variable.

11: A multiline remark describes the check for an odd number.

12: This line continues the remark.

13: OddNum holds an integer value calculated and stored in line 10. If integer OddNum variable equals the decimal calculation of the user's response divided by two, the user entered an even number, and the program needs to loop again and request the odd number once more.

14: This line terminates the event procedure.

---

**Note:** The cmdEven_Click() event procedure is identical to cmdOdd_Click() except that it checks for an even number instead of an odd number.

---

## Close the Application

You can now exit the application and exit Visual Basic. The next lesson explains more Visual Basic commands and describes how to add new controls to your form that process and display large amounts of data.

# Visual Basic in 12 Easy Lessons



# Lesson 6, Unit 11

## Arrays and Lists

### What You'll Learn

- Arrays hold data
- The advantage of arrays
- List boxes: Controls that work like arrays
- Combo boxes

Variables and controls hold the data that your Visual Basic programs process. You've now seen the different kinds of variable data types that Visual Basic supports, as well as three of the primary controls used on forms: command buttons, labels, and text boxes.

This unit extends the discussion on variables and controls by showing you a new way of managing variable data storage. By storing variables in arrays, you'll be able to process larger amounts of data with less code than you could with the kinds of variables that you've seen so far.

After you learn about the concept of arrays and how you access variable arrays, you'll discover three more controls that you can place on forms. These controls, the list box and combo box controls, give you

a simple way to display array data on forms.

# Arrays Hold Data

***Concept:*** An array is a table of values in memory. Unlike variables to which you assign different names, multiple variables stored in arrays all have the same name. You'll reference the different data values using a subscript.

All the variables that you've worked with so far require a unique variable name. As you define variables, you define both the variable data types and the variable names. Figure 11.1 shows what you have defined once Visual Basic executes the following variable definitions and assignments:

```
Dim Age As Integer

Dim Limit As Long

Dim Salary As Single

Dim DogName As String

' Store data in the variables

Age = 34

Limit = 40294

Salary = 982343.23 ' A Visual Basic programmer's pay

DogName = "Ralph"
```

**Figure 11.1. Variables appear in several different memory locations.**

Although Visual Basic *might* define and assign the variables back to back in memory, you don't know or even care where Visual Basic stores the variables. With such variable definitions, all you need are four variables of four data types, and each of those variables has a different name.

For variables that fulfill a different purpose, such variable definitions are excellent and work well for simple as well as advanced Visual Basic applications. Nevertheless, there will be times when you'll need several variables of the same data type to hold similar data values. For example, a tag agent's application might need to tally a grand total of all license fees for that day's transactions. If there are one hundred

transactions, the only way that you know to define one hundred variables is to define and name each of them differently with statements such as these:

```
Dim LicFee1, LicFee2, LicFee3, LicFee4 As Currency

Dim LicFee5, LicFee6, LicFee7, LicFee8 As Currency

Dim LicFee9, LicFee10, LicFee11, LicFee12 As Currency

' The rest of the variable definitions would follow
```

When you need to define variables that hold the same kind of data values, such as this table of 100 tag agent license values, your program gets burdened down with all those variable names. Filling those variables is a problem as well. Will you supply one hundred text box controls on a form? If the clerk at the counter enters each customer's license amount as the customer pays, how can you collect the next license fee in the next variable?

Even if you could find a way to fill the one hundred variables with the daily values, how would you get a total of that data? You would have to write multiple addition statements that begin something like this:

```
DayTotal = LicFee1 + LicFee2 + LicFee3 + LicFee4

DayTotal =
DayTotal + LicFee5 + LicFee6 + LicFee7

DayTotal = DayTotal + LicFee8 + LicFee9 + LicFee10

' Ugh! This could take forever!
```

Again, the variables that you've worked with so far have been great for applications that don't need to process several sets of related data. Computers are used for processing sets of data, however. You may recall from the previous lesson that the power of a computer comes mostly from its capability to loop through code at a high rate of speed, processing large amounts of data quickly and without getting bored. Those large amounts of data *must* be stored in something other than the stand-alone variables that you've seen so far or the programmer would soon get bogged down with writing huge numbers of variable names every time another calculation was needed.

---

*Definition:* An *array* is a list of values.

---

Visual Basic supports the use of arrays to eliminate much of the tedium that you would face if every variable had to have a different name. Rather than define single stand-alone variables using several Dim statements, you can define an entire list of variables. This list, called an array, takes on the following attributes:

- Each item in the array is called an *element*.
- Every element in the array must be the same data type.
- The list has one name and each item in that list is another occurrence of that name.
- You'll use a *subscript number* to uniquely reference individual elements. The subscript number is often just called the array's *subscript*.
- Instead of using Dim, you'll often use the Static statement to define arrays.

---

**Note:** There are two additional ways to declare arrays, using Dim and the Global statement, that you'll learn about in Lesson 8.

---

Figure 11.1's variables could not be stored in an array because their data types are different. The tag agent's one hundred license fees, however, would make excellent array candidates because each fee requires the same data type (Currency).

You'll use the Static statement for this lesson to define arrays. Static works a lot like Dim except that Static defines arrays rather than single stand-alone variables. Here is the format of Dim:

```
Static ArName(subMax) As DataType
```

The *ArName* is the name that you want to call the array. *subMax* must be a number that describes the total number of array elements you need. The *DataType* is a data type that matches one of Visual Basic's data types, such as Single and Long.

---

**Warning:** As with any kind of variable, you'll have to define arrays with Static before you can use them.

---

***Stop and Type:*** Here is the simple way that the tag agent might define the 100 license fee variables:

```
Dim LicFee(100) As Currency ' Defines 100 elements
```

***Review:*** By defining an array of data, you can define multiple occurrences of variable data without having to assign each variable a different name and without having to define each variable separately.

The array will have a single name and be stored together, with all of the array values being sequentially back to back, in memory. You'll access each value using a unique subscript.

***Output:*** Figure 11.2 shows how the LicFee appears in memory.

**Figure 11.2. Array elements appear back to back and each has a unique subscript that begins at 0.**

***Analysis:*** Visual Basic automatically defines an extra array element with the subscript of 0 when you define an array. The Option Base 1 statement that you can put in the (general) procedure of any form's Code window tells Visual Basic to define arrays with a starting subscript of 1. Without Option Base 1, however, when you request 100 elements, Visual Basic throws in the extra zero subscript. Most Visual Basic programmers choose to ignore the zero subscript, however, and if you do, too, you won't be wasting too much memory by ignoring the extra array element. This book will always ignore the zero-based subscript and work with a starting subscript of 1. If you want to ignore the zero subscript, you can do that without coding the Option Base 1 statement.

Visual Basic assigns each element in the array a subscript so that your program can distinguish between each value in the array. As with all variables, Visual Basic assigns zero to each array element, and your program will overwrite those initial zeros when the program assigns data to the array.

---

**Warning:** Don't ever attempt to reference a subscript that doesn't exist for a particular array. Visual Basic would issue an error message if your program attempted to do something with LicFee(311) or LicFee(-8) because 311 and -8 are out of the range of LicFee's subscripts that range only from 0 to 100.

---

## The Advantage of Arrays

***Concept:*** Arrays eliminate much tedious coding. By using the subscript rather than a different variable name, you can use a For loop to step through arrays and access individual items in sequence.

Arrays simplify the process of initializing, printing, calculating, and storing data. If, for example, you were to define an array of 20 string variables that will hold customer names like this:

```
Dim CustNames(20) As
String
```

you *could* initialize each of those 20 strings with 20 InputBox$() function assignments, like this:

```
CustName(1) = InputBox$("What is the next customer's name?")
```

```
CustName(2) =
InputBox$("What is the next customer's name?")

CustName(3) = InputBox$("What is the next customer's name?")

CustName(4) = InputBox$("What is the next customer's name?")

CustName(5) = InputBox$("What is the next
customer's name?")

CustName(6) = InputBox$("What is the next customer's name?")

CustName(7) = InputBox$("What is the next customer's name?")

CustName(8) = InputBox$("What is the next customer's name?")

CustName(9) =
InputBox$("What is the next customer's name?")

' And so on for all 20 names
```

Wouldn't the following For loop be easier, though?

```
For Ctr = 1 To 20

 CustName(Ctr) = InputBox$("What is the
next customer's name?")

Next Ctr
```

The variable, Ctr, steps through the values from 1 to 20, assigning the result of the InputBox$() function for each of those 20 times. The following similar code could display the 20 names in message boxes:

```
For Ctr = 1 To 20
```

```
MsgBox CustName(Ctr)

Next Ctr
```

Of course, the constant display of input and message boxes can get tedious, but focus for now on the programming ease that arrays provide over variables that each have different names. Using a For loop for stepping through an array of values makes working with that array a breeze.

---

**Note:** Of course, you don't have to step through every array value when you need to work with only a few of them. For example, you could display only the 5th and 19th customer with two back-to-back message boxes like this:
MsgBox CustName(5)
MsgBox CustName(19)
An array gives you another way to reference data than by using a different variable name. Each element, however, is a unique variable known by its array name and subscript. Therefore, you can work with individual array elements as if they were stand-alone variables by accessing their individual subscripts.

---

***Stop and Type:*** Listing 11.1 contains a calculating section of code from a procedure that finds the average of the license fee transactions. The array named LicFee had already been filled with values earlier in the procedure. Those values may have come from the form, from the user with input boxes, or from a disk file.

***Review:*** The array subscripts enable you to access one or more elements from the array. The big advantage that arrays provide is enabling your program to step through the entire array using a For loop's control variable as the subscript.

**Listing 11.1. Computing an average license fee from an array.**

```
1: TotalFee = 0.0 ' TotalFee is defined as Currency

2: AvgFee = 0.0 ' AvgFee is defined as a Currency

3: For Ctr = 1 To
100 ' There are 100 license fees

4: TotalFee = TotalFee + LicFee(Ctr)

5: Next Ctr

6: ' Now that the total is computed, calculate average
```

```
7: AvgFee = TotalFee / 100.0
```

***Analysis:*** Before studying Listing 11.1, think about how you compute the average of numbers. You first add all the numbers together, then divide by the total number to get an average. That process is exactly what this code does. The For loop in lines 3 through 5 moves the values from 1 to 100 into the variable named Ctr during each loop iteration. Line 4 adds each of the array values, one at a time as controlled by the For loop, to the TotalFee variable.

After all one hundred license fees are added together, line 7 then computes the average of the one hundred values. Subsequent code could either display the average in a label or use the value for another calculation.

## List Boxes: Controls That Work Like Arrays

***Concept:*** The list box control works a lot like an active array on your form that the user can scroll through, seeing all the items in the list. Often, programmers initialize list boxes with data from arrays. Unlike most controls, you can't add values to a list box control through the Property window, but must add the values at runtime through code.

Figure 11.3 shows the location of the list box control on the Toolbox window. The list box control displays values through which the user can scroll. The list box doesn't display scroll bars if the size of the list box is large enough to display all the values. Remember that list boxes are for displaying data; the user cannot add data directly to a list box.

**Figure 11.3. The location of the list box control.**

---

**Tip:** To remind yourself how the list box control works, you may want to load and run the CONTROLS.MAK project once again and scroll through the values in the list box control in the application.

---

Table 11.1 contains the list of property values that you can set for list box controls. You've seen many of the properties before because several controls share many of the same properties.

**Table 11.1. The list box properties.**

| *Property* | *Description* |
|---|---|
| BackColor | The background color of the list box. It's a hexadecimal number representing one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you're ready to set the BackColor property. The default background color is the same as the form's default background color. |

| | |
|---|---|
| Columns | If 0 (the default), the list box scrolls vertically in a single column. If 1 or more, the list box items appear in the number of columns specified (one or more columns) which the user scrolls the list box horizontally to see all the items if needed. Figure 11.4 shows two identical list boxes, one with a Columns property of 0 and one with a Columns property of 3. |
| DragIcon | The icon that appears when the user drags the list box control around on the form. (You'll only rarely allow the user to move a list box control, so the Drag... property settings aren't usually relevant.) |
| DragMode | Either contains 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the list box control but that you, through code, can initiate the dragging if needed. |
| Enabled | If set to True (the default), the list box control can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | True (the default) if the list values are to display in boldfaced characters; False otherwise. |
| FontItalic | True (the default) if the list values are to display in italicized characters; False otherwise. |
| FontName | The name of the list box's text style. Typically, you'll use the name of a Windows TrueType font. |
| FontSize | The size, in points, of the font used for the list box values. |
| FontStrikethru | True (the default) if the list values are to display in strikethru letters (characters with a dash through each one); False otherwise. |
| FontUnderline | True (the default) if the list box values are to display in underlined letters; False otherwise. |
| ForeColor | The color of the values inside the list box. |
| Height | The height, in twips, of the list box control. |
| HelpContextID | If you add advanced, context-sensitive help to your application), the HelpContextID provides the identifying number for the help text. |
| Index | If the list box control is part of a control array, the Index property provides the numeric subscript for each particular list box control (see the next unit). |
| Left | The number of twips from the left edge of the Form window to the left edge of the list box control. |
| MousePointer | The shape that the mouse cursor changes to if the user moves the mouse cursor over the list box control. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take. (See Lesson 12.) |
| MultiSelect | If 0-None (the default), the user can select only one list box item. If 1-Simple, the user can select more than one item by clicking with the mouse or by pressing the spacebar over items in the list. If 2-Extended, the user can select multiple items using Shift+click and Shift+arrow to extend the selection from a previously selected item to the current item. Ctrl+click either selects or deselects an item from the list. |

| | |
|---|---|
| Name | The name of the control. By default, Visual Basic generates the names List1, List2, and so on as you add subsequent list box controls to the form. |
| Sorted | If True, Visual Basic doesn't display the list box values sorted numerically or alphabetically. If False (the default), the values appear in the same order in which the program added them to the list. |
| TabIndex | The focus tab order begins at 0 and increments every time that you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, the user can press Tab to move the focus to this list box. If False, the list box can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the list box control. |
| Top | The number of twips from the top edge of a list box control to the top of the form. |
| Visible | True or False, indicating whether or not the user can see (and, therefore, use) the list box control. |
| Width | The number of twips wide that the list box control consumes. |

**Figure 11.4. Two identical list boxes with different Columns property settings.**

When placing a list box control on the form, decide how tall you want the list box to be by resizing the control to the size that fits the form best. Remember that if all the list box values don't all fit within the list box, Visual Basic adds scroll bars to the list box so that the user can scroll through the values.

Table 11.2 contains all the list box events that you can program. Table 11.2 contains all the list box events that you can use in a program. You'll rarely write event procedures for list box controls, however. Most of the time, you'll let the user scroll through the list box values to see information they need; programs don't need to respond to list box events as often as they need to respond to command buttons and text boxes.

**Table 11.2. The list box control's events.**

| *Event* | *Description* |
|---|---|
| Click | Occurs when the user clicks the list box control |
| DblClick | Occurs when the user double-clicks the list box control |
| DragDrop | Occurs when a dragging operation of the list box completes |
| DragOver | Occurs during a drag operation |
| GotFocus | Occurs when the list box receives the focus |
| KeyDown | Occurs when the user presses a key as long as the KeyPreview property is set to True for the controls on the form; otherwise, the form gets the KeyDown event |
| KeyPress | Occurs when the user presses a key over the list box |
| KeyUp | Occurs when the user releases a key over the list box |
| LostFocus | Occurs when the list box loses the focus to another object |

| | |
|---|---|
| MouseDown | Occurs when the user presses a mouse button over the list box |
| MouseMove | Occurs when the user moves the mouse over the list box |
| MouseUp | Occurs when the user releases a mouse button over the list box |

Table 11.3 contains a list of list box control methods that you'll need to use for initializing, analyzing, and removing items from a list box control. Methods works like miniature programs that operate on controls. Here is the format of a method's use on a list box named lstItems:

```
lstItems.AddItem "Arizona"
```

The control name always precedes the method and the dot operator. Any data needed by the method appears to the right of the method.

**Table 11.3. List box methods.**

| *Method Name* | *Description* |
|---|---|
| AddItem | Adds a single item to the list box |
| Clear | Clears all items from the list |
| List | A string array that holds each item within the list box |
| ListCount | The total number of items in a list box |
| RemoveItem | Removes a single item from a list box |
| Selected | Determines whether the user has selected a particular item in the list box |

Use the AddItem method to add properties to a list box control. Suppose that you want to add a few state names to a list box named lstStates. The following code adds the state names:

```
'
Add several states to a list box control

lstStates.AddItem "Arizona"

lstStates.AddItem "Alabama"

lstStates.AddItem "Oklahoma"

lstStates.AddItem "New York"
```

```
lstStates.AddItem "California"


lstStates.AddItem "Nebraska"

lstStates.AddItem "Ohio"

lstStates.AddItem "Florida"

lstStates.AddItem "Texas"

lstStates.AddItem "South Dakota"

lstStates.AddItem "Nevada"

lstStates.AddItem
"Illinois"

lstStates.AddItem "New Mexico"
```

This code would most likely appear in the Form_Load() event procedure so that the list boxes are initialized with their values before the form appears and before the list boxes are seen on the form.

---

**Note:** The list box acts a little like an array.

---

Each item in a list box has a subscript just as each element in an array has a subscript. The first item that you add to a list box has a subscript of 0, the second has a subscript of 1, and so on. To remove the third item from the list box, therefore, your code can apply the RemoveItem method, as follows:

```
lstStates.RemoveItem(2) ' 3rd item has a subscript of 2
```

---

**Caution:** Keep in mind that, as you remove items from a list box, the remaining item subscripts adjust accordingly. Therefore, if a list box contains seven items, each item has a subscript that ranges from 0 to 6. If you remove the fourth item, the list box items will then range from 0 to 5; the subscript of 5 will indicate the same item that the subscript of 6 indicated before the RemoveItem method removed the fourth item.

If you want to remove all items from the list box, use the Clear method. The following simple method removes all the state names from the state list box:

```
lstStates.Clear ' Remove all items
```

You can assign individual items from a list box control that contains data by using the List method. You must save list box values in string or variant variables unless you convert list box items to a numeric data type using Val() first. The following assignment statements store the first and fourth list box item in two string variables:

```
FirstStringVar = lstStates.List(0)

SecondStringVar = lstStates.List(3)
```

The ListCount method always provides the total number of items in the list box control. For example, the following statement stores the number of list box items in a numeric variable named Num:

```
Num = lstStates.ListCount
```

The Selected method returns either a true or false value that determines whether a user has selected a list box item. The Selected method returns true for possibly more than one list box item if the MultiSelect property is set to either 1-Simple or 2-Extended. Those properties indicate that the user can select more than one item at once. Figure 11.5 shows a list box with several items selected at the same time.

**Figure 11.5. A list box with a MultiSelect property set to 1 or 2.**

***Stop and Type:*** The code in Listing 11.2 stores every selected item of a list box named lstStates in a string array. The string array is defined with enough elements to hold all the items if the user happens to have selected all fifty items that were first added to the list box.

***Review:*** A list box control holds one or more values through which the user can scroll. Unlike many other controls, the user can only look at and select items in the list box, but not add items to the list box. Through methods, you can, however, add items, delete items, and check for selected items in the list box by using the methods inside your code procedures.

**Listing 11.2. Storing all selected values in a string array.**

```
1: Dim SelectStates(50) As String
```

```
2: Dim StSub As Integer ' State array subscript

3: StSub = 1

4: For Ctr = 0 To 49 ' 50 states in all

5: If (lstStates.Selected(Ctr) =
True) Then

6: SelectStates(StSub) = lstStates.List(Ctr)

7: StSub = StSub + 1

8: End If

10: Next Ctr
```

***Analysis:*** Line 2 must define a subscript that will keep track of the state's string array. Line 3 initializes the starting subscript to 1 because the zero subscript is ignored throughout this book's programs; this is the standard followed by most Visual Basic programmers.

Lines 4 through 10 contain a loop that steps through all fifty states that were stored in the list box earlier (perhaps in the Form_Load() event procedure). If a state has been selected by the user (line 5 checks for the selection), the Selected method returns a true result and line 6 stores the selected list box item in the string array. Line 7 increments the string array's subscript to prepare the state array subscript for further possible assignments in subsequent iterations through the loop.

## Combo Boxes

***Concept:*** Combo boxes work a little like list boxes except that the user can add items to a combo box at runtime. There are three kinds of combo boxes determined by the Style property. The AddItem and RemoveItem methods are popular for combo boxes, although all of the list box methods that you learned in the previous lesson apply to combo boxes as well.

Figure 11.6 shows the location of the combo box control on the Toolbox window. No matter what kind of combo box you want to place on the form, you'll use the same combo box control on the toolbox to add the combo box to the form.

**Figure 11.6. The location of the combo box control.**

There are three kinds of combo boxes, as follows:

- A *dropdown combo box* takes up only a single line on the form unless the user opens the combo

box (by pressing the combo box's down arrow) to see additional values. The user can enter additional items at the top of the dropdown combo box and select items from the combo box.

- A *simple combo box* always displays items as if they were in a list box. The user can add items to the combo box list as well.
- A *dropdown list box* is a special list box that the user can't enter new items into, but that normally appears closed to a single line until the user clicks the down arrow button to open the list box to its full size. Technically, dropdown list boxes are not combo box controls but work more like list boxes. The reason dropdown list boxes fall inside the combo box control family is that you place dropdown list boxes on forms by clicking the combo box control and setting the appropriate combo box property (Style).

Figure 11.7 shows the three kinds of combo boxes. Each combo box contains the names of states that you saw earlier in list boxes. The first combo box, the dropdown combo box, is normally closed; when the user clicks the combo box's down arrow, the combo box opens. The third combo box, the dropdown list box, is left unopened. If the user opens the dropdown list box, the user will see a list of state names but will not be able to add to the state names because no data entry is possible in dropdown list boxes.

**Figure 11.7. The three combo box styles.**

Table 11.4 contains a description of every combo list property value that you can set in the Property window.

**Table 11.4. The combo box properties.**

| *Property* | *Description* |
|---|---|
| BackColor | The background color of the combo box. This is a hexadecimal number representing one of thousands of possible Windows color values. You can select from a palette of colors displayed by Visual Basic when you're ready to set the BackColor property. The default background color is the same as the form's default background color. |
| DragIcon | The icon that appears when the user drags the combo box control around on the form. (You will only rarely allow the user to move a combo box control, so the Drag... property settings aren't usually relevant.) |
| DragMode | Either contains 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the combo box control but that you, through code, can initiate the dragging if needed. |
| Enabled | If set to True (the default), the combo box control can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | True (the default) if the combo values are to display in boldfaced characters; False otherwise. |
| FontItalic | True (the default) if the combo values are to display in italicized characters; False otherwise. |
| FontName | The name of the combo box's text style. Typically, you'll use the name of a Windows TrueType font. |

| | |
|---|---|
| FontSize | The size, in points, of the font used for the combo box values. |
| FontStrikethru | True (the default) if the combo values are to display in strikethru letters (characters with a dash through each one); False otherwise. |
| FontUnderline | True (the default) if the combo box values are to display in underlined letters; False otherwise. |
| ForeColor | The color of the values inside the combo box. |
| Height | The height, in twips, of the combo box control. |
| HelpContextID | If you add advanced, context-sensitive help to your application), the HelpContextID provides the identifying number for the help text. |
| Index | If the combo box control is part of a control array, the Index property provides the numeric subscript for each particular combo box control. (See the next unit). |
| Left | The number of twips from the left edge of the Form window to the left edge of the combo box control. |
| MousePointer | The shape that the mouse cursor changes to if the user moves the mouse cursor over the combo box control. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take. (See Lesson 12.) |
| Name | The name of the control. By default, Visual Basic generates the names Combo1, Combo2, and so on as you add subsequent combo box controls to the form. |
| Sorted | If True, Visual Basic doesn't display the combo box values sorted numerically or alphabetical. If False (the default), the values appear in the same order in which the program added them to the list. |
| Style | The default, 0-Dropdown Combo, produces a dropdown combo box control. 1-Simple Combo turns the combo box into a simple combo box control. 2-Dropdown list turns the combo box into a dropdown list box control. |
| TabIndex | The focus tab order begins at 0 and increments every time that you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, the user can press Tab to move the focus to this combo box. If False, the combo box can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the combo box control. |
| Text | The initial value only that the user sees in the combo box. |
| Top | The number of twips from the top edge of a combo box control to the top of the form. |
| Visible | True (the default) or False, indicating whether the user can see (and, therefore, use) the combo box control. |
| Width | The number of twips wide that the combo box control consumes. |

---

**Note:** Notice that there is no MultiSelect combo box property as there is with list box controls. The user can select only one combo box item at any one time.

---

Table 11.5 contains a list of the combo box events for which you can write matching event procedures when your program must react to a user's manipulation of a combo box.

**Table 11.5. The combo box control's events.**

| *Event* | *Description* |
|---|---|
| Change | Occurs when the user changes the value in the data entry portion of the dropdown combo or the simple combo box; not available for dropdown list boxes because the user can't change data in them |
| Click | Occurs when the user clicks the combo box control |
| DblClick | Occurs when the user double-clicks the combo box control |
| DragDrop | Occurs when a dragging operation of the combo box completes |
| DragOver | Occurs during a drag operation |
| DropDown | Occurs when the user opens a dropdown combo box or a dropdown list box |
| GotFocus | Occurs when the combo box receives the focus |
| KeyDown | Occurs when the user presses a key as long as the KeyPreview property is set to True for the controls on the form; otherwise, the form gets the KeyDown event |
| KeyPress | Occurs when the user presses a key over the combo box |
| KeyUp | Occurs when the user releases a key over the combo box |
| LostFocus | Occurs when the combo box loses the focus to another object |

The combo box controls support the same methods that the list box controls support. Therefore, you can add, remove, count, and select items from the combo box if you apply the methods seen in Table 11.3.

***Stop and Type:*** Listing 11.3 contains a command button's event procedure that adds a value to a combo box's list of items. Unlike text box controls, you'll need to provide the user with some data entry mechanism, such as a command button, that informs the program when to use the AddItem method to add an item to a combo box.

---

**Note:** Listing 11.3 assumes that another procedure, such as the Form_Load() procedure, added the initial values to the combo box.

---

***Review:*** The three kinds of combo boxes offer three similar controls with which your program can interact by showing lists of values and, optionally, allowing the user to enter values in the lists. Two of the combo boxes are dropdown lists that don't consume form space until the user opens the combo boxes.

**Listing 11.3. A command button's event procedure that adds an item to a combo box.**

```
1: Sub cmdSimple_Click ()
```

```
2: ' Add the user's item to the simple combo

3: comSCtrl.AddItem comSCtrl.Text

4: comSCtrl.Text = ""

5: comSCtrl.SetFocus

6: End Sub
```

***Output:*** Figure 11.8 contains a screen that you saw in the second lesson of the book. The CONTROLS.MAK application demonstrates the simple combo box and shows how you can set up an application to add items to a list of values.

**Figure 11.8. Adding data to a simple combo box control.**

***Analysis:*** The command button in Figure 11.8 is named cmdSimple, so clicking the command button executes the event procedure shown in Listing 11.3. Line 3 stores the combo box's Text property value to that combo box's list of items. The combo box *will not* contain a user's entry in the upper data entry portion of the combo box until an AddItem method adds that entry to the list. The Text property always holds the current value shown in the data entry portion of the combo box, but the AddItem method must add that value to the list.

As soon as the user's entry is added, line 4 erases the data entry portion of the combo box. After all, the user's text will now appear in the lower listing portion of the combo box (thanks to line 3), so line 4 clears the data entry area for more input. In addition, line 5 sets the focus back to the combo box (the focus appears in the data entry area that line 4 cleared) so that the user is ready to add an additional item to the combo box.

## Homework

### General Knowledge

1. How can arrays help you process large amounts of data?
2. What is an array element?
3. What is an array subscript?
4. If an array contains 20 elements, how many names does the array have?
5. If an array contains 20 elements, how many subscripts does the array have (assume that there is an Option Base 1 statement in the (general) procedure section of the Code window).
6. True or false: Every element in an array must be the same data type.

7. How can you differentiate between array elements?
8. What statement defines arrays inside event procedures?
9. Assuming that there is an Option Base 1 statement in the (general) procedure section of the Code window, what is the starting subscript for all arrays that you subsequently define in the program?
10. Which loop works best for array processing?
11. Which controls work like arrays?
12. True or false: The user can enter new values in list boxes.
13. True or false: The user can select more than one value in a list box.
14. True or false: A list box may consist of more than one column.
15. Why are methods important for list box and combo box controls?
16. Which event procedure do programmers often use to initialize list and combo box controls?
17. True or false: There are three kinds of combo boxes.
18. True or false: There are three kinds of combo box controls on the toolbox.
19. True or false: The list box methods work well for the combo boxes also.
20. True or false: List and combo box controls have subscripts just as arrays have.
21. What is the method that indicates whether a list box item is selected?
22. True or false: A simple combo box can be opened with a user's click of the simple combo box's down arrow.

## Write Code That...

1. Paul wants to ensure that all items in his list box are sorted. Which property should Paul set to 1-Sorted?
2. Suppose that you want to define an array of three high school student ages in a list. Write the Static statement that defines the ages.
3. What list box property would you set to 1 or 2 if you wanted to allow the user to select from multiple list box items at one time?
4. Mary wants to erase all list box items when the user clicks a special command button named cmdGoAway. Write the event procedure needed to erase the list box named lstVals if the user clicks the cmdGoAway command button.
5. Why should you offer a command button for the user to use when adding items to a combo box control?

### Find the Bug

1. Carla needs to define a list box with 24 items in it. In the Form_Load() event procedure, Carla attempts this:
   Static lstData(24) ' Define 24 list boxes
   Tell Carla what she's doing incorrectly.

## Extra Credit

Write an application that asks the user, with an InputBox() function, whether he or she wants to see a dropdown combo box, a simple combo box, or a dropdown list box. Ask the user, in the InputBox() prompt string, to type a 1, 2, or 3 to indicate which of the three kinds of combo boxes are requested. Use a loop to check that the user enters a proper value, and if the user enters a value other than 1, 2, or 3, keep prompting until the user enters one of the three values. Then display a list of ten ice cream flavors in that particular combo box control. Hint: Assign the user's entry to the combo box's Style property.

# Visual Basic in 12 Easy Lessons

---



---

# Lesson 6, Unit 12

## Checks, Options, and Control Arrays

### What You'll Learn

- Option buttons offer choices
- Check boxes offer more choices
- Option buttons come in multiple sets
- Control arrays simplify programming

You'll find that this unit is easy because it introduces two new controls that you've seen in many Windows programs, and it also builds upon arrays that you mastered in the previous unit. The check box and option button controls described here provide the user with choices of data values and options that the user can select. Unlike list and combo box controls, the check box and option button controls (shown in Figure 12.1's Toolbox window) are perfect controls for offering the user a limited number of choices.

**Figure 12.1. The check box and option button controls.**

# Option Buttons Offer Choices

***Concept:*** The option button control gives the user the ability to select one item from a list of several items that you display in a series of option button choices. Visual Basic keeps the user from selecting more than one option button at a time. The option button's events offer several ways for you to manage the option button selections.

List boxes and combo boxes are perfect controls for displaying scrolling lists of items from which the user can select and, in the case of combo boxes, add to. Unlike list and combo boxes, option buttons are good to use when you must offer the user a list of fixed choices that your program knows ahead of time.

---

**Tip:** Think of option buttons as a multiple-choice selection from which the user can choose one item.

---

Option buttons are sometimes called *radio buttons*. Perhaps you're old enough to remember pre-digital car radios that had five or six buttons sticking out with preset statements assigned to each button. At any one time, you could depress one button because the radio could play only one station at a time. When you pressed a button, any other button pressed inward immediately clicked out. The buttons were designed so that only one button at a time could be chosen.

Take the time to load CONTROLS.MAK and press the Next control command button until you see the option buttons shown in Figure 12.2. Try to click more than one option button and you'll see that Visual Basic keeps you from doing so.

**Figure 12.2. You may select only one option button at a time.**

Table 12.1 lists the property settings for the option button controls that you can set from within the Property window when you place option buttons on the form. There are several properties that you've seen before on other kinds of controls.

**Table 12.1. The option button properties.**

| *Property* | *Description* |
|---|---|
| Alignment | Either 0 for left justification (the default) or 1 for right justification of the option button's caption. If you choose to left justify, the option button appears to the left of the caption. If you choose to right justify, the option button appears to the right of the caption. |
| BackColor | The background color of the option button. This is a hexadecimal number representing one of thousands of possible Windows color values. You'll be able to select from a palette of colors displayed by Visual Basic when you're ready to set the BackColor property. The default background color is the same as the form's default background color. |

| | |
|---|---|
| Caption | The text that appears in an option button. If you precede any character in the text with an ampersand, &, that character then acts as the option button's access key. |
| DragIcon | The icon that appears when the user drags the option button around on the form. (You'll only rarely allow the user to move an option button, so the Drag... property settings aren't usually relevant.) |
| DragMode | Either contains 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the option button but that you, through code, can initiate the dragging if needed. |
| Enabled | If set to True (the default), the option button can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | True (the default) if the Caption is to display in boldfaced characters; False otherwise. |
| FontItalic | True (the default) if the caption is to display in italicized characters; False otherwise. |
| FontName | The name of the option button caption's style. Typically, you'll use the name of a Windows TrueType font. |
| FontSize | The size, in points, of the font used for the command button's caption. |
| FontStrikethru | True (the default) if the caption is to display in strikethru letters (characters with a dash through each one); False otherwise. |
| FontUnderline | True (the default) if the caption is to display in underlined letters; False otherwise. |
| ForeColor | The hexadecimal color code of the caption text's color. |
| Height | The height, in twips, of the option button. |
| HelpContextID | If you add advanced context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| Index | If the option button is part of a control array, the Index property provides the numeric subscript for each particular option button. |
| Left | The number of twips from the left edge of the Form window to the left edge of the option button. |
| MousePointer | The shape that the mouse cursor changes to if the user moves the mouse cursor over the option button. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take. (See Lesson 12.) |
| Name | The name of the control. By default, Visual Basic generates the names Option1, Option2, and so on as you add subsequent option buttons to the form. |
| TabIndex | The focus tab order begins at 0 and increments every time that you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, the user can press Tab to move the focus to this option button. If False, the option button can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the option button. |
| Top | The number of twips from the top edge of an option button to the top of the form. |
| Value | Either True or False (the default) indicating whether the option button is selected. |

| | |
|---|---|
| Visible | True or False, indicating whether the user can see (and, therefore, use) the option button. |
| Width | The number of twips wide that the option button consumes. |

Table 12.2 contains a list of the option button's events that trigger event procedures you can write. Generally, the Click event procedure is the most commonly used so that an application can perform a specific action when the user selects a particular option button.

**Table 12.2. The option button's events.**

| *Event* | *Description* |
|---|---|
| Click | Occurs when the user clicks the option button with the mouse |
| DblClick | Occurs when the user double-clicks the mouse over the option button |
| DragDrop | Occurs when a dragging operation over the option button completes |
| DragOver | Occurs during a drag operation |
| GotFocus | Occurs when the option button receives the focus |
| KeyDown | Occurs when the user presses a key as long as the KeyPreview property is set to True for the controls on the option button; otherwise, the control gets the KeyDown event |
| KeyPress | Occurs when the user presses a key over the option button |
| KeyUp | Occurs when the user releases a key |
| LostFocus | Occurs when the option button loses the focus |

***Stop and Type:*** Load and run this book's project named TRANSL.MAK. Listing 12.1 contains the code for the project's event procedures. The program allows the user to choose the target language into which "Good Morning" is translated. There is room on the form for only one translation, so the option buttons ensure that the user can select one translation at most.

***Review:*** The option button controls enable you to add choices to the form. The user will be allowed to select from at most one choice by clicking (or tabbing to change the focus and pressing Enter) the option button of the desired option. By adding code to the option buttons' Click event procedures, you can respond to the user's choice.

**Listing 12.1. The Form_Load() and four option button Click() procedures.**

```
1: Sub
Form_Load ()

2: ' Make sure that all option buttons are

3: ' False when first displaying the form
```

```
4: optFre.Value = 0

5: optIta.Value = 0

6: optSpa.Value = 0

7: optPig.Value = 0

8: End Sub

9:

10: Sub cmdExit_Click ()

11: End

12: End Sub

13:


14: Sub optFre_Click ()

15: ' Send the French message

16: lblTrans.Caption = "Bonjour"

17: End Sub

18:

19: Sub optIta_Click ()

20: ' Send the Italian message

21: lblTrans.Caption = "Buon giorno"
```

```
22: End Sub

23:

24: Sub
optPig_Click ()

25: ' Send the Pig latin message

26: lblTrans.Caption = "oodGa ayDa"

27: End Sub

28:

29: Sub optSpa_Click ()

30: ' Send the Spanish message

31: lblTrans.Caption = "Buenos dìas"

32: End Sub
```

***Output:*** Figure 12.3 shows the TRANSL.MAK screen just after the user selects the Italian option button. The Italian translation appears in the shaded label at the bottom of the screen.

**Figure 12.3. The user wants to *parla Italiano bene!***

***Analysis:*** Listing 12.1 looks a little different from the other listings that you've seen so far in this book. The listing contains more than one event procedure. As you've gathered by now, a Visual Basic program consists of lots of event procedures and possibly a (general) procedure. In Lesson 8, you'll learn about other kinds of procedures that a Visual Basic program may contain.

Although the default Value property for all option buttons is 0, meaning unselected, Visual Basic automatically selects an option button (the one with the lowest TabIndex property value) when loading the application. Therefore, lines 4 through 7 in Listing 12.1 set all four option button Value properties to 0 upon loading the form so that there is no option button set until the user selects one. (The four option buttons are named optFre, optIta, optPig, and optSpa.)

Lines 10 through 12 provide the familiar Exit command button event procedure that terminates the program when the user presses the Exit button.

Lines 14 through 32 complete the code with the four option button Click event procedures. The user triggers one of the four procedures by selecting one of the four option buttons. The shaded translation label at the bottom of the screen that will hold the translation is named lblTrans. The four option button Click event procedures change the Caption property of the lblTrans object to the appropriate translation that matches the option button's caption.

## Check Boxes Offer More Choices

***Concept:*** The check box control works a lot like the option button. You'll place several check boxes on a form; each check box represents a user-selected choice. Unlike option buttons, the user can select more than one check box item at a time.

Figure 12.4 contains the check box controls found in this book's CONTROLS.MAK application. If you load and run the program, the check boxes are the fourth set of controls on which you'll click. As you can see from Figure 12.4, when the user selects a check box, the check box is marked with an X, indicating a true selection. Two of the three check boxes are selected in Figure 12.4.

**Figure 12.4. The check box controls allow for multiple selections.**

---

**Note:** Although the check box controls in Figure 12.4 don't contain access keystrokes, you can add access keystroke selection to check boxes just as you can for option button controls.

---

Table 12.3 contains the property values available for check box controls. Most of the properties are equivalent to the option button properties.

**Table 12.3. The check box properties.**

| *Property* | *Description* |
|---|---|
| Alignment | Either 0 for left justification (the default) or 1 for right justification of the check box's caption. If you choose to left justify, the check box appears to the left of the caption. If you choose to right justify, the check box appears to the right of the caption. |
| BackColor | The background color of the check box. This is a hexadecimal number representing one of thousands of possible Windows color values. You'll be able to select from a palette of colors displayed by Visual Basic when you're ready to set the BackColor property. The default background color is the same as the form's default background color. |
| Caption | The text that appears in a check box. If you precede any character in the text with an ampersand, &, that character then acts as the check box's access key. |
| DragIcon | The icon that appears when the user drags the check box around on the form. (You'll only rarely allow the user too move a check box, so the Drag... property settings aren't usually relevant.) |

| | |
|---|---|
| DragMode | Either contains 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the check box but that you, through code, can initiate the dragging if needed. |
| Enabled | If set to True (the default), the check box can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | True (the default) if the Caption is to display in boldfaced characters; False otherwise. |
| FontItalic | True (the default) if the caption is to display in italicized characters; False otherwise. |
| FontName | The name of the check box caption's style. Typically, you'll use the name of a Windows TrueType font. |
| FontSize | The size, in points, of the font used for the command button's caption. |
| FontStrikethru | True (the default) if the caption is to display in strikethru letters (characters with a dash through each one); False otherwise. |
| FontUnderline | True (the default) if the caption is to display in underlined letters; False otherwise. |
| ForeColor | The hexadecimal color code of the caption text's color. |
| Height | The height, in twips, of the check box. |
| HelpContextID | If you add advanced, context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| Index | If the check box is part of a control array, the Index property provides the numeric subscript for each particular check box. |
| Left | The number of twips from the left edge of the Form window to the left edge of the check box. |
| MousePointer | The shape that the mouse cursor changes to if the user moves the mouse cursor over the check box. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take. (See Lesson 12.) |
| Name | The name of the control. By default, Visual Basic generates the names Check1, Check2, and so on as you add subsequent check boxes to the form. |
| TabIndex | The focus tab order begins at 0 and increments every time that you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, the user can press Tab to move the focus to this check box. If False, the check box can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the check box. |
| Top | The number of twips from the top edge of a check box to the top of the form. |
| Value | Either 0-Unchecked (the default), 1-Checked, or 2-Grayed, indicating whether the check box is selected. |
| Visible | True or False, indicating whether the user can see (and, therefore, use) the check box. |
| Width | The number of twips wide that the check box consumes. |

Table 12.4 contains the events available for check box controls. As with option buttons, the Click event is generally the most commonly coded event procedure.

**Table 12.4. The check box's events.**

| *Event* | *Description* |
|---|---|
| Click | Occurs when the user clicks the check box with the mouse |
| DragDrop | Occurs when a dragging operation over the check box completes |
| DragOver | Occurs during a drag operation |
| GotFocus | Occurs when the check box receives the focus |
| KeyDown | Occurs when the user presses a key as long as the KeyPreview property is set to True for the controls on the check box; otherwise, the control gets the KeyDown event |
| KeyPress | Occurs when the user presses a key over the check box |
| KeyUp | Occurs when the user releases a key |
| LostFocus | Occurs when the check box loses the focus |

***Stop and Type:*** Load and run this book's project named CHECK.MAK. Listing 12.2 contains the code for the project's event procedures. The program allows the user to choose from one to four special formatting options for the poem inside the boxed label. The poem can accept any or all four formatting options, so the check box controls enable the user to select one or more of the formatting options.

***Review:*** The check box controls allow more than one selection from the user at a time, unlike the option button controls, which allow only a single option to be selected at a time.

**Listing 12.2. The code for the CHECK.MAK project.**

```
1: Sub Form_Load ()

2: ' Fill the label with a poem.

3: ' The poem will display initially

4: ' using the default label properties.

5:
Dim Newline As String

6: ' The newline sends a carriage return and

7: ' line feed sequence to the poem so that
```

```
8: ' the poem can appear as a multiple-line

9: ' poem.

10: Newline = Chr$(13) + Chr$(10)

11: ' Fill the label with the poem

12: ' and
concatenate the newline characters

13: lblPoem = "Visual Basic is the best."

14: lblPoem = lblPoem & Newline

15: lblPoem = lblPoem & "It's much better than the rest."

16: lblPoem = lblPoem & Newline

17: lblPoem =
lblPoem & "If you ever hear differently,"

18: lblPoem = lblPoem & Newline

19: lblPoem = lblPoem & "Give them major misery!"

20: End Sub

21:

22: Sub chkBack_Click ()

23: ' Set the poem label's background to Red or white


24: If (lblPoem.BackColor = RED) Then
```

```
25: lblPoem.BackColor = WHITE ' Default

26: Else

27: lblPoem.BackColor = RED

28: End If

29: End Sub

30:

31: Sub chkFore_Click ()

32: ' Set the poem label's foreground to Green or Black

33: If
(lblPoem.ForeColor = GREEN) Then

34: lblPoem.ForeColor = BLACK ' Default

35: Else

36: lblPoem.ForeColor = GREEN

37: End If

38: End Sub

39:

40: Sub chkItal_Click ()

41: ' Set the poem label's caption to italicize or not

42: If (lblPoem.FontItalic =
True) Then
```

```
43: lblPoem.FontItalic = False ' Default
44: Else
45: lblPoem.FontItalic = True
46: End If
47: End Sub
48:
49: Sub chkUnd_Click ()
50: ' Set the poem label's caption to underline or not
51: If (lblPoem.FontUnderline = True) Then
52:
lblPoem.FontUnderline = False ' Default
53: Else
54: lblPoem.FontUnderline = True
55: End If
56: End Sub
57:
58: Sub cmdExit_Click ()
59: End
60: End Sub
```

[ic:output]Figure 12.5 contains a figure that shows the poem when the user selects two of the check box options.

**Figure 12.5. Selecting two check boxes at the same time.**

***Analysis:*** Lines 1 through 20 contain the longest Form_Load() event procedure that you've seen in this book. The purpose of the procedure is to initialize the label control with the multiline poem. There is no MultiLine property for labels, so you have to trick Visual Basic into displaying multiple lines inside the label.

---

*Definition:* A *control character* produces an action, not a text character.

---

The Chr$() function (called the *character string function* or, sometimes, the *character function*) accepts a number inside its parentheses. The number must come from the ASCII table (Appendix A). Visual Basic converts that number to its ASCII character equivalent. ASCII number 13 is the *carriage return* character and is a special control character that sends the cursor to the beginning of the line, which, in the case of a label, is the beginning column in the label. ASCII number 10 is the *line feed* character that sends the cursor to the next line on the screen. The effect of the concatenated ASCII value of 13 and 10 (line 10) is that a special double control character is created that, when displayed in a label, sends the cursor to the beginning of the label's next line.

---

**Note:** The poem label's Alignment property is set to 2-Center, so the lines inside the label always display as centered within the label's border.

---

Lines 13 through 19 then build the multiline poem by concatenating one line and the newline combination string variable to the label's Caption property.

---

**Note:** But wait! Lines 13 through 19 don't even mention the Caption property! It appears that the poem's text is being sent to the label itself and not to any property. It turns out that each control has a default property that, unless you specify a different property name, acts as the default property. Therefore, Visual Basic assigns all of the poem's lines to the Caption property in lines 13 through 19 because the Caption property is the default property for labels.

---

Lines 22 through 56 contain the four check box Click event procedures that set or reset each of the four poem properties described by the check box. The If-Else checks first to see whether the poem's property is set to the checked value. If so, the true part of the If sets the property back to its default state. If the property already contains the default value, the Else sets the property to the new state.

As always, the application contains an Exit command button event procedure in lines 58 through 60 that terminate the program at the user's request.

# Multiple Sets of Option Buttons

***Concept:*** Although the user can select only one option button from all the option buttons on the form, there is a way to place multiple sets of option buttons on a form inside frames. The user can select only one option button within any frame at a time.

Figure 12.6 shows where the frame control resides on the Toolbox window. The frame control is a holder of other controls. By placing more than one frame on a form, you can group more than one set of option buttons on the form together.

**Figure 12.6. The location of the frame control.**

When you want to place several sets of options buttons on a form, be sure to place more than one frame on the form first. The frames must be large enough to hold as many option buttons as each group requires.

Be careful about placing option buttons when you want to frame them within frame controls. You must draw the option buttons *inside the frame*. You can't create the option buttons elsewhere and move them into the frame. For most applications, you can double-click controls to place them in the middle of the form, and then drag the controls to their final location and resize them. When placing option buttons inside frames, you must click (not double-click) the option button control on the Toolbox window and then draw the option button by clicking the mouse inside the frame and dragging the mouse until you've approximated the option button's size. When you release the mouse button, Visual Basic will draw the option button in the size you drew it.

***Stop and Type:*** Load and run this book's project named OPTIONS.MAK. Listing 12.3 contains the code for the project's event procedures. The program is similar to the CHECK.MAK application that you saw in the previous section. Framed groups of option buttons turn selected poem formatting properties on or off.

***Review:*** The frame enables you to group option button controls together. Be sure to place the frame before drawing option button controls inside the frame.

**Listing 12.3. The code for the framed option buttons.**

```
1: Sub Form_Load ()

2: ' Fill the label with a poem.

3: ' The poem will display initially

4: ' using the default label properties.
```

```
5: Dim Newline As String

6:
' The newline sends a carriage return and

7: ' line feed sequence to the poem so that

8: ' the poem can appear as a multiple-line

9: ' poem.

10: Newline = Chr$(13) + Chr$(10)

11: ' Fill the label with the poem

12: ' and concatenate the newline
characters

13: lblPoem = "Visual Basic is the best."

14: lblPoem = lblPoem & Newline

15: lblPoem = lblPoem & "It's much better than the rest."

16: lblPoem = lblPoem & Newline

17: lblPoem = lblPoem & "If you
ever hear differently,"

18: lblPoem = lblPoem & Newline

19: lblPoem = lblPoem & "Give them major misery!"

20: End Sub

21:
```

```
22: Sub optGreen_Click ()

23: lblPoem.BackColor = GREEN

24: End Sub

25:

26: Sub optItal_Click ()

27:
lblPoem.FontItalic = True

28: End Sub

29:

30: Sub optNoItal_Click ()

31: lblPoem.FontItalic = False

32: End Sub

33:

34: Sub optNoUnd_Click ()

35: lblPoem.FontUnderline = False

36: End Sub

37:

38: Sub optRed_Click ()

39: lblPoem.BackColor = RED
```

```
40: End Sub

41:

42: Sub optUnd_Click ()

43: lblPoem.FontUnderline = True

44: End Sub

45:

46: Sub optWhite_Click ()

47: lblPoem.BackColor = WHITE

48: End Sub

49:

50: Sub cmdExit_Click ()

51: End

52: End Sub
```

***Output:*** Figure 12.7 contains the OPTIONS.MAK application after the user selects an option in each of the three frames.

**Figure 12.7. The framed option buttons allow for multiple settings.**

***Analysis:*** The difficult part of framed option controls is placing the option buttons inside the frames, and even that's not extremely difficult. After you draw the option buttons and the frames, the usual event procedures that you've been writing work for the controls.

Lines 22 through 48 contain the Click event procedures that set the poem's label formatting properties.

# Control Arrays Simplify

***Concept:*** By putting one or more similar controls inside a control array, you gain the advantage of writing a single event procedure that can handle more than one control on the form.

---

*Definition:* A *control array* is a list of controls with the same name.

---

When you place more than one control of the same control type on a form and assign the same Name property to those controls, you're creating a control array. As with variable arrays, a subscript that begins at 0 differentiates one control from another. The Index property of each control in the array contains that control's subscript location in the array. If you want to renumber the controls in the array to begin at 1, you can do so by changing the first zero-based control's Index value to 1.

---

**Note:** Each control in a control array must be the same data type.

---

If there were five text boxes in a control array named txtBoxes, each individual control would be named txtBoxes(0), txtBoxes(1), txtBoxes(2), txtBoxes(3), and txtBoxes(4).

---

[ic: Note]As soon as you place a control on the form that has the same name as an existing control, Visual Basic makes sure that you want to begin a control array by issuing the warning message box shown in Figure 12.8. The designers of Visual Basic knew that you may accidentally attempt to place two controls on the same form with the same name, and the message box makes sure of your intent to create a control array. If you select the No command button to the warning message box, Visual Basic returns the second control to a its default name.

---

**Figure 12.8. Visual Basic ensures that you want a control array.**

***Stop and Type:*** Load and run this book's project named CHECK2.MAK. Listing 12.4 contains the code for the project's chkAll_Click() event procedure. The program is similar to the CHECK.MAK application that you saw in the previous section, except that, instead of four separate check box event procedures, CHECK2.MAK contains a single check box control array named chkAll.

***Review:*** By placing several controls in a control array, you can reference each control by its subscript rather than by a different name for each control. Rather than write several event procedures, you need to write only a single event procedure.

**Listing 12.4. The code for control array's event procedure.**

```
1: Sub chkAll_Click (Index As Integer)

2: ' A single Select Case

3: ' handles all check box formats

4: Select Case Index

5: Case 0:

6: ' Set the poem label's background to Red or white

7: If (lblPoem.BackColor =
RED) Then

8: lblPoem.BackColor = WHITE ' Default

9: Else

10: lblPoem.BackColor = RED

11: End If

12: Case 1:

13: ' Set the poem label's foreground to Green or Black

14: If (lblPoem.ForeColor = GREEN) Then

15: lblPoem.ForeColor = BLACK ' Default

16:
Else

17: lblPoem.ForeColor = GREEN
```

```
18: End If

19: Case 2:

20: ' Set the poem label's caption to italicize or not

21: If (lblPoem.FontItalic = True) Then

22: lblPoem.FontItalic = False ' Default

23: Else

24: lblPoem.FontItalic = True

25: End If


26: Case 3:

27: ' Set the poem label's caption to underline or not

28: If (lblPoem.FontUnderline = True) Then

29: lblPoem.FontUnderline = False ' Default

30: Else

31: lblPoem.FontUnderline = True

32: End If

33: End Select

34: End Sub
```

***Analysis:*** The single Select Case statement handles all four check boxes. There is only one single check box Click() procedure in the entire application even though there are four check boxes. Each check box is named chkAll, so they all a control array of check boxes and there will be only one event procedure for

each event that the programmer wants to program.

Therefore, whenever the user clicks *any* of the check boxes, the chkAll_Click() event procedure executes. How does the procedure know *which* check box triggered the event? Line 1, the event procedure's first line, gives you a good clue. Visual Basic sends the Index value of the check box that the user clicked to the event procedure. The parentheses is a mechanism that Visual Basic uses to collect values such as the index to the control array that triggered the event. You'll understand such parenthetical values in Lesson 8, but until then, be assured that after the chkAll_Click() procedure executes, an Index variable will contain the subscript of the check box in the control array that the user selected.

The Select Case is a consolidation of Listing 12.2's multiple check box Click() event procedures. As you can see, the control array saves time and memory because a single event procedure can now handle all the events in the control array.

# Homework

## General Knowledge

1. Which control offers a single choice from a multiple set of choices?
2. Which control is best for allowing the user to select from one or more options?
3. Why are option buttons sometimes called *radio buttons*?
4. What happens if the user selects an option button that's different from the option button currently selected?
5. True or false: You can write code ensuring that no option button is selected when the program first executes.
6. What does the Alignment property do for option buttons and check boxes?
7. What property determines whether an option button is selected?
8. What property determines whether a check box is selected?
9. True or false: One or more Value properties can be set for option buttons on a form (assume that the option buttons aren't placed in frames).
10. True or false: One or more Value properties can be set for option buttons on a form (assume that the option buttons are placed in frames).
11. What is a frame?
12. True or false: You can place one frame at the most on a form at any one time.
13. What is a *control character?*
14. Describe how to initialize a label control with multiline text.
15. What does the Chr$() function do?
16. What is a *control array*?
17. If a control array contains seven controls, how many names do the controls have?

18. How do you distinguish between values in a control array?
19. What is the default starting subscript for control arrays?
20. How many event procedures do you have to write for a control array that contains eight controls?
21. What property holds a control array's subscript number?
22. How does Visual Basic make sure that you want to create a control array?

## Write Code That...

1. Write the opening line of an DblClick event procedure for a command button control array named cmdAll.

**Find the Bug**

1. Opal is trying her hand at frames and option buttons. First, Opal places three frames on her form. Then, Opal double-clicks the option button control on the Toolbox window nine times and moves each control to its appropriate frame. Opal doesn't seem to be able to select more than one option button at a time on the entire form, even though the three sets of option buttons appear on different frames. What is Opal doing wrong?

## Extra Credit

1. What character does the following assignment statement place in the string variable named MyGrade?
   MyGrade = Chr$(65)

Create a form that contains ten option buttons, two sets of five in two frames. Put the ten option buttons inside two control arrays. Label each option button One, Two, Three, Four, and Five. Write two Click event procedures, one for each option button control array, that beep the PC's speaker once for whatever option buttons are selected at that time. In other words, if the user clicks the left frame's Three option button, and the right frame already has the value Two selected, you'll click the speaker five times.

# Visual Basic in 12 Easy Lessons



# Project 6

## Combining Code and Controls

### Stop & Type

This lesson taught you how to work with advanced array data as well as how to place option button and check box controls on the form. The arrays simplify your programs by letting you step through data and controls using For loops. The option buttons and check boxes offer your users additional ways to select from choices that your program offers.

You can now add control to your programs by putting loops in the code that repeat sections of the program. Computers are excellent tools for repeating calculations as many times as necessary to produce results. Loops also provide ways for you to check user input against good and bad values and repeat the input prompting as many times as necessary for the user to enter data in the required format and range.

In this lesson, you saw the following:

- How arrays help simplify the definition of multiple variables
- How control arrays help work with multiple controls (instead of several event procedures, you need to write only one)
- When to use check boxes and when to use option buttons
- Where to use framed option button groups to create sets of option buttons from which the user can select

## The Program's Description

Figure P6.1 shows the PROJECT6.MAK application as soon as you load and run the program. The project is simple but uses two sets of framed option buttons and two control arrays of option buttons. The project's controls are as follows:

- Two frames, one holding 16 color option buttons and the second holding three beeping options
- 16 option buttons inside the left frame that select a background color for the form
- 3 option buttons inside the right frame that determine how many beeps the user wants to hear
- An Exit command button to terminate the application

**Figure P6.1. The project's opening screen.**

The application is simple *because* of the control arrays. Control arrays greatly simplify an application in which several similar controls are needed to perform similar functions. Without the control arrays, this extremely introductory project would be burdened by at least *twenty* event procedures just to perform the same task that it currently performs with only *three* event procedures!

## The Program's Action

When you first run the program, the form's background color is set to Black simply because the first option button in the left frame is selected. None of the beeping options is set when the program first executes.

You can select one of the beeping options in addition to the color options. Remember that when option buttons appear in separate frames, the option buttons are grouped to allow for one selection from each group. Without the frames, you could select a color *or* select a beep option button, but not both.

Click the second beeping option button and you'll hear two beeps. Select a different color option button and the form's background color changes accordingly.

## The Event Procedures

Listing P6.1 contains PROJECT6.MAK's event procedures. The Index value sent to the optBeep_Click() and to the optColor_Click() event procedures are used inside each procedure to beep or set an appropriate form color.

---

**Note:** The QBColor() function is a built-in Visual Basic that returns a hexadecimal color code so that you don't have to know anything about hexadecimal values. When you use an integer value from 0 to 15 inside QBColor()'s parentheses, Visual Basic function sets a color that matches that number. The colors

correspond to the order shown on the form. The Index values for the sixteen color values are numbered from 0 to 15, so the event procedure can use that color code inside QBColor().

---

**Listing P6.1. The color changing, beeping, and exit event procedures.**

```
1: Sub optCol_Click (Index As Integer)

2: ' One of the color option buttons

3: ' triggered this event

4:
frmOpt.BackColor = QBColor(Index)

5: End Sub

6: Sub optBeep_Click (Index As Integer)

7: ' The beeping option buttons have Index

8: ' properties that begin with 1 that

9: ' correspond to the number of beeps.

10: Dim Ctr As Integer

11: Dim Delay As
Long

12: For Ctr = 1 To Index

13: Beep

14: For Delay = 1 To 80000

15: ' Do nothing but waste time...
```

```
16: Next Delay

17: Next Ctr

18: End Sub

19: Sub cmdExit_Click ()

20: End

21: End Sub
```

## Descriptions

1: The color-changing option buttons make up the control array named optCol, so the name of the Click event procedure is optCol_Click(). No matter *which* of the 16 color option buttons the user clicks, this event procedure executes.

2: A remark helps explain the purpose of the event procedure.

3: A remark helps explain the purpose of the event procedure.

4: Sets the form's background color property to the QBColor() value of the Index that corresponds to the option button clicked.

5: Terminate the event procedure.

---

5: Without control arrays, each option button would require its own event procedure.

---

6: The beeping option buttons make up the control array named optBeep, so the name of the Click event procedure is optBeep_Click(). No matter *which* of the 3 beeping option buttons the user clicks, this event procedure executes.

7: A remark helps explain the purpose of the event procedure.

8: A remark helps explain the purpose of the event procedure.

9: A remark helps explain the purpose of the event procedure.

10: Define an integer variable used for the loop control variable.

11: Define a long integer variable used for slowing down the loop.

12: Loops enough times to match the option button's Index value, which, according to the Property window, contains the values from 1 to 3.

13: Beep the computer's speaker.

14: Begin a huge For loop that slows down the beeps.

15: A remark is the only thing inside the innermost nested loop.

---

15: A nested loop slows down the computer's beeping speed.

---

16: Continue wasting some time by repeating the inner loop.

17: Beep again if the beeps have not completed yet.

18: Terminate the event procedure.

## Close the Application

You can now exit the application and exit Visual Basic.

Are you getting tired of beeping and color-changing programs? The simple applications that you've been working with have kept your mind focused on the Visual Basic environment and language. You've now reached the halfway point of this book's education. You have enough tools in your Visual Basic programming repertoire to begin writing more powerful applications, and you'll do just that in the next lesson.

# Visual Basic in 12 Easy Lessons



# Lesson 7, Unit 13

## The Built-In Functions

### What You'll Learn

- A function review
- The numeric functions
- The string functions
- General functions

If you want to build a house, you'll probably buy prefabricated windows and pre-hung doors instead of framing and hanging them yourself. Unless you want to spend extra time doing the work that you don't have to do, using the prefabricated windows and pre-hung doors makes quick work of those housing elements and allows you to concentrate your energies on the more esthetic items (such as talking your spouse into allowing you to partition two-thirds of the master bedroom for a great computer work area...).

Visual Basic's built-in functions save you all kinds of programming time just as the pre-built housing

parts do when home building. The built-in functions work like miniature programs that perform common tasks so that you can concentrate your efforts elsewhere. If, for instance, you need to round a single-precision value to an integer, Visual Basic supplies not one but *three* built-in functions that round numbers to integer values.

---

**Tip:** You've already seen several built-in functions in this book, such as Val() and Chr$(). Usually, though not always, a function that contains a dollar sign after its name is a *string function*, and a function without a dollar sign in its name is a *numeric function*. This unit teaches you about both numeric and string functions.

---

## A Function Review

***Concept:*** Functions accept one or more *arguments* and do work with those arguments. The function then returns a single value.

When you use a function inside a Visual Basic statement, it is said that you're *calling* a function. For example, the following statement *calls* the Val() function:

```
Num = Val(txtAmt.Text) ' Convert entry to number
```

---

*Definition:* An *argument* is a value that you pass to a function.

---

The parentheses after a function name hold one or more arguments. In the preceding statement, txtAmt.Text is the argument. Some functions, such as Val(), accept only one argument. Some functions accept more than one argument. Some functions don't require any arguments, and for those functions, the parentheses are optional. The arguments inside the parentheses are often called the *argument list*. A function can require none, one, or several arguments.

When you specify function arguments, you are *passing* the arguments to the function. The function works on the argument values that you pass to the function and produces some sort of value from the arguments. As Figure 13.1 illustrates, you send a function one or more arguments and the function, after using the arguments in the argument list, is said to return a value. That value is the answer to the function, more commonly called the *return value*. Functions are said to *receive arguments* and *return a value*.

**Figure 13.1. A function works on its arguments and returns one value.**

---

**Note:** It's worth making this very clear: A function always returns a single value and *never more*. Although a function's argument list can sometimes contain five or more arguments, a function never returns more than one value.

---

If a function requires more than one argument, you'll separate those arguments by one or more commas. For example, here is a function named Right$() that requires two arguments, MyName and 10, in its argument list:

```
Last = Right$(MyName, 10) ' Strip off last name
```

You'll always have to do something with a function's return value. Here are some things that you might do with a return value:

- Display the return value in a control on the form by assigning the function's return value to a control
- Assign the return value to a variable
- Use the return value inside a calculation
- Use the return value as an argument to another function by nesting one function inside the argument list of another function

Basically, anything that you might do with a variable or a constant, you'll do with a function's return value. For example, you learned in Unit 7 how Val() works. Val() converts its string (or variant) argument to a number. The converted number is Val()'s return value. Val() will convert its argument to a single, return value; therefore, you can't place Val() on a line by itself like this:

```
Val(UserAddress) ' Invalid!
```

Val() will accept the string named UserAddress and convert that string to a number. You must make sure that the program does something with that number! The function is just a predefined, built-in routine, supplied for you by Visual Basic, which operates on the argument list and returns a value that you've got to do something with. Here are four uses of Val() that use the return value in different ways:

```
lblHouseNum.Caption = Val(UserAddress) ' To a label
```

```
MyHouseNum = Val(UserAddress) ' To a variable
```

```
OldAge = 65 + Val(txtAge.Text) ' A calculation
```

```
Last = Right$(UserName, Val(Ans)) '
Another function
```

An argument might even be an expression. The following Val() function first concatenates the string expression used as the argument, and then returns the numerically converted value of that string argument:

```
Number = Val(aStr1 & aStr2)
```

---

**Note:** The argument list's data type doesn't always match the function's return value data type. For example, a function might take a string argument and return a number. A function might require both a string and a numeric argument, and return a string.

---

The built-in functions never change their arguments. The built-in functions use their arguments to create a new value: the return value.

***Review:*** There are several built-in functions in Visual Basic that your programs can call. Programs that call functions must do something with the return values from those functions. A function might require one argument or more. If you send an argument list to a function, that function operates on those arguments and returns a single value based on the argument list.

## The Numeric Functions

***Concept:*** Several numeric functions work to save you programming time when you're working with numbers. There are conversion functions, scientific functions, and trigonometric functions.

As you read through the following pages, don't worry about memorizing every function. Try to get a general idea of the kinds of functions supplied by Visual Basic. The functions work to save you time. If you need to convert a number or compute a standard formula, the chances are great that Visual Basic includes a function that does the work for you. For example, there is no reason to write an advanced trigonometric sine function, because Microsoft already wrote one for you.

---

**Note:** You don't have to be a math lover to use and appreciate the Visual Basic numeric functions. Actually, the less you like math, the *more* you'll appreciate the fact that Visual Basic supplies all these functions for you so that you never have to write the code to accomplish the same thing.

---

## Converting Numbers

Visual Basic supplies three functions that convert single-precision and double-precision numbers to integer values. When writing applications, you might need to round numbers down or up to their nearest integers. Table 13.1 contains the three integer conversion functions.

**Table 13.1. The integer conversion functions.**

| *Function* | *Description* |
|---|---|
| CInt() | Rounds fractional values of .5 and more to the next highest integer |
| Fix() | Truncates the fractional portion |
| Int() | Rounds the number down to the integer less than or equal to its arguments |

Both Fix() and Int() treat positive integers the same way. Both of the following statements return and store 14:

```
ans1 = Int(14.6)

ans2 = Fix(14.6)
```

The Fix() and Int() functions behave differently for negative numbers. The remarks to the right of the following statements indicate the functions' return values:

```
ans3 = Int(-14.6) ' Stores -15
```

```
ans4 = Fix(-14.6) ' Stores -14
```

The negative number less than or equal to -14.6 is -15; hence, the return values for Int() shown in the first statement's remark.

The CInt() function returns truly rounded numbers, as shown here:

```
ans5 = CInt(14.1) ' Stores 14

ans6 = CInt(14.6) ' Stores 15

ans7 = CInt(-14.1) ' Stores -14

ans8 =
CInt(-14.8) ' Stores -15
```

There are functions similar to CInt() that convert arguments of any data type to any other data type. Table 13.2 contains the rest of the data conversion functions.

**Table 13.2. The data type conversion functions.**

| *Function* | *Description* |
|---|---|
| CCur() | Converts the argument to an equivalent currency data type |
| CDbl() | Converts the argument to an equivalent double-precision data type. |
| CLng() | Converts the argument to an equivalent long integer data type |
| CSng() | Converts the argument to an equivalent single-precision data type |
| CStr() | Converts the argument to an equivalent string data type |
| CVar() | Converts the argument to an equivalent variant data type |

---

**Caution:** If the converted argument won't fit in the target data type, Visual Basic issues the Overflow error message shown in Figure 13.2. You've got to make sure, perhaps through an initial If statement, that the argument fits within the range of the converted data type.

---

Normally, the following assignment stores .1428571 in the label named lblValue:

```
lblValue.Caption = (1 / 7) ' Assigns .1428571
```

The following, however, adds precision to the answer for a more accurate calculation:

```
lblValue.Caption = CDbl(1 / 7) ' Assigns .142857142857143
```

**Figure 13.2. Don't overflow the target data type.**

---

Don't even *mention* other bases!: If you *really* want to get fancy with numeric conversions, Visual Basic supports four functions that convert their decimal base-10 arguments to octal (base-8) and hexadecimal (base-16) return values. You'll want to use these functions if you write advanced applications that deal with other number bases.
The two hexadecimal functions, Hex() and Hex$(), convert their numeric arguments to a variant and string hexadecimal value, respectively. The two octal functions, Oct() and Oct$(), convert their numeric arguments to a variant and string octal value, respectively.
If you think that you won't use these functions, forget them! These functions, especially the hexadecimal functions, come in handy for programmers who write system-level applications such as memory- and disk-maintenance programs.

---

The Asc() function converts its string argument to a corresponding ASCII table value. Generally, you'll pass a one-character string (or a variant that can equate to a string) value to Asc(). If the argument contains more than one character, Asc() ignores all characters following the first one.

The following statement stores the number 65 in the numeric variable named Initial because 65 is the uppercase letter *A* in the ASCII table:

```
Initial = Asc("A")
```

Table 13.3 lists the remaining math functions that Visual Basic provides. Many of the functions are scientific and mathematical. You may not need any or all of the functions unless you write heavy, math-intensive applications.

**Table 13.3. The scientific and mathematical functions.**

| *Function* | *Description* |
|---|---|
| Abs() | Returns the absolute value of the argument |
| Atn() | Returns the computed arc tangent of the argument expressed in radians |
| Cos() | Returns the computed cosine of the argument expressed in radians |
| Exp() | Returns the base of the natural logarithm argument |
| Log() | Returns the natural logarithm of the argument. |
| Sin() | Returns the computed sine of the argument expressed in radians |
| Sqr() | Returns the square root of the argument |

| | |
|---|---|
| Tan() | Returns the computed tangent of the argument expressed in radians |

**Tip:** If you compute trigonometric values on arguments expressed as degrees instead of radians, multiply the argument by *pi* and divide by 180. The following expression assigns the sine of 38 degrees to a variable named sVal:

```
sVal = Sin(38 * 3.14159 / 180)
```

***Review:*** The numeric functions will help you write programs when you need to include mathematical calculations in the code. Many business, graphics, and research applications sometimes utilize complex routines, and Visual Basic's math functions will help shorten your programming time and improve the accuracy of your code.

## The String Functions

***Concept:*** There are several Visual Basic string functions that manipulate string data better than most other programming languages allow. The string functions allow you to strip away characters from string data and change strings in various ways.

The Visual Basic programming language supports strings using the variable-length and fixed-length strings that you've seen defined throughout the first half of this book. The string manipulation provided by Visual Basic is perhaps more powerful than any other non-BASIC programming language in widespread use today.

You've already seen the Chr$() function that converts an ASCII number to its character equivalent. (There is a related Chr() function that returns a character in the variant data type also.) The Str$() (and the related variant-returning Str() function) converts a number to a string.

**Tip:** Convert numbers to strings when you want to display numeric quantities inside a message box.

Suppose that the user's age were stored in a numeric variable named Age. If you wanted to display the user's age in a message box, you couldn't do so without first converting the age to a string value as follows:

```
MsgBox "Your age is now" & Str$(Age)
```

The string case-conversion functions convert their string arguments to uppercase or lowercase character strings. Table 13.4 lists these functions.

**Table 13.4. The case-conversion functions.**

| *Function* | *Description* |
|---|---|
| LCase() | Returns the string argument as a variant data type that's all lowercase letters |
| LCase$() | Returns the string argument as a string data type that's all lowercase letters |
| UCase() | Returns the string argument as a variant data type that's all uppercase letters |
| UCase$() | Returns the string argument as a string data type that's all uppercase letters |

If the argument already contains one or more characters in the target case, the functions leave those characters alone. The remarks following these statements indicate the function return values:

```
lowerName = LCase("Larry") ' Returns larry

upperName =
UCase("Smythe") ' Returns SMYTHE
```

---

*Definition:* A *substring* is a portion of a string.

---

Visual Basic contains three string functions that return left, center, or right portions of a string. These are the Left$(), Mid$(), and Right$() functions. All three functions are said to return a substring value. Here are the formats of these functions:

```
Left$(StringVal, length)

Right$(StringVal, length)

Mid$(StringVal, startVal, length)
```

---

**Note:** The Mid$() function is called the *midstring* function.

---

The Left$() function returns *length* characters from the *StringVal*. The *StringVal* might be a string

constant or variable. Likewise, Right$() returns *length* characters from the *StringVal*. Mid$() returns *length* characters from the *StringVal* beginning at the *startVal* character.

The remarks following these statements indicate the function return values:

```
Title = "Something Good!"

lTitle = Left$(Title, 4) ' Some

rTitle = Right$(Title, 5) ' Good!

mTitle = Mid$(Title, 5, 8) ' thing Go
```

Figure 13.3 shows how Visual Basic strips these substrings from the longer string values.

**Figure 13.3. Substring functions pull substrings from strings.**

The Len() function returns the number of characters stored in strings. Often, you'll have to adjust control sizes to hold long strings. You can use the Len() function as a guide to calculate how wide a control must be. The following statement stores 14 in the variable named Length:

length = Len("Blue and Green")

---

**Note:** The Len() function works for numeric values as well. If you pass a numeric variable, constant, or expression to Len(), Len() returns the amount of memory needed to hold that numeric value.

---

The LTrim$() and RTrim$() functions trim extra spaces from the beginning or end of a string. If extra spaces don't exist, the LTrim$() and RTrim$() functions do nothing. LTrim$() returns the argument's string without any leading spaces. RTrim$() returns the argument's string without any trailing spaces. The Trim$() function handles both jobs by trimming both leading and trailing spaces from a string.

Here are the formats of the string-trimming functions:

```
LTrim[$](StringExpression)

RTrim[$](StringExpression)

Trim[$](StringExpression)
```

---

There are also equivalent variant functions called LTrim(), RTrim(), and Trim().

---

The following statements trim spaces from the beginning, end, or both sides of strings:

```
LStr1 = LTrim$(" Jonah") '
Stores Jonah

RStr2 = RTrim$("Jonah ") ' Stores Jonah

AnySt3 = Trim$(" Jonah ") ' Stores Jonah
```

Without the trimming functions, the spaces are copied into the target variables along with the name Jonah.

Earlier, you learned how to use Str$() to convert a number to a string. Because Str$() always converts positive numbers to strings with a leading blank (where the imaginary plus sign appears), you can combine LTrim$() with Str$() to eliminate the leading blank. The first of the following two statements stores the leading blank in st1. The second uses LTrim$() to get rid of the blank before storing the string into st2.

```
st1 = Str$(234) ' Stores " 234"

st2 = LTrim$(Str$(234)) ' Stores "234"
```

***Review:*** The string functions manipulate strings and return values based on string arguments. As you write more powerful programs, you'll need to extract substrings and convert the case of strings in various ways to display string values on the form and in controls.

## General Functions

***Concept:*** There are a handful of general functions that don't fall into a strict numeric or string category. These general-purpose functions add power to your programming skills. This section explains how to detect data types using the Is...() and *VarType*() functions. You may not see a use for these functions quite yet, but familiarize yourself with them. In Lesson 8, you'll learn how to pass data back and forth between Visual Basic routines, and there are times when you'll need to know which data type was passed to you using one of the functions taught here.

Table 13.5 contains *data inspection* functions, often called the *Is...()* functions. When you store a data value in a variant variable (which can accept any data type), Table 13.5's functions returns a true or false result that indicates whether the argument can be converted to a specific data type.

**Table 13.5. The Is...() data inspection functions.**

| *Function Name* | *Description* |
|---|---|
| IsDate() | Determines whether its argument can be converted to a valid date |
| IsEmpty() | Determines whether its argument has been initialized with any value since the argument's original definition |
| IsNull() | Determines whether its argument holds a Null value (such as an empty string) |
| IsNumeric() | Determines whether its argument holds a value that can be converted to a valid number |

The following code ensures sure that a valid string appears in the variable named ans before applying a UCase$() function:

```
If IsString(ans) Then

NewAns = UCase$(ans)

Else

 MsgBox "You didn't type a letter!"

End If
```

Empty variables differ from Null values and zero values. Empty variables indicate that nothing has been entered into variables. The IsNull() function checks to see whether its control argument contains a Null value.

---

**Tip:** Use IsNull() to see whether a control or field on a form contains data. Use IsEmpty() just for variables.

---

The VarType() function determines the data type of its argument. Table 13.6 lists the return values from the VarType() function. VarType() returns no values other than the nine listed in the table.

**Table 13.6. The VarType() function's return values.**

| *Return* | *Description* |
|---|---|
| 0 | Indicates that the argument is Empty |
| 1 | Indicates that the argument is Null |
| 2 | Indicates that the argument is Integer |
| 3 | Indicates that the argument is Long |
| 4 | Indicates that the argument is Single |
| 5 | Indicates that the argument is Double |
| 6 | Indicates that the argument is Currency |
| 7 | Indicates that the argument is Date |
| 8 | Indicates that the argument is String |

***Stop and Type:*** Listing 13.1 makes sure that the user entered a value into the txtAge text box control.

***Review:*** The general-purpose functions return values that indicate data contents and data types. You can use these functions to see whether variables and controls have been initialized before calculating with those values.

**Listing 13.1. Checks to make sure that the user entered a value.**

```
1: If IsNull(txtAge.Text) Then

2: MsgBox "You didn't type anything!"

3: Else

4: MsgBox "Thanks for entering a value."

5: End If
```

***Analysis:*** Line 1 ensures that the user has typed *something* in the control named txtAge. Subsequent code could then use the VarType() function to make sure that the user entered a value of the correct data type required by the user.

## Homework

### General Knowledge

1. What is a function *argument*?

2. True or false: A function's return value must match the argument list's data type.
3. True or false: Some functions require no arguments, some one, and some several.
4. True or false: The function's arguments must all match in data type.
5. Why are there three integer functions?
6. What value does Visual Basic store in Ans given the following assignment statements?
   Ans = Int(61.32)
   Ans = Int(-61.32)
   Ans = Fix(-61.32)
   Ans = Cint(421.51)
7. True or false: Int(), CInt(), and Fix() all return the same value for positive arguments.
8. What function performs the opposite of Chr$()?
9. What value appears in the string variable named AList after these assignments?
   AList = UCase$("AbCdEfG")
   AList = LCase$("AbCdEfG")
10. What would the following assignment statement store in the variant variable named Anything?
    Anything = Str$(Val("78.1"))
11. Write the values stored in each of the following assignment statements:
    AStr = Left$("Sams", 1)
    AStr = Left$("Sams", 3)
    AStr = Right$("Sams", 1)
    AStr = Right$("Sams", 3)
    AStr = Mid$("Sams", 2, 3)
12. True or false: Depending on the arguments, Mid$() could return the same values as Left$() and Right$().
13. True or false: Both of the following produce the same value when used inside an expression:
    Chr$(67)
    Asc("C")
14. What is the name of the function that converts a number to a string?

## Write Code That...

1. Suppose that you needed to display a numeric variable's value inside the prompt string of an input box. Describe how you would you concatenate such a variable to the prompt string?
2. How could you find out the amount of memory consumed by 250 double-precision variables?
3. Write the code that uses an input box to get a number from the user and then uses a message box to display the square root of that number.
4. Write the code that uses two separate input boxes to ask the user for a first and last name, and then display in a message box the total number of letters in both names.
5. Write an assignment statement that stores the ASCII value of "P" in a variable named ValAsc.

## Find the Bug

1. Rudy gets an overflow message when attempting the following assignment into the integer variable named Weight:
   Weight = CInt(32768 * Num)
   Describe the problem for Rudy.

## Extra Credit

1. Add to Listing 13.1 to make sure that the user entered an integer in the txtAge.Text control.

Write a program that stores the 256 ASCII characters (from ASCII 0 to ASCII 255) in a string array that's defined to hold 256 elements.

# Visual Basic in 12 Easy Lessons



# Lesson 7, Unit 14

## Working with Dates, Times, and Formats

### What You'll Learn

- Getting the date and time
- Setting the date and time from Visual Basic
- Determining how much time has passed
- Using date arithmetic
- Getting serial dates and times

This unit completes your learning of Visual Basic's built-in functions by exploring the built-in date, time, and formatting functions. This is the first unit of the book in which you'll work directly with the variant data type even though you've seen references to Variant throughout the book.

In addition to the functions, this unit discusses the several date and time statements that augment the work you'll do with date and time function values. Date and time values are critical for time keeping, transaction recording, and reporting, so you must be able to write programs that track and work with such

values. Visual Basic supports one of the most comprehensive library of date and time functions, as you'll see in this unit.

---

**Note:** The most important part of learning many of the date and time values is learning how Visual Basic stores date and time values in memory.

---

The formatting power of Visual Basic is superb as well. You'll see the formatting tables that convert your program output to any format needed. Visual Basic supports date, time, string, and number formatting capabilities.

## Getting the Date and Time

***Concept:*** The Now(), Date$() and Time$() functions (and their cousins, the Date() and Time() functions) look inside your computer's clock and calendar to retrieve the current date and time for your program's use.

In ancient times (a little over eight years ago), computers couldn't remember the date and time if you turned them off. They've come a long way! Somebody thought of the brilliant idea of putting a battery inside the computer so that the computer would remember the date and time every time you powered on the machine. Of course, batteries had been in wrist watches for years before then, but computer makers were so busy trying to squeeze 360K of storage onto one of those little 5 1/4-inch floppy disks that they didn't get around to the battery for a while.

Visual Basic gives your Windows programs access to the date and time values stored internally in your computer. Assuming that the date and time are set properly, you can assign the date and time function return values to variables or display those values in controls.

---

*Definition: 24-hour time* measures time from 0 to 24 hours.

---

Some of the time values returned from these functions return a 24-hour time. If you're used to expressing time values using a 12-hour a.m. or p.m. time, you'll have to get accustomed to the 24-hour time in which Visual Basic adds 12 to all time values after 12:59 PM. Therefore, 3:45 in the morning is 3:45, but 3:45 in the afternoon is 15:45.

---

**Note:** You can use the Format() function to change any 24-hour time to a 12-hour clock using the AM and PM indicators if you want to.

---

The Now(), Date$(), Date(), Time$(), and Time() functions are functions that don't accept arguments. As such, Visual Basic removes the parentheses if you type them after these functions. Nevertheless, this

book does use the parentheses after the functions to remind you that they are built-in functions.

The Now() function returns both the date and time. Now() returns a Variant data type in the following format:

*mm/dd/yy hh*:*mm*:*ss* [AM][PM]

where *mm* is a 2-digit month, *dd* is a 2-digit day, and *yy* is a 2-digit year value. The time appears to the right of the date. The time section of Now()'s return value uses a 12-hour clock rather than the 24-hour clock used by Time$() and Time(). *hh* is a 2-digit hour, *mm* is a 2-digit minute value, and *ss* is a 2-digit second value. Just for your information, Visual Basic stores this complete value internally as a double-precision value because only the double-precision value is large enough to hold that much information. Even though Visual Basic stores the date and time internally in a double-precision value, Now() returns the value formatted as shown inside the variant data type.

If you've set your computer's version of Windows to an international setting that requires a different date and time format, these functions return date and time values that match your country's setting.

Date$() returns the system date string in the following format:

mm-dd-yyyy

Date() returns the value in the variant data type. The difference between Date() and Date$() is that Date() doesn't return leading zeros in day or month numbers less than 10, Date() doesn't append 19 to the year, and Date() inserts forward slashes instead of hyphens between the date values.

---

**Tip:** Given the closeness to the year 2000, use Date$() exclusively so that the full year is returned in case you still use your program when the century turns.

---

Time$() returns the system time in a string data type in the following 24-hour format:

hh:mm:ss

where *hh* is the hour (from 00 to 23), *mm* is the minute (from 00 to 59), and *ss* is the second (from 00 to 59). Time() returns the system time in a variant data type in the following 12-hour format:

*hh*:*mm*:*ss* [AM] [PM]

where Time$() always returns either AM or PM after the time to indicate the time of day.

***Stop and Type:*** Listing 14.1 contains several assignment statements whose remarks describe the date or time value returned from the functions.

***Review:*** The date and time functions return values set inside your computer so that you can use and display those values in your program. The function that you use, Now(), Date(), or Time() (or the Date$() and Time$() counterparts) return various date and time formats.

**Listing 14.1. Accessing time and date values.**

```
1: lblNow.Caption = Now() ' 7/9/97 07:48 PM

2: lblDate1.Caption = Date() ' 7/9/97

3: lblDate2.Caption = Date$() ' 07/09/1997

4: lblTime1.Caption = Time() ' 07:48:24 PM


5: lblTime2.Caption = Time$() ' 19:48:24
```

***Analysis:*** The assignment statements assume that the current date and time is July 9, 1997 at 7:48:24 in the evening. The different formats offer you the choice of how you want the date and time values displayed in the target labels.

## Set the Date and Time From Visual Basic

***Concept:*** The Date and Time statements (not the functions) enable you to change the computer's date and time values from within a Visual Basic application. The date and time values remain set until you change them again, either through subsequent Date and Time statements or through DOS or Windows commands.

The DOS DATE and TIME commands enable you to check and set your computer's date and time settings. The Windows Control Panel program inside the Main program group also enables you to set these values from within Windows. Be sure to check your computer's date and time every month or so to make sure they're accurate.

Visual Basic includes the Date and Time statements that set the date and time values of your computer's clock and calendar. Be sure to keep the parentheses off the Date and Time statements, or Visual Basic will think that you're improperly using the corresponding functions.

Here are the formats of the Date and Time statements:

```
Date[$] = dateExpression
```

and

```
Time[$] = timeExpression
```

As with the corresponding functions, there are two versions of each statement, and the dollar sign distinguishes between the versions. If you don't specify the trailing dollar sign, you must enter the *dateExpression* as an unambiguous date value, and the *dateExpression* must be a string or date data type. All of the following set the computer's current date to July 9, 1998:

```
Date = 7/9/1997

Date = 07/9/97

Date = July 9, 1997

Date = Jul 9, 1997

Date = 9-Jul-1997

Date = 9 July 1997

Date = 9 July 97
```

The *dateExpression* must contain a valid date between January 1, 1980 and December 31, 2099, or Visual Basic generates an error.

If you do specify the trailing dollar sign (as in Date$), you can enter the *dateExpression* only in the following formats:

```
Date$ = 07-9-97

Date$ = 7-9-1998

Date$ = 7/9/97
```

```
Date$ = 7/9/1997
```

Owing to the many date formats, Visual Basic recognizes just about any way that you're used to specifying the date.

If you don't specify the trailing dollar sign, you can enter the *timeExpression* as either a 12-hour clock or a 24-hour clock with quotation marks, as follows:

```
Time = "7:48 PM"
```

or

```
Time = "19:48"
```

If you do specify the trailing dollar sign (as in Time$), you can enter the *timeExpression* in any of these formats:

```
hh
```

```
hh:mm
```

```
hh:mm:ss
```

You must use a 24-hour clock value when using *Time$*.

---

**Tip:** Using these Time$ formats, you change only what you want to change. If your time zone has just turned to daylight savings time, for example, you can change just the hour.

---

***Stop and Type:*** Listing 14.2 contains a section of code that enables the user to change both the date and time.

***Review:*** The Date, Date$, Time, and Time$ statements enable you to change the computer's internal date and time settings. Those settings remain in effect until you or the user changes them again.

**Listing 14.2. Letting the user change the date and time.**

```
1: Dim newDate As Variant

2: Dim newTime As Variant

3: MsgBox "The
current date is " & Date$ ' Calls function

4: newDate = InputBox("What do you want to set the date to?")

5: If IsDate(newDate) Then

6: Date$ = newDate

7: End If ' Don't do anything if a good date isn't entered

8: MsgBox "The
date is now " & Date$

9: MsgBox "The current time is " & Time$ ' Calls function

10: newTime = InputBox("What do you want to set the time to?")

11: If IsDate(newTime) Then

12: Time$ = newTime

13: End If ' Don't do
anything if a good time isn't entered

14: MsgBox "The time is now " & Time$
```

***Analysis:*** After defining two variant variables in lines 1 and 2 to hold the date and time values entered by the user, the program displays the date in line 3 and asks the user to change the date in line 4. Line 5 uses the IsDate() function, which you learned about in the previous unit, to check that the user entered a proper date value. If the user didn't enter a valid date, the program skips line 6 and displays the current set date without change in line 8.

---

**Note:** Visual Basic won't allow *any* invalid date to slip through. If the user were to enter 2-30-96, the IsDate() function in line 5 would know that February doesn't have 30 days and would not consider the date to be valid.

---

Lines 9 through 14 perform the same routine for the time.

## How Much Time Has Passed?

***Concept:*** Visual Basic includes a Timer() function that enables you to compute the elapsed time between two time periods. Using Timer(), you can find out how many seconds have elapsed since midnight.

The Timer() function returns the number of seconds that have elapsed since your computer's clock was last midnight. Timer() requires no arguments; hence, Visual Basic removes Timer()'s parentheses if you type them, but this book uses the parentheses, as is the standard, to remind you that Timer() is a function and not a command.

Assuming that it's 11:40 p.m., the following statement stores 85206.9 in the variable named TMid:

```
TMid = Timer()
```

Timer() returns a single-precision value. If you divide 85206.9 by 60, you'll get the number of minutes since midnight, and if you divide by another 60, you'll get the number of hours (a few less than 24) since midnight.

---

**Warning:** The Timer() function differs greatly from the timer control on the toolbox that you'll learn about in Lesson 10, "Making Programs Real World."

---

How can determining the number of seconds since midnight help you as a Visual Basic programmer? Timer() is perfect for timing an event. Suppose that you want to ask the user a question and determine how long it takes the user to answer. First, save the value of Timer() before you ask the user; then, subtract that value from the value of Timer *after* he or she answers. The difference of the two Timer() values is the number of seconds that the user took to answer.

***Stop and Type:*** Listing 14.3 contains code that asks the user for a math answer and times how long the user takes to respond.

***Review:*** The moment that your program executes the Timer() function, Visual Basic checks the computer's internal clock and returns the number of seconds since midnight.

**Listing 14.3. Test the user's math speed.**

```
1: Dim Before, After, timeDiff As Variant

2: Dim mathAns As String

3:
Before = Timer ' Save the time before asking

4: Do

5: mathAns = InputBox("What is 150 + 235?", "Hurry!")

6: Loop Until Val(mathAns) = 285

7: After = Timer ' Save the time after answering

8: ' The difference between the time values


9: ' is how many seconds the user took to answer

10: timeDiff = After - Before

11: MsgBox "That took you only" + Str$(timeDiff) & " seconds!"
```

***Output:*** Figure 14.1 shows the input box that asks the user for the answer.

**Figure 14.1. Test the user's math ability!**

***Analysis:*** Line 3 saves the Timer() setting right before the input box that requests the user's answer appears. The Do loop in lines 4 through 6 then keeps asking the user for the answer until the user enters the appropriate response of 285. As soon as the user enters the correct response, line 7 stores the Timer() value at that point. Line 10 then computes the difference of the two values to determine the number of seconds that the response took.

## Serial Dates and Times

***Concept:*** Visual Basic supports the storage of serial date and time values. A serial value is a number stored as a VarType 7 (the Date data type, as you learned in the previous unit). The serial number allows

you to break up dates into their day, month, and year values and allows you to break up times into their hour, minute, and second values.

---

*Definition:* A *byte* is one character of memory.

---

All the date and time functions that you've seen in this unit have been working with serial values. A serial value is the internal representation of a date or time, stored in a VarType 7 or a Variant data type. Visual Basic actually stores these values as double-precision values to ensure the full storage of time and date so that accurate date arithmetic can be performed. Visual Basic uses eight bytes memory of storage for double-precision values, which is enough room to hold the combined time and date value.

The following functions are explained in this section:

- DateSerial()
- DateValue()
- TimeSerial()
- TimeValue()
- Day()
- Month()
- Year()

All of these functions convert their arguments to serial date values. You then can use those serial date values to manage and modify specific parts of date and time values. Here is the format of the DateSerial() function:

```
DateSerial(Year, Month, Day)
```

where *Year* is an integer year number (either 00 to 99 for 1900 to 1999, or a four-digit year number) or expression, *Month* is an integer month number (1 to 12) or expression, and *Day* is an integer day number (1 to 31) or expression. If you include an expression for any of the integer arguments, you specify the number of years, months, or days from or since a value.

These two DateSerial() function calls return the same value:

```
d =
DateSerial(1990, 10, 6)
```

and

```
d = DateSerial(1980+10, 12-2, 1+5)
```

The DateSerial() function ensures that your date arguments don't go out of bounds. For example, 1992 was a leap year, so February of 1992 had 29 days. However, the following DateSerial() function call appears to produce an invalid date because February, even in leap years, can't have 30 days:

```
d = DateSerial(1992, 2,
29+1)
```

Nothing is wrong with this function call because DateSerial() adjusts the date evaluated so that d holds March 1, 1992, one day following the last day of February.

The DateValue() function is similar to DateSerial() except that the DateValue() function accepts a string argument, as the following format shows:

```
DateValue(stringDateExpression)
```

The *stringDateExpression* must be a string that Visual Basic recognizes as a date (such as those for the Date$ statement described earlier in this chapter and valid dates entered by the user). If you ask the user to enter a date one date part value at a time (asking for the year, then the month, and then the day separately), you can use DateSerial() to convert those values to an internal serial date. If you ask the user to enter a full date (that you capture into a string variable) such as **July 16, 1996**, DateSerial() converts that string to the internal serial format needed for dates.

The TimeSerial() and TimeValue() functions work the same as their DateSerial() and DateValue() counterparts. If you have three individual values for a time of day, TimeSerial() converts those values to an internal time format (the Variant or VarType 7). Here is the format of TimeSerial():

```
TimeSerial(Hour, Minute, Second)
```

The TimeSerial() function accepts expressions for any of its arguments and adjusts those expressions as needed, just as the DateSerial() function does.

If you have a string with a time value (maybe the user entered the time), the TimeValue() function converts that string to a time value with this format:

```
TimeValue(stringTimeExpression)
```

The Day(), Month(), and Year() functions convert their date arguments (of Variant or the VarType 7 data type) to a day number, month number, or year number. These three functions are simple. Here are their formats:

```
Day(DateArgument)

Month(DateArgument)

Year(DateArgument)
```

You often pass today's date (found with Now()) to the Day(), Month(), and Year() functions as shown here:

```
d = Day(Now())

m = Month(Now())

y = Year(Now())
```

The current date's day of week number, month number, and year are stored in the three variables d, m, and y, respectively.

***Stop and Type:*** Listing 14.4 contains a short section of code that asks the user for a date, uses the IsDate() function to ensure that the date is proper, and then uses the Month(), Day(), and Year() functions described here to break the date into its individual parts.

***Review:*** The serial date value that Visual Basic uses for all date and time values enable you to use various functions to break apart and piece back together various date and time combinations.

**Listing 14.4. Working with serial date values.**

```
1: Dim UserDate As Variant

2: Dim DStr As String

3:

4: ' Ask the user for a
date

5: Do
```

```
6: UserDate = InputBox("What's the date?")

7: Loop While Not IsDate(UserDate)

8:

9: DStr = "Month: " & Month(UserDate)

10: DStr = DStr + ", Day: " & Day(UserDate)

11: DStr = DStr + ", Year:
" & Year(UserDate)

12:

13: ' Display the date broken down

14: MsgBox DStr
```

***Output:*** Assuming that the user types **1/2/98** for the date in response to this code's InputBox() function call, Figure 14.2 contains the message box that shows the various parts of the dates.

**Figure 14.2. Through functions, you can break up a date.**

***Review:*** Lines 1 and 2 define two variables needed by the rest of the program. The variant UserDate variable will hold a date value entered by the user, and the DStr variable will hold a message box string.

Lines 5 though 7 loop until the user enters an appropriate date value. The loop continues as long as IsDate() reveals that the user's date value is invalid. The loop quits asking the user for a date when line 7 determines that the date is valid.

Lines 9 through 11 build a fairly complicated string displayed by line 14's message box function. The lines concatenate several strings together, building upon the DStr variable. The Month(), Day(), and Year() functions all pick off the month, day, and year values from the date entered by the user to display the different values inside the message box, as shown in Figure 14.2.

## Formatting with Format()

*Definition:* A *logical* value is the result of a relation or a True or False control value.

---

***Concept:*** The Format() function formats numeric and logical values so that you can display that data in the exact format that you require. Until this section, you couldn't ensure that currency values displayed to two decimal places, but Format() enables you to format currency and all other numbers the way you need.

Visual Basic can't read your mind, so it doesn't know how you want numbers displayed in your applications. Although Visual Basic sometimes displays none, one, or two decimal places for currency values, you'll almost always want those currency values displayed to two decimal places.

---

**Note:** As with the date and time functions, if you've set your computer's international settings to a country other than the U.S.'s (as done for this book), your formatted currency values may differ from those shown here. Some countries swap the use of commas and decimal places from those used in the U.S.

---

The two Format() functions are Format$() and Format() and they differ only in their return data types. Format$() returns a string and Format() returns a variant data type. Here is the format of Format():

```
Format$(Expression, FormatStr)
```

Often, you'll assign the result of the Format$() function to other variables and controls. Generally, you'll perform all needed calculations on numeric values before formatting those values. After you've performed the final calculations, you'll then format the values to string (or variant) data types and display the resulting answers as needed.

The *Expression* can be a variable, expression, or constant. The *FormatStr* must be a value from Table 14.1.

---

**Note:** Visual Basic contains many format strings in addition to the ones shown in Table 14.1. You can even develop your own *programmer-defined format strings*, although this book doesn't go into those.

---

---

*Definition:* A *thousands separator* is a decimal point or comma inside numbers over 999.

---

**Table 14.1. The fixed *FormatStr* values.**

| *FormatStr* | *Description* |
|---|---|
| "Currency" | Ensures that a dollar sign, $, appears before the formatted value followed by a thousands separator (the country setting determines whether the thousands separator is a comma or a decimal), and that two decimal places show. Visual Basic displays negative values in parentheses. |
| "Fixed" | Displays at least one digit before and two digits following the decimal point with no thousands separator. |
| "General Number" | Displays the number with no thousands separator. |
| "Medium Time" | Displays the time in 12-hour format and the AM or PM indicator. |
| "On/Off" | Displays On if the value contains a nonzero or True value and displays Off if the value contains zero or a False value. These values correspond to a special internal representation of computer values called *binary numbers*. |
| "Percent" | Displays the number, multiplied by 100, and adds the percent sign to the right of the number. |
| "Scientific" | Displays numbers in scientific notation. |
| "Short Time" | Displays the time in 24-hour format. |
| "True/False" | Displays True if the value contains a nonzero or True value, and displays False if the value contains zero or a False value. |
| "Yes/No" | Displays Yes if the value contains a nonzero or True value, and displays No if the value contains zero or a False value. |

---

You'll Rarely Need Format Codes: If the predefined formats from Table 14.1 don't match the format you need, you can define your own using special formatting codes. This unit would be at least twice as long as it is if all the programmer-defined formats were shown here. The good news is that, when you do define your own formats, you'll almost always use just a combination of the pound sign and zeros to format the values you need.

Each pound sign in the format indicates where a digit goes, and the zero indicates that you want either leading or trailing zeros. The following assignment displays the value of Weight to three decimal places:

```
lblMeas.Caption = Format$(Weight, "######.000")
```

You could also request that *no* decimal point should appear by formatting a fractional value such as Weight, and Visual Basic will round the number as needed to fit the target format. The following assignment displays Weight with no decimal places shown on the screen:

```
lblMeas.Caption = Format$(Weight, "######")
```

---

***Stop and Type:*** Listing 14.5 contains a series of formatting function calls that convert numeric and logical values to formatted variant data types that you can display.

***Review:*** There are two formatting functions. Format$() returns a string and Format() returns a variant value. The Format$() function formats values so that the values look the way you want them to look. The predefined format strings make it easy to display numbers and logical values in the format that you require.

**Listing 14.5. Formatting numeric and logical values.**

```
1: Dim FormValue As String

2:

3: ' Change 12345.678 to $12,345.68

4: FormValue1 = Format$(12345.678, "Currency")

5:

6: ' Change 12345678 to 12345.68

7:
FormValue2 = Format$(12345.678, "Fixes")

8:

9: ' Change .52 to 52.00%

10: FormValue3 = Format$(.52, "Percent")

11:

12: ' Change 1 to Yes

13: FormValue4 = Format$(1, "Yes/No")

14:

15: ' Change 0 to No

16: FormValue5 =
Format$(0, "Yes/No")
```

```
17:

18: ' Change 1 to True

19: FormValue6 = Format$(1, "True/False")

20:

21: ' Change 0 to False

22: FormValue7 = Format$(0, "True/False")
```

***Analysis:*** Line 4 converts the decimal single-precision number to a formatted currency string. If you were to display FormValue1 in a label's caption, the caption would show $12,345.68. Line 7 formats the same number for a noncurrency value with two decimal places. Line 10 changes a fractional number to its corresponding decimal percentage value and adds the percent sign after the formatted percent to show the percentage format. Line 13 changes the nonzero value of 1 to Yes. You can format any nonzero or relational result to a Yes, No, True, or False value, as shown in the program's remaining lines, so the values appear exactly the way you want them to look.

## Homework

### General Knowledge

1. True or false: Now() returns information for *both* the current date and time.
2. Where do the time and date functions get their values?
3. What is the difference between 12-hour time and 24-hour time?
4. True or false: 9:23 can be either 12-hour or 24-hour time.
5. True or false: 9:23 a.m. can be either 12-hour or 24-hour time.
6. What is the 24-hour time value for 7:54 a.m.?
7. What is the difference between the Date$() and Date() functions?
8. True or false: Visual Basic supports both a Now() and a Now$() function.
9. Does Now() return its time value using a 12-hour or 24-hour clock?
10. True or false: The time and date functions ignore your international time and date settings.
11. True or false: Date() and Time() set your computer's date and time.
12. Which function, Date() or Date$(), should you use if you think that your computer program will still be in use in the year 2000?

13. Describe the purpose of the Timer() function.
14. What is a *byte*?
15. What data type does a serial date value require?
16. How does the TimeValue() function relate to the Hour(), Minute(), and Second() functions?
17. What is a *logical value*?
18. What functions format numeric output for you?
19. What is the difference between Format() and Format$()?
20. What is a *thousands separator*?
21. What is the difference between the "Fixed" and "Currency" fixed-format strings?

## Find the Bug

1. Consider the user's input shown in Figure 14.3. The program is setting a time with this input box request:
2. Time$ = InputBox("What do you want to set the time to?")
3. Why do you suppose that Visual Basic indicates (as soon as the user presses OK in the input box) that an error occurred?

**Figure 14.3. Something's going to be wrong here.**

## Extra Credit

Write code that asks the user for the time that she or he clocked into work and then asks for the time that she or he clocked out. Display in three labels the total amount of seconds worked, the total number of minutes worked, and the total number of hours worked.

# Visual Basic in 12 Easy Lessons



# Project 7

## Functions and Dates

### Stop & Type

This lesson taught you all about Visual Basic's built-in numeric and string functions. These functions save you loads of programming time because you won't have to take the time or make the effort to write code that performs common tasks. There are numeric functions for number conversions and for scientific and for mathematical applications. The string functions manipulate and change strings to match the format you need.

In addition to the numeric and string functions, Visual Basic also includes a comprehensive set of date, time, and formatting functions that perform tasks unheard of in many other programming languages.

In this lesson, you saw the following:

- Why functions speed program development
- When to use one of the three integer conversion functions
- What math and scientific functions Visual Basic provides
- Which date and time functions to use when you need to extract or compute date and time values
- How the Format() function allows you to determine how output will look

### The Program's Description

Figure P7.1 shows the PROJECT7.MAK application as soon as you load and run the program. The project's form contains a frame with four option buttons and two command buttons.

**Figure P7.1. The project's opening screen.**

---

**Note:** The time zone calculations used in this project assume that you reside in the Central Standard Time zone. Of course, you may or may not live in the CST zone, but the program has to assume some kind of zone to base the time zone change calculations.

---

The program's option buttons select a different city for a local time display in the large time label inside the frame. The time updates only when you first run the program and when you click an option button.

The Birthday Fun command button calculates the year in which you can retire and go on that two-year cruise that you've always dreamed of.

## The Program's Action

Try this: Click through the various option button city selections to see the time zone changes, both in three U.S. time zones and also the Italian time zone change in Rome. The option buttons form a series of elements in an option button control array, and the index values for the option buttons help the code beneath the buttons (shown in the next section) determine which time zone to compute.

---

**Tip:** Before looking at the code for this project, try to determine how you would add or subtract to and from the current time zone, using Visual Basic code, to compute a different time zone. The practice will do you a lot of good and will help solidify the previous unit's time function discussion.

---

Click the Birthday Fun command button and you'll see the input box shown in Figure P7.2 that asks the user for his or her birthday. A subsequent message box then tells the user the year that the user can retire, based on a retirement age of 65.

**Figure P7.2. An input box gets the user's birthday.**

## The Time Zone Change Routine

Listing P7.1 contains the event procedure for the city time zone option buttons stored in the control array named optCity.

**Listing P7.1. Determining the time in different time zones.**

```
1: Sub optCity_Click (Index As Integer)

2: '
Adds or subtracts a value from the time

3: ' that matches the time zone selected

4: Dim DiffVal As Integer ' Hours to add or subtract

5:

6: ' Test the option button's index

7: Select Case Index

8: Case 0:

9: ' NYC

10: DiffVal = 1

11: Case 1:

12:
' Tulsa, the Central Standard Time default

13: DiffVal = 0

14: Case 2:

15: ' Los Angeles

16: DiffVal = -2

17: Case 3:

18: ' Rome
```

```
19: DiffVal = -7

20: End Select

21: lblTime.Caption = Format$(TimeSerial(Hour(Now) + DiffVal, Minute(Now),
Second(Now)), "Medium Time")

22: End Sub
```

## Descriptions

1: The option button array is named optCity, so the name of the Click event procedure is optCity_Click(). The index value also is passed to the event procedure so that the procedure knows which option button triggered the event.

2: A remark helps explain the purpose of the event procedure.

3: The remark continues that helps explain the purpose of the event procedure.

4: Define an array that will hold the number of hours to add or subtract to the current time to obtain a new time zone value.

5: A blank line to separate the variable definition from the rest of the code.

6: A remark helps explain the purpose of the subsequent Select Case.

7: Begin selecting the proper code to execute based on the index of the option button selected.

8: A match for the first option button (labeled New York).

9: A comment describing this case selection.

10: Add one hour to the current (Central Standard Time assumed) time value.

11: A match for the second option button (labeled Tulsa).

12: A comment describing this case selection.

13: No change to the current Central Standard Time value.

14: A match for the third option button (labeled Los Angeles).

15: A comment describing this case selection.

16: Subtract two hours from the current (Central Standard Time assumed) time value.

17: A match for the fourth option button (labeled Rome).

18: A comment describing this case selection.

19: Subtract seven hours from the current (Central Standard Time assumed) time value.

20: Terminate the body of the Select Case.

21: Update the caption with the new formatted time value. The TimeSerial() function allows you to specifically change only the hour.

---

21: With TimeSerial(), you have access to separate parts of the time.

---

22: Terminate the event procedure.

## Computing Retirement

When the user clicks the Birthday Fun command button, the program asks for the user's birthdate and calculates the retirement year based on the user's birthday. Listing P7.2 contains the event procedure for the retirement age calculation.

**Listing P7.2. Calculating the year of retirement.**

```
7: ' Filter out the birth year

8: BYear = Year(BDate)

9: ' Add the number of years to retirement

10: RetYear = BYear + 65

11: MsgBox "You can retire in" & Str$(RetYear)

1: Sub cmdBirth_Click ()

2: ' Asks the user for a
birthdate and determine retirement

3: Dim BDate As Variant

4: Dim BYear, RetYear As Integer

5: ' Get the birthday
```

```
6: Do

7: BDate = InputBox("When is your birthday (dd/mm/yy)?", "Birthday")

8: Loop Until IsDate(BDate) Or BDate =
""

9: If BDate <> "" Then ' If not Cancel

10: ' Filter out the birth year

11: BYear = Year(BDate)

12: ' Add the number of years to retirement

13: RetYear = BYear + 65

14: MsgBox "You can retire in" &
Str$(RetYear)

15: End If

16: End Sub
```

## Descriptions

1: The command button's Name property contains cmdBirth, so the name of the Click event procedure is cmdBirth_Click().

2: A remark explains the purpose of the procedure.

3: Define a variant variable that will hold the user's birthdate.

4: Define two integer variables. One will hold the user's birth year and the other will hold the year of retirement.

5: A remark that explains the purpose of the subsequent code.

6: Begin a loop to get a valid date.

7: Ask the user for his or her birthdate.

8: Keep looping until the user enters a valid date or until the user presses Cancel.

9: Perform retirement routine on date as long as long as the user doesn't press Cancel.

10: A remark that explains the purpose of the subsequent code.

11: Strip off the year of the birthdate and save the year.

---

11: Year() pulls the year from a date value.

---

12: A remark that explains the purpose of the subsequent code.

13: Add 65 to the birth year to find the year of retirement.

14: Display the retirement message.

15: Terminate the If statement.

16: Terminate the event procedure.

## Close the Application

You can now exit the application and exit Visual Basic. The next lesson explains how to build more complex programs by splitting common code that you write into several separate non-event procedures.

# Visual Basic in 12 Easy Lessons



# Lesson 8, Unit 15

## Subprograms

### What You'll Learn

- Introducing subprograms
- Subroutines: code routines
- External .BAS modules
- Functions procedures

By the very nature of the material, this unit presents more theory than many of this book's other units have done. Before you can move up to the next level of Visual Basic programming, you must master certain programming techniques needed for writing large-scale applications.

Until now, each unit has added to your Visual Basic language vocabulary. So far, the more you learned, the larger your event procedure got. This unit *reduces* the size of your event procedures! Starting in this unit, you'll see how to break up your programs into smaller and more numerous procedures than you've seen so far.

# Introducing Subprograms

***Concept:*** Break your programs into as many small but logical sections as possible. The smaller routines make your programming and subsequent maintenance easier.

In many traditional programming languages such as COBOL and FORTRAN, a program is like a long book without chapters: The code goes on and on and the program's length is exceeded only by the boredom programmers face trying to wade through the code hunting down errors. You've learned enough about Visual Basic so far to know that a Visual Basic program consists of a lot more than a long program listing. A Visual Basic program consists of the following:

- A form with controls that act as the program's background and user interface
- A general-purpose procedure named (general), found in the Code window's Object dropdown list
- Event procedures that tie the controls together and add specific direction and calculations to the application
- A CONSTANT.TXT file that provides named constants used by your code

---

**Note:** The (general) procedure in any program's Code window is often called the *declarations section*.

---

This unit will expand on the purpose of the (general) procedure and will explain how to add different kinds of procedures to your programs.

You now know two kinds of procedures: event procedures, which execute as events occur, and the (general) procedure. The (general) procedure is really not an executable procedure in the way that event procedures execute. The only statements that you can place in the (general) procedure are data definition statements such as Dim statements and option statements such as these:

```
Option Base 1
```

and

```
Open Explicit
```

Many Visual Basic programmers don't worry about using the Option Base 1 statement, which, as you learned in Lesson 6, specifies that all array subscripts will begin at 1 rather than 0 (the default). Rather than use Option Base 1, most programmers just ignore the first subscript of 0 and act as if the subscripts in all arrays begin at 1.

The Option Explicit statement is a helpful statement that you can place in the (general) procedure to tell Visual Basic to look for any undefined variables and to issue an error if Visual Basic finds any. By

requiring that you explicitly define all variables before you use them, misspelled variable names almost never cause the problem they would otherwise.

Assume that your program does *not* include the Option Explicit statement and you type the following in an event procedure:

```
Dim Sales As Currency

Sale = 294.43

lblOut.Caption = Sales ' Outputs
zero
```

Visual Basic outputs zero in the label because Sales contains zero, and the variable that you thought was named Sales, which you accidentally named Sale, holds the value of 294.43, which the user won't see.

If you were to add the Option Explicit statement in the (general) procedure (the only place you can put Option statements), Visual Basic would display the error message box shown in Figure 15.1, because Visual Basic correctly deduces that you have yet to define Sale but you're trying to store a value in Sale.

**Figure 15.1. Define all variables before using them or you'll get this error.**

---

*Definition:* Variable *declaration*, in Visual Basic, means the same thing as variable *definition*.

---

---

**Note:** The Options Environment menu displays a list of options that contain a Require Variable Declaration option that you can set to True. If a program's (general) procedure includes the Option Explicit statement, the program requires that all variables be defined before their use no matter how the Options Environment menu is set.

---

There are additional procedures that you can place in your programs. The general name for these procedures is *subprogram*. These procedures, just like the event procedures, are like small versions of a Visual Basic program. In a way, all procedures are the building blocks of the overall application code because all the procedures work together to comprise the complete application.

There are two kinds of Visual Basic subprograms: *subroutine procedures* and *function procedures*. Often, we abbreviate *subroutine procedures* to just *subroutines* and we abbreviate *function procedures* to *functions*. This may cause confusion due to the built-in functions such as Int() that you read about in the previous lesson. This book reserves the term *function procedure*, or just *function*, for the function procedures that you write, and this book refers to the functions supplied by Visual Basic as the *built-in functions*, as has been the case throughout the earlier lessons.

**Note:** Event procedures are actually specialized subroutine procedures. It's important, however, to distinguish between event procedures and the general-purpose subroutines you'll write that aren't tied to events. Therefore, this book will continue to refer to event procedures by that name, and this book will refer to subroutine procedures either *subroutine procedures* or just *subroutines*.

***Review:*** This section, as well as most of this unit, is concerned with getting the terminology straight that you'll read about and use throughout the rest of this book. Visual Basic programs are actually just small sections of code named *subprograms* that you write and that appear along with a (general) definitions section. There are two kinds of subprograms: subroutine procedures (which include event procedures) and function procedures. The rest of this unit explains more about these Visual Basic program divisions and also explains how to store subprograms in external files that more than one Visual Basic application can share.

## Subroutines: Code Routines

***Concept:*** A subroutine procedure always begins with the Sub statement and always ends with the End Sub statement. A subroutine procedure may or may not be an event procedure. Non-event procedures are general-purpose subroutines that you can write and add to any program.

Listing 15.1 contains an event procedure that you've seen in almost every program in this book.

**Listing 15.1. A common event procedure.**

```
1: Sub cmdExit_Click()

2: End

3: End Sub
```

*Definition: Wrapper lines* are lines of code that start and terminate procedure.

Lines 1 and 3, the wrapper lines, confirm that the Exit command button's Click event procedure is a subroutine procedure, as mentioned previously.

Event procedures are specific subroutines tied directly to control events. There will be times, many times in fact, when you'll write subroutines that aren't tied to any events whatsoever. When you write a section of code that your application will have to execute more than once, that section of code is a great candidate for a general-purpose, non-event subroutine.

*Definition:* To *call* a subroutine means to execute the subroutine from elsewhere in the program.

Suppose that your company has a specialized routine for calculating a cost of sales value. In writing a sales-computation data entry program, you find that you must execute this code several different places in the program. In other words, you never execute the routine over and over in a loop; instead, you may find that three or four different event procedures need to include this same calculation. Rather than type the exact code in three or four different places, type the code *just once* in its own subroutine and *call* that subroutine from each event procedure that needs the calculations.

Follow these steps to create a non-event subroutine procedure:

1. Make up a name for the procedure using the same naming conventions that you use for variables. Be sure to name the subroutine something meaningful. If you were computing cost of sales, the name CostOfSales would be a perfect name.
2. Open the Code window if it's not already open.
3. Press the PgDn key or click the Code window's vertical scroll bar to move the Code window down to the end of any event procedure. In other words, position the Code window's text cursor to the end any End Sub statement that you can find.
4. On a blank line below the End Sub statement, type the Sub statement followed by your new subroutine name. In other words, you would type **Sub CostOfSales** after any event procedure's End Sub statement.
5. As soon as you type the Sub statement, Visual Basic recognizes that you're starting a new subroutine. Visual Basic finishes the End Sub wrapper line and displays your new subroutine by itself in the Code window, as shown in Figure 15.2.
   If you were to open the Proc dropdown list, you would see the new CostOfSales general purpose subroutine among the list of procedures.

**Figure 15.2. Visual Basic gives your subroutine its own place in the Code window.**

**Note:** Notice that Visual Basic always adds parentheses after all subroutine names. Nothing you do will get rid of those parentheses. You'll use those parentheses in the next unit.

6. You can now insert the body of the subroutine in the middle of the two wrapper statements. As you add a line and press Enter, Visual Basic adds additional lines, so the subroutine grows as large as needed. Listing 15.2 contains a sample cost of sales subroutine that could be used for the procedure.

**Listing 15.2. The cost of sales subroutine.**

```
1: Sub CostOfSales ()
```

```
2: ' Computes a cost of sales and displays
3: ' that cost in the appropriate label
4: Dim GrossSales As Currency
5: Dim CostSales As Currency
6: Dim OverHead As Single

7: Dim InventoryFctr As Single
8: Dim PilferFctr As Single
9:
10: ' Store initial values from the form
11: GrossSales = txtGross.Text
12: InventoryFctr = txtTotalInv.Text * .38
13: PilferFctr = txtPilfer.Text
14: OverHead = .21 ' Fixed overhead

15:
16: CostSales = GrossSales - (InventoryFctr * GrossSales)
17: CostSales = CostSales - (PilferFctr * GrossSales)
18: CostSales = CostSales - (OverHead * GrossSales)
```

```
19: lblCost.Caption = Format$(CostSales, "Currency")

20: End
Sub
```

**Figure 15.3. You can use the New Procedure dialog box to open a new procedure.**

Although the body of the cost of sales subroutine is only 18 lines long, you certainly don't want to type those same 18 lines in every event procedure that needs to execute those 18 lines. If you've typed the code in its own subroutine as shown in Listing 15.2, you never have to type that code again. Instead, you'll just call that subroutine from every place in the program that needs the code executed.

---

**Tip:** You can take a shortcut when you want to open a new procedure. Select the View New Procedure from the menu bar. Visual Basic displays the small New Procedure dialog box shown in Figure 15.3. Clicking Sub opens a new subroutine procedure, and clicking Function opens a new function procedure (described later in this unit). Type the new procedure name in the Name text box and click OK. Visual Basic will open the new procedure and type the wrapper lines for you.

---

The Call statement executes subroutines. Here is the format of Call:

```
[Call] subName [(argumentList)]
```

Unit 16 describes the use of the *argumentList*. Not all procedures will contain an argument list. Call is optional, but if you do use the Call keyword when you call subroutines, you also must include the parentheses. Just remember: If no Call, no parentheses; if Call, use parentheses. Either way, the argument list may or may not be required.

---

**Tip:** You can move from procedure to procedure inside the Code window using the PgUp and PgDn keys. If you press F2, the shortcut access key for the Window Procedures command, Visual Basic displays a list of procedures; select a procedure and Visual Basic takes you to that procedure in the Code window.

---

You'll learn about the argument list in the next unit. For now, concentrate on the use of Call. Whenever any procedure in your program needs to trigger the execution of the CostOfSales subroutine, you'll only need to code the following Call statement:

```
Call CostOfSales () ' Executes all of the subroutine
```

The following statement is equivalent because, without Call, you don't need the parentheses:

```
CostOfSales ' Executes all of the subroutine
```

If three different procedures need the cost of sales routine calculated, isn't it easier (and less error-prone) to type the code *once* in its own subroutine and then simply call that subroutine from every place in the program that needs it? You can call subroutine procedures from event procedures as well as from other non-event subroutine procedures. Figure 15.4 shows the program flow when several event and subroutine procedures call the CostOfSales procedure. When one of the calling procedures calls CostOfSales, the CostOfSales takes over, executes, and the program's execution flow continues where it left off before taking time off.

**Figure 15.4. The subroutines work like detours inside your program.**

The Call statement also executes event procedures. You can, even if the user didn't trigger an event, trigger any event from within the code by using Call to execute the event procedure. Suppose that the Exit command button performs several end-of-program tasks such as erasing the form, asking the user's permission to exit inside a message box, writing the last of data files to disk, and printing a final report on the printer. If you find that you need to perform a program exit even if the user has yet to click Exit, you can call the cmdExit_Click() event procedure yourself.

---

**Note:** If you ever need to exit a subroutine procedure before the procedure's normal termination for example if the user cancels a subroutine input box use the Exit Sub statement.

---

***Review:*** When the same set of code lines needs to appear in more than one location in your program, consider putting those lines in their own subroutine procedure. You'll need to start a new procedure only if you use the Sub" or select the New Procedure command from the View menu. Place a Call to the subroutine procedure in all other program locations where you want the subroutine code to execute. You can call either a subroutine procedure or an event procedure using Call. When Visual Basic completely executes the procedure, Visual Basic returns execution to the code that performed the Call, and execution resumes at the statement following the Call.

## External .BAS Modules

***Concept:*** Over time, you'll write several routines, such as the previous section's CostOfSales subroutine procedure, that you will need in more than one application. Rather than write the same subroutine procedure in every application, you can store the subroutine procedure in a module file, along with other common procedures that you write, and add that file to any application that uses the file's procedures.

---

*Definition:* A *module* contains a form and the procedures used by that form.

---

Every program that you've seen so far in this book has consisted of single modules. A module is a collection of all event procedures, subroutine procedures, and function procedures, along with a form holding the controls. With the Visual Basic Primer system, you can add additional code modules to your form's module. A code module is an external file, stored separately on the disk, that contains the code of procedures that you write. These procedures are general-purpose procedures that perform calculations on data that you'll soon learn to pass to and from those procedures.

The CONSTANT.TXT file is a specialized kind of module that contains only a (general) procedure. Although you could add additional procedures to CONSTANT.TXT, you should neither add nor take away from the contents of CONSTANT.TXT. With each version of Visual Basic that Microsoft releases, Microsoft updates the contents of CONSTANT.TXT, and you'll want to replace the old CONSTANT.TXT file with the new one when and if you upgrade to a different version of Visual Basic.

You can add additional code modules to your program by selecting File New Module (or by clicking the New Module toolbar button, the second toolbar button from the left). As soon as you request a new module, Visual Basic opens a new Code window for that module. Visual Basic automatically names the first module that you add MODULE1.BAS (adding that file to the Project window's file list for that application). If you were to add a second module, Visual Basic would name that module MODULE2.BAS, and so on. Figure 15.5 shows the new Code window that opens when you add a new module to an application.

**Figure 15.5. The new Code window from the MODULE1.BAS external module file.**

Once you add code to a module file, you should save the file under a name that's different from the default name. Keep the .BAS file extension, however. The .BAS filename extension is usually reserved in Visual Basic for the general purpose external module files that you'll write and use. Even though CONSTANT.TXT is an external module file that breaks the .BAS file-naming rules, CONSTANT.TXT is an external module file.

The Project window for each application reflects which form file and module files (as well as CONSTANT.TXT if you use it) that each application requires. When you create a module, you're creating a stand-alone file on the disk that more than one application can use as long as you add that module to the applications' Project windows.

---

**Note:** Remember that the Project window doesn't hold files. The Project window holds only an application's list of needed files. You'll have to ensure that you distribute your Visual Basic applications with all files listed in each application's Project window (described in the .MAK file) if you write and distribute your Visual Basic applications to others.

---

Suppose that you wrote several subroutine procedures that printed your company's name and address, calculated city sales taxes for sales in all of your division's 20 regions, and made a backup of data files if the date hits the first of the month. You may find that, although you originally wrote these procedures for a general ledger system that you were writing, you need those same procedures in other programs. You can open a new module from within the general ledger application's Code window, and cut and paste those procedures from the general ledger application to the new module. If you then saved the module

under the name MYPROC.BAS, the general ledger application would be able to call that module's procedures (owing to the fact that Visual Basic added the module to the Project window when you first created the module), as well as *any other application to which you selected File Add and added the MYPROC.BAS module to.*

All programs automatically contain a form module. The module that you work in, using the Code window, that always displays when you open a form's or control's event procedure, is the form module. Additional modules that you add to the application, as well as the module supplied in CONSTANT.TXT, are sometimes called *non-form modules*.

***Review:*** Over time, you'll create module files when you run across useful procedures that you might need elsewhere. You can have as many modules on your disk as you want and add any or all of them to subsequent applications that you write. In a way, you're then building new applications faster by reusing code (through the Call statement) that you wrote for other applications and stored in the general-purpose modules.

## Functions Procedures

***Concept:*** As with the built-in functions, you can write your own general-purpose function procedures, often just called *functions*, that aren't tied to specific events. You can call these functions from your Visual Basic application. You can place these functions in their own external code modules or add them to an application's Code window. Function procedures work a lot like subroutine procedures; you can call them from elsewhere in the program. Unlike subroutine procedures, however, function procedures return values.

You can write your own function procedures to augment the built-in procedures supplied by Visual Basic. If you run across a needed calculation that converts a needed measurement, and you feel that you'll need that same conversion several places in your program, add that code to a function procedure. The function will be able to make the calculation and return the answer for the calculation to the calling routine.

As explained in the previous lesson, "Functions and Dates," when you call built-in functions, you must do something with the return value. You can't code the Int() function on a line by itself like this:

```
Int(Amount) ' Invalid!
```

Int(Amount) will return the amount converted to an integer, and you must do something with the returned integer. Therefore, you'll usually assign the return value like this:

```
lblAmt.Caption = Int(Amount)
```

or you'll use the function in a calculation that needs the value like this:

```
WholePart = Int(Amount) +
Estimate
```

You can write the code for your own numeric and string function procedures and add them to Visual Basic's repertoire. The functions that you write aren't quite as built in as Visual Basic's built-in functions because your functions don't become part of Visual Basic's language. Nevertheless, as with subroutine procedures, you can code the function procedures inside your application's Code window as well as store function procedures by themselves or along with subroutine procedures in external modules so that more than one application has access to the functions.

To write a new function procedure within a Code window, whether that Code window is your application form's Code window or an external .BAS module that you've opened, you can select the View New Procedure from the menu and click the Function option button on the New Procedure dialog box. (In the previous section, you clicked Sub to open a new subroutine.) After you type a new name for the function in the Name text box, Visual Basic adds the function wrapper lines like the ones shown here for a new function procedure named MultiplyIt:

```
Function MultiplyIt ()

End Function
```

A function procedure always begins with the Function statement and ends with the End Function statement. You'll add code to the body of the function between the two wrapper lines.

---

**Tip:** If you add a dollar sign after the function name, such as Reverse$(), Visual Basic assumes that you want to open a string function that will return a string value. If you omit the dollar sign, Visual Basic assumes that your function will return the Variant data type in which you can return numbers or string values. To keep things clear, always use the dollar sign for string functions.

---

A function is the same in every way as a subroutine except that the function returns a value. To return the function's value, assign the return value *to the name of the function*. Don't use Call to call a function. All you do to call a function is to use the function procedure's name inside an expression or statement.

---

**Note:** If you ever need to exit a function procedure before the function's normal termination for example, if the user cancels an input box use the Exit Function statement.

---

***Stop and Type:*** Listing 15.3 contains a function that computes the postage for a letter or package as follows:

1. The Post Office charges 32 cents for the 8 ounces.
2. Add 15 cents for each additional 4 ounces above the first 8.
3. The weight must be below 24 ounces.

Listing 15.3 assumes that the letter or package weight appears in a text box control named txtWeight.Text. In addition, the weight must appear as ounces. Therefore, any application that uses this function must make sure that a text box named txtWeight exists and holds the total package weight before calling the function.

***Review:*** When you want special routines that calculate or manipulate string values, and Visual Basic doesn't supply those routines through internal built-in functions such as CInt() and Mid$(), write your own function procedures. Before the function procedure terminates, assign the function's return value to the name of the function. Visual Basic uses this mechanism (the assigned name) to return the value assigned to the name back to the calling code that needs the result.

**Listing 15.3. A function that calculates postage and returns the cost.**

```
1: Function Postage ()

2: ' Calculate postage based on the

3: ' weight of a letter or
package

4: Dim PostHold As Currency

5: Dim weight As Integer

6:

7: ' Grab the weight value from the text box

8: ' and convert to number for comparison

9: weight = Val(txtWeight.Text)

10:

11: Select Case weight
```

```
12: Case Is <= 8:

13: PostHold
= .32

14: Case Is <= 12:

15: PostHold = .47

16: Case Is <= 16:

17: PostHold = .62

18: Case Is <= 20:

19: PostHold = .77

20: Case Is < 24:

21: PostHold = .92

22: Case Is >= 24:

23: MsgBox "Weight is more than 24 ounces!",
MB_ICONEXCLAMATION, "Error"

24: PostHold = 0

25: End Select

26:

27: Postage = PostHold ' Return the value

28: End Function
```

***Analysis:*** Line 3 defines a variable that will hold an interim postage amount throughout the Select Case. Actually, the extra PostHold variable is not strictly needed because each Case could assign directly to the function name. Nevertheless, assigning to the function name at the bottom of the function makes for better self-documentation of the code. When the function finishes, line 26 completes the function's task by assigning the calculated value to the function name.

---

Without Sharing, Your Code Is Limited: This unit's discussion of subroutines and functions can only be complete once you master the next unit's material on passing of data from procedure to procedure. Until now, a procedure could only work with data defined inside that procedure. The only exception was the function procedure that you just learned about, in which one procedure can use a value returned from another.
The next unit shows you how you can share variables between functions. The controls that you've been using have been available to all procedures in your program but not the variables. Only when you learn how to pass data from one procedure to another can you see the real power of subroutine and function procedures.

---

# Homework

## General Knowledge

1. True or false: A subroutine procedure is the same as an event procedure.
2. How do you call subroutines?
3. How do you call function procedures?
4. Name two ways that you can begin entering a new procedure in the Code window.
5. How can you determine whether a function procedure returns a string or a variant data type?
6. What is the primary difference between a subroutine procedure and a function procedure?
7. How do you indicate the return value of a function?
8. What is the shortcut access key to move from procedure to procedure?
9. How does a programmer-defined function procedure differ from a built-in function?
10. What is the name for a file that holds general-purpose procedures?
11. What filename extension should module files have?
12. Name a limitation of the procedures given your current knowledge so far. (Hint: Read the last sidebar in this unit.)
13. Name two statements that can appear in the (general) procedure.
14. True or false: You can call event procedures from other event procedures.

15. True or false: You can call event procedures from subroutine procedures.

## Write Code That...

1. Write the two wrapper lines for a subroutine procedure named PrReport().
2. Write the two wrapper lines for a function procedure named GetValue().
3. Suppose that you were writing a function named GetPi() that returned the value of the mathematical *pi* (which is approximately 3.14159). Write the statement that returned the value.

## Find the Bug

1. 19. Describe what's wrong with this Call:

   Call MySub arg1, arg2

## Extra Credit

Write a string-reversal function named Reverse$() that takes the string stored in the text box named txtAString and reverses the string, returning the string in the function name of Reverse.

# Visual Basic in 12 Easy Lessons



# Lesson 8, Unit 16

## Arguments and Scope

### What You'll Learn

- Three kinds of variable scope
- Global variables
- Module variables
- Local variables the safest variables
- Passing arguments
- Receiving two ways: by address and by value

Now that you can write subprograms including subroutine and function procedures stored in modules it's time to learn how those routines can communicate with each other. Until now, all variables that you've defined have been *local variables*. A local variable is known only within the procedure in which you

define the variable. Therefore, if one procedure calculated a variable and a second procedure needed to work with that same variable, the two procedures would have no way to share that variable in a way that both could work with the variable.

Until now, if two or more procedures had to work with a value, that value had to be a value from a control on the form. Controls are available to all procedures within the application. There won't always be controls for all values, however, especially the intermediate values needed in many large applications. Only those values displayed to the user should be stored in controls. The remaining intermediate values that you'll work with must stay in variables and be available to all procedures that work with those variables.

## Three Kinds of Variable Scope

***Concept:*** There are three kinds of variable scope: local, module, and global. This section explains how to define variables that fit in any of those three kinds of scope.

---

*Definition: Scope* determines how much of a program can access a variable.

---

You already know about all the data types that variables can take on. There are integers, strings, and decimal variables, as well as variant variables that can accept any data type. Not only do variables have names, contents, and data types, but variables also take on one of these three kinds of scope:

- Local variable scope: All variables that you define within a procedure have local variable scope. These variables are known as *local variables*. Local variables are usable and available only from code within their own procedures. The Dim, Static, and ReDim statements all define local variables. This book describes only Dim for local variables because of Dim's widespread use.
- Module variable scope: Each module, including your regular form module as well as any modules that you create or add to your program's Project window, has a (general) procedure in which you can define variables using Dim that have *module* scope. These variables are known as *module* variables. Module variables are usable and available to all procedures within that module. Therefore, if you define variables in the (general) procedure of a module file named MYPROCS.BAS, those variables are module variables, and any procedure within that module can access those variables. Nevertheless, procedures within the form's regular Code window module can't access the other module's variables.
- Global variable scope: Variables that you define using the Global statement inside the (general) procedure of any non-form module have *global* scope. These variables are known as *global* variables. When needed, this book will make the distinction between the form's module (the Code window that all forms provide) and additional modules that you add or create.

---

**Note:** If you were to look at CONSTANT.TXT, the module file that you added to AUTOLOAD.MAK, you would see that all of its named values are defined inside the (general) procedure. The location of

those definitions and their definition using the Global statement dictate that all values inside CONSTANT.TXT are defined at the module level. If Microsoft defined all of CONSTANT.TXT's named constant values from within a CONSTANT.TXT procedure other than (general), no other procedure could access the values within CONSTANT.TXT.

---

When you need to define variables, you should no longer automatically use the Dim statement. Now that you know how to write programs that contain several procedures, one or more of which might need to share data, you'll need to decide not only *how* you want to define the variables (using Dim or Global), but also *where* you want to define those variables. Your choice depends on the scope that the variables need. If a variable is used only within a single procedure, there is no reason to define that variable to have module or global scope.

***Review:*** The scope of data determines the extent to which other parts of the program can access that data. Controls always have global scope because any procedure in any of a program's modules can use control data. Variables defined in a module's (general) procedure using the Global keyword are also global and available from everywhere in the program. Variables defined in the (general) procedure using Dim are module variables and are available from any procedure in the module. Variables defined using Dim inside procedures (other than the (general) procedure) are local variables and are available only within that procedure.

## Global Variables

***Concept:*** The Global statement defines global variables. You can use Global only within a module's (general) procedure. Often, programmers add the Const keyword, as done throughout CONSTANT.TXT, to define named constant values that are variables whose values can't change.

The Global statement is similar to Dim. Here is the format of Global:

```
Global [Const] VarNa me [AS DataType] [=
value]
```

You can define variables only as global variables in the (general) declarations procedure of a module. Every procedure within that entire application can access the global variables. Therefore, a module's Form_Load() procedure can initialize a global variable that you defined, and another module's function procedure can access and change that same variable. If you use the Const keyword, you must also assign the variable an initial value but *not* use a data type. Without Const, you must specify the data type.

---

**Tip:** Define constant names that contain all uppercase letters. Throughout the program, you'll more easily be able to distinguish variable from constant names.

---

The following statement, appearing in the (general) procedure of any non-form module, defines an

integer global variable named MyGlobal. Any procedure within the entire application can initialize, access, and change the variable.

```
Global MyGlobal As Integer
```

The Const keyword enables you to name constants that you'll use throughout the rest of the program. For example, if your company's number of divisions is eight, you might want to define a global named constant like this:

```
Global Const NUMDIVS = 8
```

A named constant can *never* be changed, by user input, an assignment, or through any other kind of variable-changing statement for the rest of the program. NUMDIVS will always hold the value of 8. The advantage of using a named constant over the value of 8 everywhere in the program that needs to use the number of divisions is that, if the number of divisions change, you have to change the number in only one place, and the rest of the program automatically uses the new value. Also, the Const specifier keeps you or someone who modifies your code from accidentally overwriting the value elsewhere in the code. Figure 16.1 shows the error message box that Visual Basic displays if any line in the program attempts to change a named constant.

**Figure 16.1. Visual Basic reminds you if you attempt to change a named constant.**

The reason that you must specify an initial constant value is because the Global statement is the *only* place in a program where you can assign values to constants.

***Stop and Type:*** Listing 16.1 contains three global variable and two global named constant definitions. The code must appear in the (general) procedure of a non-form module or Visual Basic will issue an error message.

***Review:*** The Global statement enables you to define global variables. If you use the Const modifier to name global constants, you can't define the global's data type, but you must initialize the global constant at the time that you define the named constant.

**Listing 16.1. Defines five global values.**

```
1: Option Explicit

2:

3: ' Three global variables
```

```
4: Global G1
As Single

5: Global G2 As Double

6: Global G3 As Single

7:

8: ' Two named constants

9: Global Const G4 = 495.42

10: Global Const G5 = 0
```

***Analysis:*** Line 1 specifies that all variables within the module must be explicitly defined. The Option Explicit statement means that there's little chance that you'll misspell a variable name. Lines 4 through 6 define three global variables. Any procedure in the entire program, even procedures in other modules, can initialize, read, and change G1, G2, and G3. Lines 9 and 10 define two globally named constant values called G4 and G5. Both of the constants are defined in lines 9 and 10 because G4 and G5 can't be initialized anywhere else in the program.

## Module Variables

***Concept:*** Module variables are also defined in (general) procedures. Rather than use Global, you use Dim to define module variables. Module variables are available to all procedures within the module in which they are defined. Therefore, module variables are more limited in scope than global variables because they aren't available globally.

Module variables are considered slightly safer variables than global variables. Only the module in which you define module variables can access and change those variables. Therefore, technically, you could have two different modules in a single application, and each module could have the same module variable with the same name. Each module's module variables, however, would be separate variables. The code that accesses and changes the module variable within one of the modules changes only that module's variable.

---

**Caution:** Despite the fact that two modules within the same application can be named the same variable name without conflict, you should try not to duplicate variable names because of the maintenance confusion such double-defined variables can cause.

---

Unlike global variables, which you can define only in non-form modules, you can define module variables in the form's module or within any external module in your program. Generally, programmers rarely use module variables; most applications need either global values that are available through the application or local values that are specifically available only to individual procedures.

***Stop and Type:*** Listing 16.2 defines two module variables. The module variables must be defined in the (general) procedure of any module within the application.

***Review:*** Module variables, defined with Dim in the (general) procedure of any module, are available to any procedure within that module.

**Listing 16.2. Code that defines two module variables.**

```
1: Dim M1 As Integer

2: Dim M2 As Double
```

***Analysis:*** Lines 1 and 2 define two module variables named M1 and M2. The definitions must appear within some module's (general) procedure. The variables aren't global because Dim defines them instead of Global.

## Local Variables The Safest Variables

***Concept:*** Local variables are available only within the procedures in which they are defined. Use the Dim statements to define local variables. You may define local variables only within individual non-declarations procedures; you can't define local variables within the (general) procedure.

Global and module variables aren't considered as safe as local variables. Suppose that only three procedures within a large application need access to a variable. If you limit that variable's scope to those three procedures, you won't accidentally change those variables in other procedures by using the variables where you never intended to use them.

Most of your program's working variables should be local. You'll define those variables, as you've been doing throughout this book, and those variables aren't available outside their procedures. All of the Dim statements that you saw in this book, before this unit, defined local variables.

---

**Caution:** Local variables defined with Dim last *only* as long as the procedure's execution lasts.

---

When a procedure ends, its local variables defined with Dim go away and their values disappear. If execution were to re-enter that procedure, Visual Basic would define those variables all over again and initialize them again. Therefore, if execution enters a procedure a second time, the values of all local variables defined with Dim the last time that the procedure executed would no longer be in effect.

***Stop and Type:*** Listing 16.3 shows a command button's event procedure that defines a local variable and uses that variable and a module variable to compute a value for a label's caption.

***Review:*** When you declare a variable at the top of a procedure using Dim, you declare a local variable that Visual Basic recognizes only for the life of that procedure.

**Listing 16.3. Using a module and a local variable within a procedure.**

```
1: Sub cmdInvFactor_Click()

2: ' Adds an inventory factor to a module-level

3: ' inventory total when the user clicks the

4: ' inventory command
button

5: Dim FactAdd As Single

6: FactAdd = .13

7: ' Add to the module variable named InventoryTotal

8: lblInvent.Caption = InventoryTotal + FactAdd

9: End Sub
```

***Analysis:*** Line 1 begins a new event procedure. Therefore, any variable defined within the procedure, using Dim, must be a local variable. Line 5 defines the local variable named FactAdd. Line 8 then adds FactAdd to a module variable named InventoryTotal to compute the final label's result.

## Passing Arguments

***Concept:*** Local variables are considered the safest, and generally the best kinds of variables to define. However, there will be many times when a subroutine or function procedure needs a value from another's local variable. For example, suppose that one procedure calculates a value, and a second procedure must use that value in a different calculation before displaying a result on the form. This section explains how to pass local data *from* the procedure that defines the local variable *to* other procedures that need to work with that value.

When you call the built-in functions, you pass one or more arguments to the function so that the function's internal code has data to work with. When you call your own subroutine and function procedures, you also can pass arguments to them. The arguments are nothing more than the passing procedure's local variables that the receiving procedure needs to work with.

Once you pass data, that data is still local to the original passing procedure, but the receiving procedure has the opportunity to work with those values. Depending on how you pass the arguments, the receiving procedure might even be able to change those values so that when the passing procedure regains control, its local variables have been modified.

Figure 16.2 illustrates the terminology used in passing and receiving arguments from one procedure to another. A procedure, referred to as the *calling procedure*, passes one or more of its local variables, referred to as *arguments*, to the receiving procedure. The only reason that parentheses exist following a procedure name is to hold the arguments sent to the receiving function.

**Figure 16.2. Showing the proper procedure-calling terminology.**

---

**Note:** As you already know, if the receiving procedure is a function procedure, the function procedure usually returns a value to the calling procedure.

---

In Figure 16.2, the receiving arguments are I, J, and K. The receiving procedure uses the same names are the calling procedure in this figure, and that's usually the case. However, you don't have to use the same names. In other words, if RecProc() received the three variables as X, Y, and Z, then RecProc() would have three variables to work with named X, Y, and Z, and those three variables would have the same values as I, J, and K in the calling procedure. Therefore, the receiving names don't have to match the passed names, although they typically do to eliminate any confusion.

Declare the data type of all received arguments. If you must pass and receive more than one argument, separate the passed arguments and the received arguments (along with their declared data types), with commas. The following statement passes the three values to the subroutine in Figure 16.2:

```
Call RecProc(I, J, K)
```

The following statement begins RecProc() procedure:

```
Sub RecProc (I As Integer, J As Integer, K As Single)
```

The calling procedure already knows the data types of I, J, and K, but those values are unknown to RecProc(). Therefore, you'll have to code the data type of *each* received argument so that the receiving function knows the data type of the sent arguments.

If a subroutine or function procedure is to receive arrays, don't indicate the array subscripts inside the

argument list. The following Sub statement defines a general-purpose subroutine procedure that accepts four arrays as arguments:

```
Sub WriteData (CNames() As String, CBalc() As Currency,
CDate() As Variant, CRegion() As Integer)
```

The built-in UBound() function returns the highest subscript that's defined for any given array. The following statement, which might appear inside the WriteData() subroutine, stores the highest possible subscript for the CNames() array, so the subroutine won't attempt to access an array subscript outside the defined limit:

```
HighSub = UBound(CNames)
```

---

**Warning:** Don't forget that the Call statement is funny about the argument parentheses. If you use Call, you must also enclose the arguments in parentheses. You may omit the Call keyword, but if you do, omit the parentheses as well. Here is an equivalent Call statement as that shown in Figure 16.2:
RecProc I, J, K ' No Call, no parens!

---

***Stop and Type:*** Suppose that you're writing a set of programs for a bookstore's inventory and customer tracking purposes. The owners of the bookstore require that the user enter a category code that describes the kind of item being tracked or bought, and the program will print the description of that category as the purchase is entered or as the item is tracked through inventory. In other words, you'll ask the user for a category code of 1, 2, 3, 4, or 5, and the program will return a description in a string message variable describing that category. The following Select Case would work:

```
Select Case CatCode

 Case 1:

 CatMessage = "Book"

 Case 2:

 CatMessage = "Magazine"

 Case 3:
```

```
 CatMessage = "Newspaper"

Case 4:

 CatMessage = "Writing Supplies"

 Case 5:

 CatMessage = "Software"

 Case Else:

 CatMessage = "The category code is in error"

End Select
```

The problem is that this lengthy Select Case is needed throughout the program and needed by several other Visual Basic programs that you're writing as well. You're much better off storing this code as a general-purpose function procedure such as the one shown in Listing 16.4, and calling the function from wherever the code is needed, like this:

```
CatDesc = DispCatCode$(CatCode) ' CatDesc is a string
```

The function procedure enables you to pass the category code number argument and return the string description that matches that argument. As long as you store the procedure in a module file, any application that you add the module file to will be able to call the function and receive the message.

***Review:*** By passing arguments between function and subroutine procedures, your procedures can share local data. If a procedure doesn't need access to another's local variable, you'll never pass that procedure the variable. By passing data only as needed, you'll be provided data access on a need-to-know basis; that is, only those procedures that need access to data get access. If all of your variables are global, any procedure could inadvertently change another's variable, and such logic errors are often difficult to find.

**Listing 16.4. A general-purpose function procedure that you can call from any other procedure.**

```
1: Function DispCatCode$(CatCode As Integer)

2: Dim
```

```
CatDesc As String

3: Select Case CatCode

4: Case 1:

5: CatMessage = "Book"

6: Case 2:

7: CatMessage = "Magazine"

8: Case 3:

9: CatMessage = "Newspaper"

10: Case 4:

11: CatMessage = "Writing Supplies"

12:
Case 5:

13: CatMessage = "Software"

14: Case Else:

15: CatMessage = "The category code is in error"

16: End Select

17: DispCatCode = CatMessage ' Returns a description

18: End Function
```

***Analysis:*** The DispCatCode$() function could be stored in an external module file along with several other subroutine and function procedures that your applications often need. The module file acts like a toolchest with tools (procedures) that you can use (call) any time you need them.

Line 1 defines the procedure to be a function that receives one integer argument from the calling code. Line 2 defines a local string variable that the Case statements will use as a temporary storage location for the appropriate description. Lines 4 through 15 then test the passed integer to see whether the category code is 1, 2, 3, 4, 5, or another value that would indicate an error. Line 17 ensures that the appropriate message is returned to the calling procedure.

## Receiving Two Ways: By Address and By Value

***Concept:*** There are two ways to receive passed arguments: by address and by value. The method that you use determines whether the receiving procedure can change the arguments so that those changes remain in effect after the calling procedure regains control. If you pass and receive by address (the default method), the calling procedure's passed local variables may have been changed in the receiving procedure. If you pass and receive by value, the calling procedure can access and change its received arguments, but those changes don't retain their effects in the calling procedure.

Subroutines and functions can always use their received values and also change those arguments. If a receiving procedure changes one of its arguments, the corresponding variable in the calling procedure *is also changed*. Therefore, when the calling procedure regains control, the value (or values) that the calling procedure sent as an argument to the called subroutine may be different than before the call.

Arguments are passed by *address*, meaning that the passed arguments can be changed by their receiving procedure. If you want to keep the receiving procedure from being able to change the calling procedure's arguments, you must pass the arguments by *value*. To pass by value, precede any and all receiving argument lists with the ByVal keyword, or enclose the passed arguments in parentheses.

---

**Tip:** It's generally safer to receive arguments by value because the calling procedure can safely assume that its passed values won't be changed by the receiving procedure. Nevertheless, there may be times when you want the receiving procedure to permanently change values passed to it, and you'll need to receive those arguments by address.

---

***Stop and Type:*** Listing 16.5 shows two subroutine procedures. One, named Changes(), receives its arguments by address. The second procedure, NoChanges() receives its arguments by value. Even though both procedures multiply their arguments by ten, those changes affect the calling procedure's variables only when Changes() is called but not when NoChanges() is called.

***Review:*** The method by which you send and receive arguments determines how long a receiving procedure's changes stay in effect. If the receiving procedure changes one or more of its received-by-address arguments, those changes remain in effect when the calling procedure regains control. Therefore, when calling a procedure that accepts arguments passed and received by address, be aware that the passed values could be different when control returns to the calling procedure.

**Listing 16.5. One procedure receives by address and the other receives by value.**

```
1: Sub Changes (N As Integer, S As Single)

2: ' Receives arguments by address

3: N = N * 2 ' Double both

4: S = S * 2 ' arguments

5: ' When the calling routine regains control,

6: ' its two local variables will now be twice

7: ' as
much as they were before calling this.

8: End Sub

9:

10: Sub NoChanges (ByVal N As Integer, ByVal S As Single)

11: ' Receives arguments by value

12: N = N * 2 ' Double both

13: S = S * 2 ' arguments

14: ' When the calling routine regains control,


15: ' its two local variables will not be

16: ' changed from their original values.

17: End Sub
```

***Analysis:*** Line 1 defines the Changes() procedure to receive two arguments by address. Therefore, when lines 3 and 4 double those two values, the calling procedure's variables will be doubled also.

Line 10 defines the NoChanges() procedure that receives its two arguments by value. The ByVal keyword tells Visual Basic that no matter what happens to the arguments in the NoChanges() procedure, the calling procedure's values will be unaffected by the change.

The following statements would call these two procedures properly:

```
Call Changes(N, S)
```

```
Call NoChanges(X, S)
```

Again, the variable names in the calling procedures do not have to match the corresponding received names in the receiving argument lists. Therefore, the calling procedure sends its local variable named X to NoChanges(), which references that value as N inside the receiving procedure. If you want to send data to *either* procedure and ensure that the sent variables can't be changed even in the procedure that receives by address, enclose each sent argument in parentheses like this:

```
Call Changes( (N), (S) )
```

```
Call NoChanges( (X), (S) )
```

The special parentheses coding overrides any by address passing and ensures that the passed values will retain their original values after the called procedures finish.

## Homework

### General Knowledge

1. What is the name of the scope of a variable defined inside a procedure?
2. What is the name of the scope of a variable defined using the Dim keyword inside a module?
3. What is name of the scope of a variable defined using the Global keyword?
4. True or false: You can define a global variable inside the form's module.
5. True or false: You can define a module variable inside the form's module.
6. True or false: You can define a module variable inside a non-form module.
7. Which kind of scope offers the least amount of safety?

8. Which kind of scope is used the least?
9. How are the values in CONSTANT.TXT scoped?
10. What does the Const keyword do?
11. Why must you assign initial values to named constants when you define named constants?
12. Where must you define all named constants?
13. True or false: Your application can have two different variables with the same name.
14. What happens to local variables when their procedures end?
15. What is the name of the procedure that sends arguments to another?
16. What is the name of the procedure that receives arguments from another?
17. True or false: Built-in functions need no arguments.
18. Why must you specify data types for each argument in receiving argument lists?
19. True or false: You sometimes use the Call statement to call function procedures.
20. True or false: You can pass and receive at most one value to and from function procedures.
21. How many ways can a procedure receive variables?
22. Describe what the term *by address* means to the program.
23. Describe what the term *by value* means to the program.
24. Describe the two ways to ensure that arguments pass by value.
25. When receiving arrays, what function tells the receiving procedure the highest subscript value available for the array?

## Write Code That...

1. Define two global variables that you could use to hold a person's last name and age.
2. Define two global constants that hold the number of days in the week and the number of months in a year.

## Find the Bug

1. What's wrong with the following global variable definitions?
   Global Const X1 As Integer
   Global Const X2 As Integer = 19
   Global X3 = 19
   Global X4 As Integer = 19
2. Merle needs to write a function procedure that accepts an integer variable and a string array that contains 45 elements. Merle keeps getting an error with the following function definition

statement. Perhaps you can tell Merle what he needs to do.
Function Report(Age As Integer, CoNames(45) As String)

## Extra Credit

1. Write a general-purpose function procedure that accepts one string argument and returns a string that contains only every other letter of the passed string. Return a null string, "", if the passed string contains fewer than 2 characters.

# Visual Basic in 12 Easy Lessons



# Project 8

## Modular Programming

### Stop & Type

This lesson taught you how to break your programs into separate code components consisting of a form, subroutine procedures, function procedures, and external module files. The various pieces of the application work together to comprise the final program that the user sees when he or she runs the program.

By breaking your program into components, you can build upon past work. You can use subroutine procedures and function procedures that you've written before. You can define files of global constants just as Microsoft did by providing you with CONSTANT.TXT. When you reuse general-purpose code that you've written before, you save program development time and make debugging much simpler.

In this lesson, you saw the following:

- What subprograms are all about
- How subroutine procedures differ from function procedures
- When to use an external module file
- Why Visual Basic supports three kinds of variable scope

- How you can pass local values from one procedure to another

## The Program's Description

Figure P8.1 shows the PROJECT8.MAK Project window. As you can see, the project consists of a form file containing the form, controls, and code pertaining to the form such as all the controls' event procedures. The project also contains the CONSTANT.TXT named constant file as well as a file named PROJECT8.BAS, an external module file designed with this application in mind.

**Figure P8.1. Project 8's Project window contains three files.**

The purpose of this project's application is to use general-purpose procedures, named constants, an external module, and the controls with which you've worked before to tie together this lesson's material.

## The Program's Action

When you load and run PROJECT8.MAK, you'll first see an input box asking for a number from 1 to 100. A default value of 50 is used in case the user wants to press Enter to accept the default. The input box keeps displaying until the user enters a number between 1 and 100, or clicks Cancel or OK to get rid of the input box and accept the default value of 50.

After the user enters a valid number, Figure P8.2 shows the PROJECT8.MAK application's form that appears after the user enters a valid number. (The number entered by the user before Figure P8.2's form appeared was 47.)

**Figure P8.2. Project 8's form window.**

The program assumes that the user's number is a decimal base 10 number and that the number represents a Fahrenheit temperature reading. If you click the Base 16 option button, you'll see the hexadecimal equivalent to the user's number next to the Converted number: description. Click the Base 8 option button and the octal (base 8) representation of the user's number appears in the Bases frame.

The Temperatures frame changes the user's Fahrenheit temperature to Celsius and back again, depending on the chosen option button.

If the user clicks the Change Number command button, the user will be asked to enter a new number.

## The External Module's Code

Listing P8.1 contains the complete code listing for the PROJECT8.BAS external module. As you can see, the module contains global named constants as well as a general-purpose function that accepts a Fahrenheit single-precision value and returns the Celsius equivalent. You can add this external module

file to any application you write that needs to convert Fahrenheit temperatures to Celsius.

**Listing P8.1. The external module file contains the application's named constants.**

```
1: Option Explicit

2:

3: ' Define global constants for the application

4:

5: ' Number base constants that match Index values

6: Global Const BASE10 = 10


7: Global Const BASE8 = 8

8: Global Const BASE16 = 16

9:

10: ' Temperature constants that match Index values

11: Global Const CELSIUS = 0

12: Global Const FAHRENHEIT = 1

13:

14: Function GetCel (ByVal Faren As Integer)

15: ' Assumes that a
Fahrenheit temperature
```

```
16: ' is passed. Returns that temp as Celsius

17: GetCel = (Faren + 40) * (5 / 9) - 40

18: End Function
```

## Descriptions

1: Requires that all variables in the module be defined (or be arguments passed from elsewhere).

2: Blank lines help separate parts of code.

3: A remark helps explain the purpose of the (general) procedure.

4: Blank lines help separate parts of code.

5: A remark that explains how the subsequent named constants relate to Index subscripting properties of the two option button control arrays.

6: Define a name for the base option button control array item with 10 for an Index. Through the Property window, these three option buttons have Index property values of 10, 0, and 16.

---

6: Instead of coding subscripts, the rest of the program uses named constants.

---

7: Define a name for the option button control array item with 8 for an Index.

8: Define a name for the option button control array item with 16 for an Index.

9: Blank lines help separate parts of code.

10: A remark helps explain the purpose of the subsequent code.

11: Define a name for the temperature option button control array item with 0 for an Index. Through the Property window, these two option buttons have Index property values of 0 and 1.

12: Define a name for the option button control array item with 1 for an Index.

13: A blank line separates the (general) procedure from the function procedure.

14: The general-purpose function procedure begins, receiving a single argument by value.

15: A remark helps explain the purpose of the function.

16: The remark continues on the next line.

17: Calculate the return value by assigning the converted Fahrenheit temperature to the function name.

---

17: Always return a value when writing function procedures.

---

18: Terminate the function.

## The Form Module's Code

Listing P8.2 contains the complete code listing for the PROJECT8.MAK file that uses the PROJECT8.BAS external module.

**Listing P8.2. The form module's code controls the program flow.**

```
1: Sub Form_Load ()

2: Call GetUserNum ' Stored in module file

3: optbase(BASE10).Value = True

4: optTemp(FAHRENHEIT).Value = True

5:

6: ' Trigger the event procedures

7: ' for option button frames

8: Call optBase_Click(BASE10)

9:
Call optTemp_Click(FAHRENHEIT)

10: End Sub

11:
```

```
12: Sub GetUserNum ()

13: ' A subroutine procedure that gets a

14: ' number from 1 to 100 from the user

15: ' and displays the number on the form

16: Dim UserNum As Variant

17: Do

18: UserNum =
InputBox("Enter a number from 1 to 100", "Ask", "50")

19: If (UserNum = "") Then ' Check for Cancel

20: ' Restore previous value

21: UserNum = lblUserNum.Caption

22: Exit Sub

23: End If

24: Loop While (UserNum
< 1) Or (UserNum > 100)

25: lblUserNum.Caption = UserNum

26: End Sub

27:

28: Sub optBase_Click (Index As Integer)
```

```
29: ' Determines what base the converted

30: ' number displays in the base frame

31: Select Case Index

32: Case BASE10:

33: '
No change

34: lblBaseOut.Caption = lblUserNum.Caption

35: Case BASE16:

36: lblBaseOut.Caption = Hex$(lblUserNum.Caption)

37: Case BASE8:

38: lblBaseOut.Caption = Oct$(lblUserNum.Caption)

39: End Select

40: End Sub

41:

42: Sub optTemp_Click (Index
As Integer)

43: ' Determines what temperature appears

44: Select Case Index

45: Case CELSIUS:

46: lblTempOut.Caption = GetCel(Val(lblUserNum.Caption))
```

```
47: Case FAHRENHEIT:

48: lblTempOut.Caption = lblUserNum.Caption

49: End Select

50: End Sub

51:


52: Sub cmdChange_Click ()

53: ' Asks the user once again for the number

54: ' and calls appropriate click event

55: ' procedures to update two frames

56: Call GetUserNum

57: optbase(BASE10).Value = True

58: optTemp(FAHRENHEIT).Value = True

59: Call
optBase_Click(BASE10)

60: Call optTemp_Click(FAHRENHEIT)

61: End Sub

62:

63: Sub cmdExit_Click ()
```

```
64: End

65: End Sub
```

## Descriptions

1: Define the procedure that executes right before the user sees the form.

2: Call the subroutine procedure that asks the user for a number between 1 and 100.

3: Activate the Base 10 option button. When the form finally appears, the Base 10 option button will be selected.

4: Activate the Fahrenheit option button. When the form finally appears, the Fahrenheit option button will be selected.

5: A blank line helps separate parts of a program.

6: A remark explains subsequent code.

7: The remark continues.

8: Trigger a click event procedure for the option buttons with the Bases frame. This event initializes the frame.

---

8: Code can trigger event procedures.

---

9: Trigger a click event procedure for the option buttons with the Fahrenheit frame. This event initializes the frame.

10: Terminate the subroutine event procedure.

11: A blank line separates the Form_Load() procedure from the subroutine procedure that follows.

12: Define the subroutine procedure that requests a number from the user.

13: A remark explains subsequent code.

14: The remark continues.

15: The remark continues.

16: Define a variant variable that will capture the result of the input box.

17: Begin a loop that will ask the user for a number.

18: Display an input box and wait for the user's response.

19: Don't change the user's previously selected number (or the property defaults if this is the first time the user has been asked for a number) if the user clicks the Cancel command button.

---

19: Always check for the user's Cancel command button selection.

---

20: A remark explains subsequent code.

21: Put the number back to its current value.

22: Terminate the subroutine procedure early.

23: Terminate the If that checked for the Cancel command button press.

24: Keep asking until the user enters a valid number within the range required.

25: The user has entered a valid number in the input box, so display the result.

26: Terminate the subroutine procedure.

27: A blank line separates the GetUserNum() procedure from the event procedure that follows.

28: Define the event procedure for the Bases option button control array.

29: A remark explains the purpose of the event procedure.

30: The remark continues on the next line.

31: The Index argument can be one of three values: 10, 16, or 8, as set in the Properties window for the three Index values. Test the value in the index to determine which option button to respond to.

32: If the user clicked the Base 10 option button...

33: A remark describes the code that follows.

34: The label inside the Bases frame matches the user's entered number because both are base 10.

35: If the user clicked the Base 16 option button...

36: Display the user's entered number as an hexadecimal string.

37: If the user clicked the Base 8 option button...

38: Display the user's entered number as an octal string.

39: Terminate the Select Case.

40: Terminate the event procedure.

41: A blank line separates the two event procedures.

42: Define the event procedure for the Temperatures option button control array.

43: A remark explains the purpose of the event procedure.

44: The Index argument can be one of two values: 0 or 1, as set in the Properties window for the two Index values. Test the value in the index to determine which option button to respond to.

45: If the user clicked the Celsius option button...

46: Display the user's entered number as a Celsius temperature.

47: If the user clicked the Fahrenheit option button...

48: The label inside the Temperature's frame matches the user's entered number because both are considered to be in Fahrenheit.

49: Terminate the Select Case statement.

50: Terminate the end procedure.

51: A blank line separates the event procedures.

52: Define the event procedure that executes when the user clicks Change Number.

53: A remark explains the purpose of the event procedure.

54: The remark continues on the next line.

55: The remark continues on the next line.

56: Call the subroutine procedure that gets a number from the user. *No* arguments are required.

---

56: If the subroutine procedure requires no arguments, do not use parentheses.

---

57: Make the Base 10 Bases option button the default.

58: Make the Fahrenheit 10 Bases option button the default.

59: Trigger the event procedure that clicks the default Base 10 option button so that the Bases frame displays the user's number just entered.

60: Trigger the event procedure that clicks the default Fahrenheit option button so that the Temperatures frame displays the user's number just entered.

61: Terminate the event procedure.

62: A blank line separates the event procedures.

63: Define the event procedure for the Exit command button.

64: End the program when this event triggers.

65: Terminate the event procedure.

## Close the Application

You can now exit the application and exit Visual Basic. The next lesson explains how to add disk file access to your applications so that you can store and retrieve long-term data in disk files.

# Visual Basic in 12 Easy Lessons



# Lesson 9, Unit 17

## File Controls

### What You'll Learn

- Needed: file dialogs
- The file controls
- How FILESEL.MAK manages the file selection

This lesson describes how to access files inside Visual Basic applications. The data stored in files is more permanent than data stored in variables. The values of variables are too volatile. If you want to store a history of customer information, for example, you must store that information on the disk. If you kept such information in arrays of variables, you would never be able to exit Visual Basic or turn off the computer because you would lose all the information!

Data inside variables is needed for processing, calculating, printing, and storing, but not for long-term storage. Before you can access data stored in files, you must be able to locate data stored on the disk. This unit teaches you how to build common file-management dialog boxes that enable the user to select files stored on the disk. Depending on your application, you may need the user to select a filename, and this unit's file dialog box discussion provides the tools that you'll need to display lists of files from which the user can choose. After the user selects or enters a filename, you'll need to utilize the file I/O commands and functions discussed in the next unit to access the data in the files.

# Needed: File Dialogs

***Concept:*** There are several ways to obtain filename information from the user. The most common way is to build a dialog box that enables the user to select a file, directory, and drive from lists that you provide.

Figure 17.1 shows a common File Add dialog box that you'll see if you select File Add from Visual Basic. You've seen this and similar File Open dialog boxes in more than one Windows application. The great thing about file dialog boxes is that the user can select from a list of filenames, directories, and drives. Although you could ask the user for a filename using an input box, allowing the user to select from a list means that the user can often find files quickly and more accurately.

**Figure 17.1. A dialog box that enables the user to select the filename, directory, and drive.**

Dialog boxes are nothing more than secondary forms that you add to your application's primary Form window. The list boxes, command buttons, and text boxes that you see on such dialog boxes are nothing more than controls that you place on those secondary forms, just as you do on the primary Form window. We're going to have a small problem with the Visual Basic Primer System, however, that comes with this book: The Visual Basic Primer doesn't allow for more than one form per application. It would seem at first glance that we won't be able to add file-selecting dialog boxes to our applications.

Have no fear. The frame control produces frames on which you can place file-selecting controls such as list boxes, command buttons, and text boxes. The frame control won't offer *exactly* the same kind of usability as a true dialog box because the user can't resize or move the frame. However, the frame does provide basically the same end result that allows us display file-selecting controls when needed.

---

**Note:** Technically, you can write a frame control with properties that enable the user to move the frame, but most of the time, the user shouldn't have that much power over controls. The user could easily hide other controls and mess up the application's design.

---

Figure 17.2 shows the file selection frame that you'll learn how to create and use in this unit. As you can see, the frame offers the same functionality as the dialog box shown in Figure 17.1. After the user selects a file, you can program the frame to disappear from view just as file-selection dialog boxes disappear.

**Figure 17.2. A file-selection frame that mimics a file-selection dialog box.**

Actually, the frame file-selection dialog box is slightly easier to use than the standard file-selection dialog box because the frame combines the File Type box with the File Name text box. When the user enters a file pattern, such as *.frm in the File Name text box, the frame immediately updates the filename list box to reflect only files that match that pattern.

***Stop and Type:*** Load and run the project named FILESEL.MAK that comes on one of this book's companion disks (the disk that does not contain the Visual Basic Primer system). You'll see the application shown in Figure 17.3. Select different drives and directories and notice how the file list updates to reflect the change. The rest of this unit describes the important components of this application.

***Review:*** Although you can't add additional forms to your Visual Basic Primer applications, you can simulate the additional forms by using the frame control to hold sets of controls that mimic dialog boxes. This unit devotes itself to explaining the FILESEL.MAK project so that you'll know how to build such file-selecting frames for your user, and you'll know how to access the selected information after the user finishes selecting a file.

**Figure 17.3. This book's FILESEL.MAK project with the file-selecting frame on the screen.**

The next section, "The File Controls," explains how to use the file controls. Each of the file controls produces a different kind of file-access list box. Figure 17.3 includes callouts to the three list boxes produced by the file controls that you'll learn about in the next section.

***Analysis:*** The FILESEL.MAK application demonstrates how you might implement a file-selection frame. The application does nothing more than display the file-selection frame when the user clicks the Display File command button, and when the user selects a file, the application erases the frame and displays a new command button labeled You Selected... so that the user can see the file, path, and drive of the file selected from the frame earlier.

This particular application allows the user to select files but not enter new filenames. In other words, this application forms the basis for a frame that enables the user to open files that already exist. If you wanted to give the user the ability to enter a name not in the list, you would have to modify the application to look in the File Name text box for a new and valid file name.

The FILESEL.MAK program uses the frame's Visible property, setting the property to True for displaying the frame and to False when the frame must not appear on the screen.

## The File Controls

***Concept:*** Visual Basic includes three file controls that you manage from a file-selection frame.

Figure 17.4 shows the Toolbox window showing the location of the three file controls. As you know, selecting a file consists not only of choosing a filename but also choosing a directory and a drive. When you place these file controls on a frame, remember that you must draw each control by dragging with the mouse (explained more fully in Lesson 6's second unit, "Checks, Options, and Control Arrays") so that each control stays with the frame when code moves, hides, or displays the frame.

**Figure 17.4. The file controls on the Toolbox window.**

Using the file-related controls requires that you tie them together through code. In other words, when the user changes the drive name, both the path name and the file list must change also to show the paths and files on the newly selected drive.

Table 17.1 lists the properties that appear in the Properties window for the drive list box control. The properties are similar to the ones that you've learned for other controls. The drive list box control contains a list of disk drives that are available on the system running the program. As with any list box control, the user selects from the drive list box at runtime, so you can't set a default drive in the list until runtime.

**Table 17.1. The drive list box properties.**

| *Property* | *Description* |
|---|---|
| BackColor | Specifies the drive list box's background color, chosen as a hexadecimal color code or from the color palette. |
| DragIcon | Contains the icon that appears when the user drags the drive list box around on the form. (You'll only rarely allow the user to move a drive list box, so the Drag... property settings aren't usually relevant.) |
| DragMode | Holds either 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the drive list box but that you, through code, can initiate the dragging if needed. |
| Enabled | Determines whether the drive list box can respond to events. If set to True (the default), the drive list box can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | Holds True (the default) if the drive are to display in boldfaced characters; False otherwise. |
| FontItalic | Holds True (the default) if the drive names are to display in italicized characters; False otherwise. |
| FontName | Contains the name of the drive list box drive name's styles. Typically, you'll use the name of a Windows TrueType font. |
| FontSize | Holds the size, in points, of the font used for the drive names. |
| FontStrikethru | Holds True (the default) if the drive names are to display in strikethru letters (characters with dashes through them); False otherwise. |
| FontUnderline | Holds True (the default) if the drive names are to display in underlined letters; False otherwise. |
| ForeColor | Specifies the color of the letters in the drive names. |
| Height | Contains the height, in twips, of the drive list box. |
| HelpContextID | If you add advanced, context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| Index | If the drive list box is part of a control array, the Index property provides the numeric subscript for each particular drive list box. (See Lesson 6.) |
| Left | Contains the number of twips from the left edge of the Form window to the left edge of the drive list box. |
| MousePointer | Holds the shape that the mouse cursor changes to if the user moves the mouse cursor over the drive list box. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take. (See Lesson 12.) |
| Name | Contains the name of the control. By default, Visual Basic generates the names Drive1, Drive2, and so on as you add subsequent drive list boxes to the form. |

| | |
|---|---|
| TabIndex | Determines whether the focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, determines whether the user can press Tab to move the focus to this drive list box. If False, the drive list box can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the drive list box. |
| Top | Holds the number of twips from the top edge of a drive list box to the top of the form. |
| Visible | Holds True or False, indicating whether the user can see (and, therefore, use) the drive list box. |
| Width | Holds the number of twips wide that the drive list box consumes. |

At runtime, you can set the drive list box's Drive property. The Drive property will contain the name of the drive as well as the volume label on the drive. If you want only the drive letter, use the Left$() function to pick off the drive letter from the left of the drive's Drive property.

Table 17.2 lists the properties that appear in the Properties window for the file list box control. Many of the properties are similar to the ones that you've learned for other controls, but a few are important file-specific properties that you'll control using Visual Basic code.

**Table 17.2. The file list box properties.**

| *Property* | *Description* |
|---|---|
| Archive | If True (the default), specifies that the file list contains files whose internal archive (backed up) properties are set. If False, files whose archive bits are not set are not displayed. |
| BackColor | Contains the file list box's background color, chosen as a hexadecimal color code or from the color palette. |
| DragIcon | Holds the icon that appears when the user drags the file list box around on the form. (You only rarely allow the user to move a file list box, so the Drag... property settings aren't usually relevant.) |
| DragMode | Either contains 1 to indicate manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the file list box but that you, through code, can initiate the dragging if needed. |
| Enabled | If set to True (the default), determines whether the file list box can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FontBold | Holds True (the default) if the file names are to display in boldfaced characters; False otherwise. |
| FontItalic | Holds True (the default) if the file names are to display in italicized characters; False otherwise. |
| FontName | Contains the name of the file list box filename's styles. Typically, you'll use the name of a Windows TrueType font. |

| | |
|---|---|
| FontSize | Specifies the size, in points, of the font used for the filenames. |
| FontStrikethru | Holds True (the default) if the filenames are to display in strikethru letters (characters with dashes through them); False otherwise. |
| FontUnderline | Holds True (the default) if the filenames are to display in underlined letters; False otherwise. |
| ForeColor | Specifies the color of the letters in the filenames. |
| Height | Holds the height, in twips, of the file list box. |
| HelpContextID | If you add advanced, context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| Hidden | If True, specifies that the file list contains files whose internal hidden properties are set. If False (the default), files whose hidden properties are not set are displayed. |
| Index | If the file list box is part of a control array, the Index property provides the numeric subscript for each particular file list box. (See Lesson 6.) |
| Left | Holds the number of twips from the left edge of the Form window to the left edge of the file list box. |
| MousePointer | Contains the shape that the mouse cursor changes to if the user moves the mouse cursor over the file list box. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take. (See Lesson 12.) |
| MultiSelect | If MultiSelect contains 0-None (the default), the user can select only one filename. If 1-Simple, the user can select more than one filename by clicking with the mouse or by pressing the spacebar over filenames in the list. If 2-Extended, the user can select multiple filenames using the Shift+click and Shift+arrow to extend the selection from a previously selected filename to the current filename, and Ctrl+click either selects or deselects a filename from the list. |
| Name | Holds the name of the control. By default, Visual Basic generates the names File1, File2, and so on as you add subsequent file list boxes to the form. |
| Normal | If the Normal contains True (the default), the file list contains files whose internal normal properties are set. If False, files whose archive bits are not set to normal are not displayed. |
| Pattern | Contains a wildcard pattern of files. * in the pattern indicates all files, and ? means all characters. *.txt means all files whose extensions are txt, and sales??.dat means all files whose first five letters are sales, the sixth and seventh letters of the filenames can be anything, and the extension must be dat. |
| ReadOnly | Contains True (the default) if the file list is to contain files whose internal read-only properties are set. Therefore, the file list displays on those files that can't be changed. If contains False, files whose read-only properties are not set are not displayed. |
| System | Holds True if the file list contains files whose internal system properties are set. If False (the default), files whose system properties are not set are displayed. |
| TabIndex | Specifies the focus tab order begins at 0 and increments every time that you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |

| | |
|---|---|
| TabStop | Contains True if the user can press Tab to move the focus to this file list box. If False, the file list box can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the file list box. |
| Top | Holds the number of twips from the top edge of a file list box to the top of the form. |
| Visible | Contains either True or False, indicating whether the user can see (and, therefore, use) the file list box. |
| Width | Holds the number of twips wide that the file list box consumes. |

Table 17.3 lists the properties that appear in the Properties window for the directory list box control. Many of the properties are similar to the ones that you've learned for other controls. As with the file list control, you can set and select values from directory list boxes at runtime.

**Table 17.3. The directory list box properties.**

| *Property* | *Description* |
|---|---|
| BackColor | Specifies the directory list box's background color, chosen as a hexadecimal color code or from the color palette. |
| DragIcon | Specifies the icon that appears when the user drags the directory list box around on the form. (You'll only rarely let the user move a directory list box, so the Drag... property settings aren't usually relevant.) |
| DragMode | Contains either 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the directory list box but you, through code, can initiate the dragging if needed. |
| Enabled | Specifies that the directory list box can respond to events if set to True (the default). If set to False, Visual Basic halts event processing for that particular control. |
| FontBold | Holds True (the default) if the directory names are to display in boldfaced characters; False otherwise. |
| FontItalic | Holds True (the default) if the directory names are to display in italicized characters; False otherwise. |
| FontName | Contains the name of the directory list box directory name's styles. Typically, you'll use the name of a Windows TrueType font. |
| FontSize | Specifies the size, in points, of the font used for the directory names. |
| FontStrikethru | Holds True (the default) if the directory names are to display in strikethru letters (characters with dashes through them); False otherwise. |
| FontUnderline | Holds True (the default) if the directory names are to display in underlined letters; False otherwise. |
| ForeColor | Specifies the color of the letters in the directory names. |
| Height | Holds the height, in twips, of the directory list box. |
| HelpContextID | If you add advanced, context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |

| | |
|---|---|
| Index | If the directory list box is part of a control array, the Index property provides the numeric subscript for each particular directory list box. (See Lesson 6.) |
| Left | Holds the number of twips from the left edge of the Form window to the left edge of the directory list box. |
| MousePointer | Specifies the shape that the mouse cursor changes to if the user moves the mouse cursor over the directory list box. The possible values are from 0 to 12 and represent a range of different shapes that the mouse cursor can take. (See Lesson 12.) |
| Name | Contains the name of the control. By default, Visual Basic generates the names Dir1, Dir2, and so on as you add subsequent directory list boxes to the form. |
| TabIndex | Specifies that the focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, specifies that the user can press Tab to move the focus to this directory list box. If False, the directory list box can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the directory list box. |
| Top | Holds the number of twips from the top edge of a directory list box to the top of the form. |
| Visible | Holds True or False, indicating whether the user can see (and, therefore, use) the directory list box. |
| Width | Holds the number of twips wide that the directory list box consumes. |

---

**Note:** At runtime, you can set the directory list box's Path property. The Path property contains the name of the disk drive as well as the full path.

---

Most of the file-related controls are common and control the placement, size, and look of controls. However, you'll have to constantly update the values of key properties for these file-related controls as the user selects different values from the controls.

For example, when the user selects a different drive name, you'll have to update both the directory control's list of directories as well as the file list control's list of filenames. When the user selects a different directory, you'll have to update the file list control's list of filenames but not change the drive list control. Keeping these file controls in sync is the only unusual requirement for using these controls. The next section explains how the FILESEL.MAK application uses code to tie together its file controls.

---

**Note:** The events available for the file controls are extremely similar to those that you've seen. The most commonly coded event procedures are Click and DblClick. In addition, the drive and directory list boxes support the Change event procedure that executes when the user changes the drive or directory in some way.

---

***Review:*** Although many properties for the file-related controls are similar to those that you know from other controls, there are a few key properties that determine the selection from which the user can select filenames. The following properties are especially critical:

- The drive list box control's Drive property
- The directory list box control's Path property
- The filename list box control's Pattern property

The next section shows how the FILESEL.MAK application keeps these controls in synchronization while the user selects among drives, directories, and files.

## How FILESEL.MAK Manages the File Selection

***Concept:*** When one file selection control changes, such as the drive list box control, that change usually affects the other file controls. You must learn how to code file controls to keep them all pointing to the same drive, directory, and file.

When the user clicks the Display Files command button, the Click event procedure in Listing 17.1 executes to set up the default drive and path.

**Listing 17.1. The initial code for the file-selection frame's display.**

```
1: Sub cmdDisp_Click ()

2: ' Set default values

3: drvDrive.Drive = "c:"

4: dirList.Path = "c:\" ' Root directory

5:

6: ' Display the file-selection frame

7: lblDirs.Caption = dirList.Path

8: txtFiles.Text =
"*.txt"
```

```
9: fraFile.Visible = True

10: ' Select the first file

11: ' in the list of files

12: filFile.ListIndex = 0

13: End Sub
```

By setting the Drive and Path properties of the drive list box and the directory list box in lines 3 and 4, the command button sets up a default listing for the C: drive and the root directory on the C: drive. Line 7 initializes the label that shows the selected directory name above the directory box (directly beneath the Directories: label). Line 8 initializes the File Name text box with a default file pattern. A common filename extension for text files is txt, so the initial pattern is set to *.txt.

Line 9 does the work of displaying the frame to the user. During the FILESEL.MAK's development, the frame's Visible property was set to False, so the user doesn't see the frame or its controls until the user presses the command button. Finally, line 12 selects the first item in the file list box so that a filename is always selected in the list upon the initial display of the frame. If no files exist for the given drive, directory, and pattern, line 12 has no effect on the program.

---

**Tip:** Manage the drive change first.

---

When writing a file-selection frame, always code the drive change event procedure before anything else. When the user changes the drive, you must ensure that the directory list changes as well. Listing 17.2 contains the Change event procedure for the drive.

**Listing 17.2. Code that executes when the user selects a different drive.**

```
1: Sub drvDrive_Change
()

2: dirList.Path = drvDrive.Drive

3: End Sub
```

The drvDrive_Change() event procedure may not seem to do much. Actually, the purpose of the procedure is to trigger *another* event. As soon as the user changes the drive, Listing 17.2 works to update the path of the directory to the new drive's directory (the current default directory that's selected on the

newly chosen drive). Therefore, when the user selects a new drive, Listing 17.2 acts to update the directory list, *which in turn* triggers the directory's Change event procedure shown in Listing 17.3.

**Listing 17.3. When the directory changes, so must the file list.**

```
1: Sub
dirList_Change ()

2: ' The selection of the directory

3: ' list box changed

4: filFile.Path = dirList.Path

5: ' Make sure that one file is selected

6: If (filFile.ListCount > 0) Then

7: ' Select the first file

8: ' in the list of files

9:
filFile.ListIndex = 0

10: End If

11:

12: lblDirs.Caption = dirList.Path

13: End Sub
```

Line 4 updates the file list box to reflect the path of the new directory chosen. Lines 5 through 10 ensures that a file is selected, even if the user didn't select one. Line 9 assigns the selection to the first file in the file list. Line 12 updates the pathname caption that appears beneath the Directories label showing the current directory. The chain reaction is now tied together so that a change in the drive changes both the directory and the file list. If the user changes only the directory and not the drive, the change in the

directory (due to the dirList_Change() event procedure) changes also.

There is one Change event that's not tied to this chain-reaction of drive-directory-file change, but one that's important to maintain in all your applications. If the user wants to see a different set of files in the current directory, the user can enter a new file pattern in the File Name text box. As soon as the user enters a new pattern, the Change event procedure shown in Listing 17.4 executes.

**Listing 17.4. The user just changed the filename pattern.**

```
1: Sub
txtFiles_Change ()

2: filFile.Pattern = txtFiles.Text

3: End Sub
```

Any change in the File Name text box produces an immediate corresponding change in the list of files shown. Therefore, as soon as the user changes the default pattern from *.txt to *.exe, for example, the filename list box updates to show only those files whose extensions are .exe.

The user is allowed to close the file-selection dialog using any of the following three methods:

- Click the OK command button
- Double-click a filename
- Click the Cancel command button

Listing 17.5 contains the Click event procedure for the OK command button. When the user closes the file-selection frame, several things must take place. The highlighted filename needs to be saved. Line 3 saves the highlighted filename by placing that filename in the File Name text box. Even though the frame will be hidden (as well as all controls on the frame, including the text box) by line 5, the text box will be available to the program, and any routine in the application can access txtFiles.Text to see what filename the user selected. In addition, the drive and directory controls will still hold their selected values, even though the user won't be able to see these controls.

**Listing 17.5. The OK closing routine.**

```
1: Sub cmdFileOK_Click ()

2: ' Save the file selected by the user

3: txtFiles.Text =
filFile.List(filFile.ListIndex)
```

```
4: ' Hide the file selection frame

5: fraFile.Visible = False

6: ' Tell surrounding code that Cancel was not pressed

7: DidCancel = False ' User didn't press Cancel

8: ' Show the "You Selected" command button


9: cmdSel.Visible = True

10: End Sub
```

There is a module variable named DidCancel, defined in the form module's (general) procedure, that holds either True or False, depending on whether the user clicks Cancel. If Listing 17.5 executes, the user selected OK, not Cancel, so line 7 sets the DidCancel variable to False in case any application using this frame needs to know whether Cancel was pressed. Line 9 closes out the event procedure by hiding the file selection frame and all its controls.

If the user double-clicks a filename, the one-line DblClick event procedure shown in Listing 17.6 executes. The DblClick event procedure does nothing more than call the OK command button's Click event procedure because the same result occurs: the file selected by the user becomes the requested file and the frame disappears from the user's view.

**Listing 17.6. The user selected a file by double-clicking the filename.**

```
1: Sub filFile_DblClick ()

2: ' Double-clicking a selected filename

3: ' does the same thing as selecting a

4: ' file and pressing the OK button.

5: ' Therefore, call the OK button's Click event.
```

```
6: Call cmdFileOK_Click

7: End
Sub
```

If the user clicks the Cancel command button, nothing should change in the program except that the frame disappears and the DidCancel variable should be set to True. Therefore, if you add this frame and its event procedures to your own applications, the DidCancel variable tells you whether the user pressed Cancel. Listing 17.7 shows the Cancel command button's Click procedure.

**Listing 17.7. The user clicked Cancel to close the file-selection frame.**

```
1: Sub cmdCancel_Click ()

2: ' Used pressed the Cancel button

3: ' Hide the frame and inform the program

4: DidCancel = True

5: fraFile.Visible = False

6: End Sub
```

***Review:*** The code behind the file selection frame describes how the file controls work in conjunction with each other. A user's change to one of the file controls can affect the other file controls. Use code to control those additional changes so that a drive or directory change updates a corresponding file list box.

Please remember that if you adopt this file-selection frame for your own application, the disk drive appears at the start of the pathname. Therefore, you don't have to read the drive list box as long as you use the full path supplied by the path list box's Path property.

## Homework

### General Knowledge

1. Which is considered more long-term: Data in variables or data in files?
2. Why is using a dialog to get a filename from the user safer than using an input box to request the filename?

3. What is a dialog box?
4. How do you implement dialog boxes using the Visual Basic Primer system?
5. How do you mimic the display and disappearance of dialog boxes when you use frames?
6. How many controls on the toolbox work with files?
7. When can you set a default drive list box: at design time or runtime?
8. What does the Pattern file list control property contain?
9. Why must you maintain synchronization with a frame's file controls?
10. When a frame contains the file controls, which file-related control should you manage first when writing code to keep the controls in sync?
11. Which file controls should change when the user clicks the drive list box?
12. Which file controls should change when the user clicks the directory list box?
13. How does a calling application determine whether the user pressed the FILESEL.MAK file frame's Cancel button?
14. Which event procedure keeps a file list current when the user changes the selected pattern of files?
15. True or false: When you hide a frame, all controls on that frame disappear also.

## Write Code That...

1. Write a file list pattern than displays all files whose filenames begin with ACCT.
2. Write a file list pattern that displays all files whose filenames end with the extension ex1, ex2, ex3, and so on.
3. Write the code that sets a drive list named drvDisk to point to all files on the D: drive and that sets a directory list named direng to all files in the VBPRIMER directory.

## Extra Credit

Change the FILESEL.MAK to enable the user to select from files that already exist or enter a new file name. You'll have to add to the frame's closing code so that the program checks the text box entry against the selected file list.

# Visual Basic in 12 Easy Lessons



# Lesson 9, Unit 18

## Simple File I/O

### What You'll Learn

- Creating the disk file background
- Opening files
- Cleaning up with Close
- Writing to files with Write
- Inputting from files with Input#
- Making it really easy with Line Input#

---

*Definition: I/O* means input and output.

---

This lesson explains how you can use Visual Basic code to manage disk file I/O. If you've collected data from the user and stored that data in variables and arrays, you can save the data to the disk for later retrieval. Also, you can access disk files from within Visual Basic for product inventory codes, amounts, customer balances, and whatever else your

program needs from the long-term data file storage.

As you master Visual Basic and upgrade to other Visual Basic products, you'll add additional controls to your Toolbox window. There are several database access controls that read and write the data you've put in databases using products such as Microsoft Access and Paradox. Even though these controls provide more power and ease than you can get by programming alone, you'll still need the fundamentals of disk access. This unit explains a little on the background of disk access and teaches some of the most important disk commands and functions that you need to work with data files.

## Disk File Background

***Concept:*** A file is a collection of related data as well as programs that you buy and write, and documents from your word processor. Generally, you'll use Visual Basic to access data and text files stored on the disk.

There are all kinds of files on your computer's disks. Every file is stored under a unique filename to its directory and disk drive. Therefore, there can't be two or more files with the same filename unless the files reside in different directories or on different disks.

---

*Definition:* A *data file* holds data on the disk.

---

This unit is concerned with data files. Data files can take on all kinds of formats. Generally, newcomers to Visual Basic should stick with data files that are textual in nature. Text files are readable by virtually any kind of program, and virtually any program can produce text files. Sometimes, text files are called ASCII files because text files consist of strings of ASCII characters.

Before Visual Basic can access a file, you or the user will have to direct Visual Basic to the exact location on the exact disk where the file is stored. If your user is selecting a file, you'll want to use the file selection frame described in the previous unit to give the user the ability to change drives, directories, and filenames easily. When your program accesses a file that the user doesn't know about, your program will have to supply the drive, directory, and filename.

---

**Note:** The project at the end of this lesson contains an application that combines the file dialog frame that you mastered in the previous unit with the file I/O commands and functions described here to build a complete file access and display program.

---

***Review:*** This unit teaches you how to access text data files stored on the disk. You'll need to supply Visual Basic with the filename, directory, and disk drive of any file with which Visual Basic works.

## Opening Files

***Concept:*** The Open statement opens files. Before Visual Basic can access a data file, Visual Basic has to open the file first. Think of the Open statement as doing for Visual Basic what an open file drawer does for you when you want to retrieve a file from a filing cabinet. The Open statement locates the file and makes the file available to Visual Basic.

The Open statement performs various tasks such as locating a file, making sure that the file exists if needed, and creating some directory entries that manage the file while the file is open. A Visual Basic program always has to open

a file, using Open, before the program can read or write data to the file.

Here is the format of Open:

```
Open FileNameStr [For mode] As [#]FileNumber
```

The *FileNameStr* must be a string value or variable that holds a filename. The filename must reside on the default drive or directory unless you specify the full path to the file. Generally, you won't have easy access to the user's current Windows default drive and directory, so you'll almost always specify the full drive and pathname inside the *FileNameStr* portion of the Open statement.

The *mode* must be a named value from Table 18.1. There are additional *mode* values, but this book won't cover the more advanced or the esoteric *mode* values. The *mode* tells Visual Basic exactly what your program expects to do with the file once Visual Basic opens the file.

**Table 18.1. Possible *mode* values for the Open statement.**

| *Mode* | *Description* |
|---|---|
| Append | Tells Visual Basic that your program needs to write to the end of the file if the file already exists. If the file doesn't exist, Visual Basic creates the file so that your program can write data to the file. |
| Input | Tells Visual Basic that your program needs to read from the file. If the file doesn't exist, Visual Basic issues an error message. As long as you use a file selection frame properly, Visual Basic will never issue an error, because the file selection frame forces the user to select a file or cancel the selection operation. |
| Output | Tells Visual Basic that your program needs to write to the file. If the file doesn't exist, Visual Basic creates the file. If the file does exist, Visual Basic first erases the existing file and creates a new one under the same name, thereby replacing the original one. |

The pound sign, #, is optional, although most Visual Basic programmers do specify the pound sign out of habit (some previous versions of the BASIC language required the pound sign). The *FileNumber* represents a number from 1 to 255 and associates the open file with that number. After you open a file successfully (assuming that there are no errors such as a disk drive door being left open), the rest of the program uses file I/O commands and functions to access the file. The file number stays with the file until you issue a Close command (see the next section) that releases the *FileNumber* and makes the number available to other files.

---

**Note:** As with all DOS and Windows file descriptions, you can specify the drive, directory, and filename using uppercase or lowercase characters.

---

You can open more than one file simultaneously within a single program. Each command that accesses one of the files directs its activity towards a specific file using that file's *FileNumber*.

The following Open statement creates and opens a data file on the disk drive and associates the file to the file number 1:

```
Open "d:\data\myfile.dat" For Output As #1
```

If you knew that the file already existed and you needed to add to the file, you could use the Append *mode* to add to the file after this Open statement:

```
Open "d:\data\myfile.dat" For Append As #1
```

One Visual Basic program can have more than one file open at the same time. There is an advanced FILES option in your computer's CONFIG.SYS file that determines the maximum number of files that can be open at one time. If the #1 *FileNumber* was in use by another file that you opened earlier in the application, you could assign the open file to a different number like this:

```
Open "d:\data\myfile.dat" For Append As #5
```

Any currently unused *FileNumber* works; you can't associate more than one file at a time to the same *FileNumber* value.

The following Open would open the same file for input in a different program:

```
Open "d:\data\myfile.dat" For Input As #2
```

Visual Basic supplies a helpful built-in function named FreeFile() that accepts no arguments. FreeFile() returns the next available file number value. For example, if you've used #1 and #2 for open files, the next value returned from FreeFile() will be 3. FreeFile() is most helpful when you write general-purpose subroutine and function procedures that need to open files, and the procedures may be called from more than one place in an application. At each calling location, there is a different number of files open at the time. The procedure can store the value of the next available file number like this:

```
fNum = FreeFile()
```

and use the variable fNum in subsequent Open, I/O, and Close statements. No matter how many files are open, the procedure will always use the next file number in line to open its file.

***Review:*** The Open command associates files using file numbers with which the rest of the program will access the file. The three *mode* values determine how Visual Basic uses the file. If you want to write to a file, you can't use the Input mode, and if you want to read from a file, you can't use Output or Append.

## Clean Up with Close

***Concept:*** The Close statement performs the opposite job from Open. Close closes the file by writing any final data to the file, releasing the file to other applications, and giving the file's number back to your application in case you want to use that number in a subsequent Open statement.

Eventually, every program that opens files should close those files. The Close statement closes files. These are the two formats of Close:

```
Close [[#]FileNumber] [, ..., [#]FileNumber]
```

and

```
Close
```

The first format closes one or more open files, specifying the files by their open file numbers. The pound sign is optional in front of any of the file numbers. The second form of Close closes all files that are currently open. Close closes any open file no matter what mode you used to open the file.

---

**Tip:** If you create a file by opening the file with the Output *mode*, and then close the file, you can reopen the same file in the same program in the Input *mode* to read the file.

---

The following statement closes the two open files that were opened and attached to the 1 and 3 file numbers:

```
Close 1, 3
```

The following statement closes *all* files no matter how many are open:

```
Close ' Closes ALL files
```

***Review:*** Use the Close statement to close all open files before your program ends. Closing files provides extra safety, so close any and all open files when you're through accessing those files. If a power failure occurs during your program's execution, your closed files will be safely stored on the disk, but any files that are open at the time of the power failure could lose data.

## Writing to Files With Write

***Concept:*** The Write# command is perhaps the easier command to use for writing data to a file. Write# writes data of any data type to a file. Using corresponding input statements that you'll learn in the next section, you'll be able to read data that you sent to a file with the Write# command.

The Write# statement enables you to write data of any format to any disk file opened in the Output or Append mode. Write# writes strings, numbers, constants, variables, in any and all combinations to a disk file.

Here is the format of Write#:

```
Write #FileNumber [, ExpressionList]
```

The *FileNumber* must be a file number associated to a file opened with Output. If you don't specify variables or values to write, Write# writes a carriage return and line feed character (an ASCII 13 followed by an ASCII 10) to the

file, putting a blank line in the file. If you specify more than one value in the *ExpressionList*, Visual Basic writes that data to the file using the following considerations:

- Write# separates multiple items on the same line by adding commas between values
- Write# always adds a carriage return and line feed character to the end of each line written.
- Write# adds quotation marks around all strings in the file. The quotation marks make for easy reading of the strings later.
- Write# write date and time values using the following format:
  #yyyy-mm-dd hh:mm:ss#
- Write# writes the value of #NULL# to the file if the data is null (a VarType of 1).
- Write# writes nothing when the data value is empty (a VarType of 0), but does separate even empty values with commas if you write more than one value on a single line.

The code in Listing 18.1 writes a famous first century poem to a disk file named ODE.TXT. A command button named cmdWrite triggers the code's execution with its Click event when the user clicks the command button. The file appears on your C: drive's root directory.

**Listing 18.1. Writing four string values to a file named ODE.TXT.**

```
1: Sub cmdWrite_Click ()

2: ' Creates a text file and

3: ' writes a poem to the file

4: Open "c:\ode.txt" For Output As #1

5:

6: ' Write the poem

7: Write #1, "Visual Basic, Visual Basic,"

8: Write #1, "oh
how I long to see..."

9: Write #1, "A working application that"

10: Write #1, "means so much to me."

11:

12: ' Always close any open file
```

```
13: ' when done with the file

14: Close #1

15: End Sub
```

When Visual Basic closes the file in line 14, there will be a file in the user's root directory with the following contents:

```
"Visual Basic, Visual Basic,"

"oh how I long to see..."

"A working application
that"

"means so much to me."
```

Typically, quotation marks never appear when you assign or display string values. The quotation marks are usually for the programmer to indicate the start and end of string values used in a program. The only exception to the appearance of quotation marks is that Write# always adds the quotation marks around all string data, whether the data is variable or constant, that Write# outputs to a file.

---

*Definition: Append* means to add to the end of something.

---

If you open a file using the Append *mode*, the Write# statement will add to the end of the file. The program in Listing 18.2 writes a blank line and a second stanza to the poem stored in the ODE.TXT file.

---

**Warning:** Remember that if you write to a file that already exists using the Output *mode* value, Visual Basic will erase the original contents and replace the file with the subsequent writing.

---

**Listing 18.2. Code that adds to the end of a data file using the Write# statement.**

```
1: Sub cmdAddIt_Click ()

2: ' Adds to a text file already created

3: ' by writing a second verse to the file
```

```
4: Open "c:\ode.txt"
For Append As #1

5:

6: ' Add to the poem

7: Write #1, ' Writes one blank line

8: Write #1, "Visual Basic, to you I sing"

9: Write #1, "the songs a programmer knows..."

10: Write #1, "Listen carefully when I choose Run,"


11: Write #1, "or we'll surely come to blows."

12:

13: Close #1

14:

15: End Sub
```

I'll give you a chance to get a handkerchief before looking at the newly expanded poem. Here is the result of the Append *mode* value if you were to display the contents of the ODE.TXT file after running this event procedure:

```


"Visual Basic, Visual Basic,"

"oh how I long to see..."

"A working application that"

"means so
much to me."

"Visual Basic, to you I sing"

"the songs a programmer knows..."
```

```
"Listen carefully when I choose Run,"

"or we'll surely come to blows."
```

---

**Tip:** You may write data to files from variables as well as from controls on the form. Wherever you've got data that needs to be written, Visual Basic's Write# command will write that data to a disk file that you've opened.

---

***Stop and Type:*** Listing 18.3 contains a subroutine procedure that accepts four arrays of four different data types and writes that array data to a disk file named VALUES.DAT. Notice how a simple For loop can be used to write a large amount of data to a data file. The fifth argument sent to the subroutine is assumed to contain the total number of elements defined for the arrays.

***Review:*** The Write# statement provides one of the easiest file output commands inside the Visual Basic language. Write# separates multiple output values with commas and encloses all string data inside quotation marks.

**Listing 18.3. Code that writes arrays using the Write# statement.**

```
1: Sub WriteData (CNames() As String, CBalc() As Currency, CDate() As Variant,
CRegion() As
Integer)

2: ' Writes array data to a file

3: Dim ctr As Integer ' For loop control

4: ' Assumes that each array has the

5: ' same number of elements defined

6: Dim MaxSub As Integer

7: MaxSub = UBound(CNames) ' The maximum subscript

8:

9: ' Write
MaxSub lines to the file

10: ' with four values on each line

11: Open "c:\mktg.dat" For Output As #1
```

```
12: For ctr = 1 To MaxSub

13: Write #1, CNames(ctr), CBalc(ctr), CDate(ctr), CRegion(ctr)

14: Next ctr

15: Close #1

16:

17: End
Sub
```

***Output:*** Here are six sample lines from the MKTG.DAT file that the program in Listing 18.3 might write:

```
"Adams, H", 123.41, #1997-11-18 11:34:21#,
6

"Enyart, B", 602.99, #21:40:01#, 4

"Powers, W", 12.17, #1996-02-09#, 7

"O'Rourke, P", 8.74, #1998-05-24 14:53:10#, 0

"Grady, Q", 154.75, #1997-10-30 17:23:59#, 6

"McConnell, I", 9502.32, #1996-07-12
08:00:03#, 9
```

***Review:*** Line 1 defines the subroutine procedure and accepts the four passed arrays. Each array is assumed to have the same number of defined subscripts. Although not all arrays passed to a procedure would necessarily have the same number of subscripts defined, these happen to do so. Line 6 and 7 defines and initializes a value that holds the maximum number of subscripts, so the subsequent For loop can write all of the array values to the file names MKTG.DAT.

The For loop in lines 12 to 14 step through the array elements, writing a line of four array values with each iteration. Visual Basic encloses the string array data with quotation marks and encloses the variant date and time data with pound signs.

The pound signs around the date and time variant values help Visual Basic when you subsequently read the data values back into variant variables. As you can see, the date may have a missing time or the time may have a missing date. Write# still writes as much of the date and time as is available within that variant value.

The Close statement on line 15 closes the file and releases the file number back to the program.

# Inputting from Files with Input#

***Concept:*** The Input# statement reads data from files and stores the file data in your program's variables and controls. Input# is the mirror-image statement to the Write# statement. Use Input# to read any data that you send to a file with Write#.

The Input# statement reads data into a list of variables or controls. Here is the format of Input#:

```
Input #FileNumber [, ExpressionList]
```

Again, the bottom line to using Input# is that Input# is the mirror image of the Write# statement that produced the file data. When you write a program that must use data from a data file, locate the program's Write# statement that originally created the data file, and use that same format for the Input# statement.

---

**Tip:** Be sure to open all files that you'll read from using the Input file mode value, or Visual Basic will display an error message.

---

Listing 18.4 reads and displays in a scrolling list box the poem stored in the ODE.TXT file that was created in earlier listings.

**Listing 18.4. Code that reads a poem using the Input# statement.**

```
1: Sub cmdList_Click ()


2: ' Reads a poem from a text file and

3: ' adds the poem, one line at a time,

4: ' to the list box

5: Dim ctr As Integer ' For loop control

6: Dim PoemLine As String ' Holds each poem line

7: Open "c:\ode.txt" For Input As #1

8:

9: '
```

```
Read the poem

10: For ctr = 1 To 9

11: Input #1, PoemLine

12: lstPoem.AddItem PoemLine

13: Next ctr

14:

15: ' Always close any open file

16: ' when done with the file

17: Close #1

18:

19: End Sub
```

Figure 18.1 shows the list box that would appear as a result of Listing 18.4's execution.

**Figure 18.1. The contents of a file shown in a list box.**

---

*Definition:* A *record* is a row in a file.

---

When reading data from a file, you can very easily cause an error by attempting to read more data than the file holds. Listing 18.4 assumed that there were only nine records in the poem file to read, and that happened to be the case. For data files that hold data such as customer balances and employee pay values, however, the number of records varies because you'll add and remove records as transactions take place.

The Eof() function is Visual Basic's built-in *end of file* function that senses when an input reaches the end of file. Here is the format of the Eof() function:

```
Eof(FileNumber)
```

Eof() returns True if the most recent reading of the input file just reached the end of the file and returns False if the input file still has data left to be read. Most data input programs loop until the Eof() condition is true. Perhaps the best way to use Eof() is with a Do Until-Loop that follows this general format:

```
Input #1, VariableList ' Read the first record

Do Until (Eof(FileNumer) = True)

 ' Process the record just read

 Input #1, VariableList ' Get more
data

Loop
```

If there are none, one, or four hundred records in the file, this format of Do Until will keep reading, but will stop as soon as the end of file is reached. Many programmers often increment an integer counter variable inside the loop to count the number of records read. The counter is useful if you're reading the file's data into arrays.

---

**Note:** If you read file data into arrays, be sure to dimension more than enough array elements to hold the maximum number of records expected.

---

***Stop and Type:*** Listing 18.5 reads the file written earlier using a series of Write# statements back in Listing 18.3. The body of the code isn't shown. The code is supposed to output the file's contents to a printed paper report. You won't master the reporting commands until Lesson 11, so comments were used in place of the actual reporting commands.

***Review:*** As long as you output to files using Write#, you'll easily read that same data back again using Input#. Input# is the mirror-image command to Write#. The combination of Write# and Input# makes file I/O simpler than possible in most programming languages.

**Listing 18.5. Reading and reporting file data using Input#.**

```
1: Sub ReadData ()


2: ' Reads array data from a file and reports the data

3: ' Assume that 200 values were read

4: Static CNames(200) As String, CBalc(200) As Currency

5: Static CDate(200) As Variant, CRegion(200) As Integer

6: Dim NumVals As Integer ' Count of records


7: Dim ctr As Integer ' For loop control
```

```
8:

9: NumVals = 1 ' Start the count

10: ' Reads the file records assuming

11: ' four values on each line

12: Open "c:\mktg.dat" For Input As #1

13: Input #1, CNames(NumVals), CBalc(NumVals),
CDate(NumVals), CRegion(NumVals)

14: Do Until (Eof(1) = True)

15: NumVals = NumVals + 1 ' Increment counter

16: Input #1, CNames(NumVals), CBalc(NumVals), CDate(NumVals), CRegion(NumVals)

17: Loop

18:

19: ' When loop ends, NumVals holds one too many


20: NumVals = NumVals - 1

21:

22: ' The following loop is for reporting the data

23: For ctr = 1 To NumVals

24: ' Code goes here that outputs the array

25: ' data to the printer

26: '
```

```
27: Next ctr

28: Close #1

29:

30: End Sub
```

***Analysis:*** Lines 4 and 5 define four arrays that will hold the file's data. The arrays are defined to hold, at most, 200 values. There is no error checking to be sure that the reading doesn't read more than 200 records, but an exercise at the end of the chapter gives you the chance to add this array-checking code.

Line 9 initializes NumVals to hold the count of file records. The count begins at 1 because the first array subscript that will hold incoming data is 1. (As with all programs in this book, this code ignores the 0 subscript that all arrays contain unless you specify Option Base 1.)

Line 12 opens the file for input and associates the file to the file number 1. The first input occurs on line 13. The input attempts to read the first record in the file that contains four values. Line 14's relational test for an end of file condition will immediately be true if the file is empty. Otherwise, the body of the loop executes. Line 15 increments the record count once more owing to line 16's input into the next set of array elements. The loop continues until the end of file is reached and line 14's relational test becomes false.

Line 20 must decrement the NumVals variable by 1 because the last attempt at reading the file will always produce an end-of-file condition. Therefore, the final Input# was unsuccessful and the record count can't contain that unsuccessful read.

After you read all the data into the arrays, you can work with the array data just as you normally work with array data. You can calculate statistics based on the data, display the array using list boxes and combo boxes, or print the data to paper.

## Line Input# Really Makes it Easy

***Concept:*** The Line Input# command reads data from open data files. Unlike Input#, Line Input# reads each line of data in the file into a string variable. You don't have to specify separate variable names after a Line Input# because Line Input# requires a single string value. Line Input# reads data from any file whose lines end with a carriage return and line feed sequence. (Most files do.)

The Line Input# command is simple to use for reading entire records into a single variable. Whereas Input# reads each records values individually, Line Input# reads an entire record data, commas, quotation marks, and everything else into a string or variant variable or control.

Here is the format of Line Input#:

```
Line Input #FileNumber, VariableName
```

No matter how many record values appear in the file associated with file number 3, the following Line Input# statement reads an image of the record into the string variable named aRecord:

```
Line Input #3, aRecord
```

***Review:*** The Line Input# command reads records from data files into string or variant variables. The Input# command reads each record's individual data values into a single variable (or control). The Line Input# command reads entire records into string or variant variables. The project at the end of this lesson uses Line Input# to read an entire file using a *single* string variable.

# Homework

## General Knowledge

1. What is a file?
2. What are the purposes of files?
3. True or false: Your computer system can contain more than one file with the same filename.
4. What statement prepares a file for use by a program?
5. What are the three modes possible in the Open statement?
6. What is the purpose of the file number?
7. If a file doesn't exist and you open the file for output, what does Visual Basic do?
8. If a file does exist and you open the file for output, what does Visual Basic do?
9. If a file doesn't exist and you open the file for appending, what does Visual Basic do?
10. If a file does exist and you open the file for appending, what does Visual Basic do?
11. If a file doesn't exist and you open the file for input, what does Visual Basic do?
12. What is the range for the Open statement's file number?
13. True or false: When opening a file, you must specify the filename using lowercase letters.
14. What CONFIG.SYS command limits the number of files that you can have open at one time?
15. Which function locates the next available file number?
16. Which statement cleans up a file after you're done with the file?
17. Why is the Close command recommended?
18. Which open files does the following statement close?
19. Close
20. Which statement writes data to a file?
21. What character does Write# use to separate values in the output?
22. What character does Write# place around date and time values?
23. What character does Write# place around string values?
24. Which statement is the mirror-image input statement for Write#?
25. What is a *record*?
26. Which function tests for the end of file?
27. Which statement reads an entire record into a string or variant variable?

## What's the Output?

1. What does the file record look like after the following Write# completes?
   Write #1, "Peach", 34.54, 1, "98"

## Find the Bug

1. Rusty the programmer wants to read an entire record into a string variable named MyRecord, but his attempt shown as follows doesn't work. Help Rusty fix the problem.
   Input #1, MyRecord

## Write Code That...

1. Write a Close command that closes an input file associated with file number 3 and an output file associated with file number 19.
2. Write the Open command needed to open a file named NAMES.DAT for append mode. Associate the file to file number 7.
3. Change Listing 18.5 to display a message box and exit the file-reading loop if the input data file contains more than 200 records. (The arrays holding the incoming data can't hold more than 200 values.)

## Extra Credit

Write a subroutine procedure that writes four records of your four best friends' first names, ages, weights, and IQs (remember, these are your friends, so be gentle). Write a second procedure that reads that data into string variables one record at a time.

# Visual Basic in 12 Easy Lessons



# Project 9

# Extending Data and Programs

## Stop & Type

This lesson taught you how to create a file-selection frame that mimics a file dialog box as well as controls the file controls from the toolbox. You learned that your code must keep the file controls in synchronization with each other or problems will occur, such as the directory list box that points to a drive that doesn't contain the directory listed in the directory list box.

You learned how to use Visual Basic's file I/O commands and functions to read, append to, and write data files. Accessing data files is relatively simple.

In this lesson, you saw the following:

- Why you need a file dialog box for file selection
- How to manipulate the file controls to keep them in sync
- How to open files and associate files to a file number
- Why the Write# command is the perfect command for sending data to a data file in an easy-to-read format
- When to use Input# and when to use Line Input# to read data from files

### The Program's Description

Figure P9.1 shows the PROJECT9.MAK opening Form window. As you can see, the project starts off extremely simply by displaying three command buttons.

**Figure P9.1. Project 9's application begins with a simple form.**

Two controls, a file-selection frame and a large text box, reside on the form, but these controls are invisible to the user when the user first runs the program. When the user selects the Select a File command button, the file-selection frame that you learned about in Unit 17 appears. The user is expected to select a filename.

Once the user selects a filename and double-clicks the name, or presses the OK command button on the file-selection frame, the file-selection frame goes away again (its Visible property is set to False) and the simple three-button form returns. If the user then clicks the Display File command button, the program reads the contents of the selected file into a single (but long) string variable. The program then displays that string variable in the text box whose Visible property is changed to True so that the user can see and scroll through the file. A sample file being displayed in the text box is shown in Figure P9.2.

**Figure P9.2. Displaying the contents of the selected file.**

## Studying the File's Input

The majority of PROJECT9.MAK contains the same frame control and code found in Unit 17's file-selection frame control. The primary difference lies in PROJECT9.MAK's Display File command button's Click code shown in P9.1.

---

**Warning:** If you display a file that's not an ASCII text file, the file will appear garbaged in the text box. For example, if you select a Microsoft Excel spreadsheet, you won't see the spreadsheet inside the text box, but you will see a compressed binary representation of the spreadsheet.

---

The code uses the Line Input# statement to read each record in the file that the user selected in the file-selection frame. The line-by-line description explains the code inside the cmdDisp_Click() procedure shown in Listing P9.1.

**Listing P9.1. Reading an entire file into a string variable.**

```
1: Sub cmdDisp_Click ()

2: ' Read the whole file (up to 30,000 characters)
```

```
3: ' into a single string variable. Display that

4: ' string variable in a text box.

5: Dim count As
Integer ' Holds character count

6: Dim FileSpec, ALine As String

7: Dim FileHold As String ' Holds entire file

8: Dim NL As String

9:

10: NL = Chr$(13) + Chr$(10)

11:

12: ' Gather the filename into a single string

13: ' Add an ending backslash if
the path has none

14: If (Right$(dirList.Path, 1) <> "\") Then

15: FileSpec = dirList.Path & "\" & txtFiles.Text

16: Else

17: FileSpec = dirList.Path & txtFiles.Text

18: End If

19:

20: ' Open the file
```

```
21: Open
FileSpec For Input As #1

22:

23: ' Read up to the first 30,000 characters

24: Line Input #1, ALine ' Read a line

25: Do Until (EOF(1) = True)

26: ALine = ALine + NL ' Add a newline

27: ' Make sure that the read won't overshoot string limit

28: If
(count + Len(ALine)) > 30000 Then

29: Exit Do

30: End If

31:

32: FileHold = FileHold & ALine

33: count = Len(FileHold) ' Update count

34: Line Input #1, ALine ' Read a line

35: Loop

36: Close #1

37:
```

```
38: txtFile.Text = RTrim$(FileHold)


39: txtFile.Visible = True

40: End Sub
```

## Descriptions

1: The command button to display the file is named cmdDisp, hence the event procedure subroutine name of cmdDisp_Click().

2: A remark helps explain the purpose of the procedure.

3: The remark continues.

4: The remark continues.

5: Define an integer variable that will hold the length of the string variable as the program reads the file into the variable.

6: Define a string named FileSpec that will hold the pathname and filename. Also define a string named ALine that will hold each record from the file.

7: Define a string variable that will hold the entire file.

8: Define a string variable that will hold the carriage return and line feed or newline character.

9: A blank line helps separate parts of the procedure.

10: Define the newline character by concatenating a carriage return and line feed character together.

---

10: Use the ASCII table when you need to concatenate control codes.

---

11: Blank lines help separate parts of code.

12: A remark that explains this section of the procedure.

13: The remark continues.

14: If the selected path doesn't end with a backslash, the code must add one.

---

14: A backslash, \, always precedes a filename.

---

15: Concatenate a backslash after the selected path and before the filename.

16: Otherwise... (the selected path already ends with a backslash).

17: Concatenate the selected path and filename.

18: Terminate the If.

19: Blank lines help separate parts of code.

20: A remark explains the purpose of this section of code.

21: Open the selected file for input. Assign the file to the filenumber 1.

22: Blank lines help separate parts of code.

23: A remark helps explain this section of the code.

24: Read the first record.

25: Loop as long as the end of file is not yet reached.

26: Append a newline character to the record just read.

---

26: Line Input# does not read the file's newline character combination.

---

27: A remark explains this section of the code.

28: String variables can contain only slightly more than 30,000 characters. Therefore, ignore the record just read and quit reading if the record length puts the number of characters over 30,000 characters.

29: Terminate the procedure if the string limit was reached.

30: Terminate the If.

31: A blank line helps separate parts of the code.

32: Append the record to the string variable that holds all the records read to that point.

33: Update the count of the file length read so far.

34: Read the next line from the file.

35: Continue the loop.

36: Close all files when you're done with them.

37: A blank line helps separate parts of the code.

38: Trim off any excess spaces from the file string and display the file in the text box.

39: Make the text box visible.

40: Terminate the subroutine procedure.

## Close the Application

You can now exit the application and exit Visual Basic. The next lesson explains how to add menus to your application and manipulate a new control, the timer control.

# Visual Basic in 12 Easy Lessons



# Lesson 10, Unit 19

## Menus Improve Usability

### What You'll Learn

- The Menu Design window
- Adding the Menu Bar
- Adding pull-down menus
- Connecting menus to event procedures
- Adding an extra touch to Help

This lesson explains how to add menus to your applications. Why add menus? As you already know, menus add commands to your application that the command buttons alone can't handle. Although a command button often performs the same task as one of the menu options, Windows users expect to see menus. Common program commands such as exiting a program reside on the menu's list of commands.

How do you decide which menu options to include? Look at any popular Windows programs. Most

Windows programs contain common menu commands and features. Visual Basic is one such program. Many of the Visual Basic pull-down menus contain the same commands as Microsoft Word and Microsoft Excel.

Microsoft products aren't the only Windows programs with common menu items, either. When you write a program, you want users to feel comfortable with your interface. The user responds well to familiarity. Only when the interface contains common commands will the user be likely to use your program without fuss.

## The Menu Design Window

***Concept:*** Visual Basic makes creating and placing menu bar items into your application as easy as pushing command buttons and typing a few keystrokes. The Menu Design window contains menu description tools that enable you to create the application's menu bar, menu commands, and shortcut access keys to all of your applications.

The Menu Design window is a dialog box that you access from the Form window by pressing Ctrl+M or by selecting Window Menu Design from Visual Basic's own menu bar. Figure 19.1 shows the Menu Design window and names the parts of the Menu Design window.

**Figure 19.1. The Menu Design window.**

The Menu Design window creates your menu, but you still need to write event procedures that tie menu command selections to actions taken by your application. When the user selects a menu command, Visual Basic generates an event, just as it generates an event when the user clicks a command button.

---

**Note:** Learning to add menus to your programs involves a mastery of the Menu Design window more than anything else. After you use the Menu Design window to create the menu, the menu's event procedures work just like the other event procedures that you've been writing throughout this book.

---

As a prelude to adding a menu bar to your applications, study Figure 19.2 to learn the proper names for menu-related elements of a menu bar. This unit will walk you through the creation of a menu by adding a menu bar to the previous lesson's project application named PROJECT9.MAK.

**Figure 19.2. The parts of a menu.**

---

**Note:** This unit doesn't add event procedures to *every* menu command that you will add to PROJECT9.MAK. This unit explains how to create the menu and the menu's elements, and also explains how to hook up a couple of event procedures to the menu. Once you see how to create the menu and add an event procedure to a few menu commands, you know enough to add event procedures to all the rest.

---

***Review:*** Add menus to your applications using the Menu Design window. The Menu Design menu

enables you to add the menu bar, pull-down menu commands, separator bars, and shortcut access keystrokes to menu commands. After you create the menu, you'll write event procedures for each menu command. When the user selects a menu command, that menu command's event procedure will automatically execute.

## Adding the Menu Bar

***Concept:*** An application's menu bar is one of the easiest parts of the menu system to add. This section walks you through the steps necessary to add a menu bar. Subsequent sections add pull-down menu items to each of the menu bar commands.

The Menu Design window makes adding a menu bar to any application simple. Load the PROJECT9.MAK file now and get ready to add a menu bar that consists of the following commands:

- File
- Edit
- View
- Help

Before doing anything else, save the PROJECT9.MAK form and project files under different names. By saving the files under different names, you'll protect the contents of the project so that the project matches the description given at the end of the previous lesson. Of course, if you *do* inadvertently change PROJECT9.MAK, you can always copy the files named PROJECT9.FRM and PROJECT9.MAK from this book's companion disks back to your hard disk.

The following steps assume that you have the PROJECT9.MAK file loaded and that you want to save a copy of the form and project file under the names MYMENU.FRM and MYMENU.MAK.

1. Select File, Save File As to open the Save File As dialog box and enter the new name, MYMENU.FRM, at the File Name prompt. Press Enter or click OK to close the dialog box. The File Save As dialog box saves only the form file.
2. Select File, Save Project As to open the Save Project As dialog box and enter the new name, MYMENU.MAK, at the File Name prompt. Press Enter or click OK to close the dialog box.

Now you have a copy of the PROJECT9.MAK project called MYMENU.MAK that you can modify by adding a menu. Lesson 9's project will remain intact.

---

**Tip:** This book could go into a lot of detail explaining all the nuances of the Menu Design window. Luckily, you don't need all that preliminary detailed description. The Menu Design window is most easily mastered by jumping in and building a menu from scratch.

---

Every command on a menu bar, as well as the menu commands and separator bars that appear when you display a pull-down menu, has properties just as the other controls do. The Menu Design window acts like a dialog box that helps you set menu property values. The Property window is perfect for the other

controls, but as you'll see, menus require a few extra kinds of property choices that the other controls don't need.

With the MYMENU.MAK application still loaded, follow these steps to add the menu bar:

1. Press Ctrl+M to open the Menu Design window. For this section of the unit, you're adding only the menu bar commands. Each menu bar command requires a caption (specified by the Caption property) and a name (specified by the Name property).
   Optionally, you can set other properties, such as an Enabled property, which determines whether the menu item is grayed out and unavailable for certain procedures, as well as a Visible property, which determines when the user can see the menu bar command. Generally, you'll rarely change these extra property values from their defaults for the menu bar commands.
2. At the Caption prompt, type **&File**. The ampersand, as with the other controls' caption properties, indicates an access keystroke of Alt+F for the File menu item. As you type the caption, notice that Visual Basic adds the caption in the Menu Design window's lower section.

---

**Note:** The bottom half of the Menu Design window contains a description of the full menu, whereas the top half of the Menu Design window contains a description of individual items in the menu.

---

3. Press Tab to move the focus to the Name text box, and type **mnuFile**. The application will refer to the File menu bar item by the name mnuFile as needed. The rest of the Menu Design window's default property settings are fine as set. Your screen should look something like the one in Figure 19.3.
   The only shortcut access key available for menu bar commands is the underlined Alt+keystroke that occurs as the result of the Caption property's underlined letter. Don't attempt to select a Ctrl+keystroke from the Shortcut dropdown list box for the menu bar commands. Ctrl+keystroke shortcut access keystroke combinations are available only for pull-down menu commands.
   Don't press Enter or click the OK button to close the Menu Design window just yet, because you've got to add the additional menu bar commands before closing the window.

**Figure 19.3. The File menu bar command is now added to the menu.**

---

Naming Menu Items: The event procedures within any Visual Basic application's code reference menu items by their menu item names. Preface all menu items, both menu bar as well as pull-down menu items, with the mnu caption prefix so that you can easily distinguish menu commands from variables and from the other controls as you work with the application's code.
Generally, Visual Basic programmers follow the naming standard of naming menu bar items mnu, followed by the name of the item. Therefore, File is named mnuFile, Edit is named mnuEdit, and so on. As you add additional items to the pull-down menus, preface each of those items with the mnu prefix as well as the name of the menu bar command, and *then* append the name of the pull-down menu's item. Therefore, the File Exit item will be named mnuFileExit, View Normal will be named mnuViewNormal, and so on. The names then clearly describe the menu items that they represent.

---

Adding the remaining menu bar commands requires little more than adding the three items to the lower window of the Menu Design window. These steps complete the creation of the MYMENU.MAK's menu bar:

1. Click the Menu Design window's Next command button to inform Visual Basic that you want to add the next item. The lower window's highlight bar drops down to the next line in preparation for the next menu item.
2. Type **&Edit** at the Caption text box and press Tab. Name the second menu bar item mnuEdit. Click the Next command button to prepare the Menu Design window for the next menu bar item.
3. Type **&View** and press Tab to move the focus to the Name text box. Type **mnuView** and select Next to prepare the Menu Design window for the final menu bar item.
4. Type **&Help** and press Tab to move the focus to the Name text box. Type **mnuHelp**. Your screen should look like the one in Figure 19.4.

**Figure 19.4. The Menu Design window with all four menu bar items entered.**

Close the Menu Design window be pressing Enter or clicking the OK command button. Immediately, Visual Basic displays the new menu bar across the top of the application's Form window.

***Review:*** The Menu Design window provides the tools needed to add a menu bar along with the menu bar's pull-down menus and commands. Adding the menu bar items involves little more than typing the item name and caption. You're now ready to add the individual items to the menu. The next section explains how to complete this unit's File menu bar pull-down menu.

## Adding Pull-Down Menus

***Concept:*** Each menu bar command opens a pull-down menu that consists of a series of commands, separator bars, and access keystrokes. These pull-down menus are sometimes called *submenus*. The four arrow key command buttons inside the Menu Design window enable you to indent the pull-down menu commands from their matching menu bar commands, to show which items go with which menu bar commands.

Now that you've added the menu bar, you can add the individual items to the pull-down menus. You didn't have to complete the menu bar before completing each pull-down menu. You could have added the File command to the menu bar and then completed the File pull-down menu before adding the View command to the menu bar. The order in which you add menu items doesn't matter at all.

The File pull-down menu will contain the following items:

- The New command
- The Open command with a shortcut access keystroke of Ctrl+O
- The Close command
- A separator bar
- The Exit command

After you add these submenu items, you can hook up the menu commands to event procedures that you

write, which is explained in the next section.

Adding submenu items consists of following steps that are virtually identical to the ones you followed when adding the menu bar items. The difference is that the additional Menu Design window options become more important because you'll apply these options more frequently to the pull-down menu items. Table 19.1 explains the remaining Menu Design window properties.

**Table 19.1. The Menu Design window's remaining property explanations.**

| *Property* | *Description* |
|---|---|
| Checked | Indicates whether a menu item has a check mark next to the item. Generally, you'll add check marks to menu commands that perform on or off actions, such as a View menu that contains a Highlighted command. The check mark appears when you, at design time or through code, set the menu item's Checked property to True. The check mark goes away (indicating that the item is no longer active or selected) when you set the Checked property to False. |
| HelpContextID | This is a code that matches a help file description if you add help files to your application. |
| Index | If you create a menu control array rather than name individual menu items separately, the Index property specifies the menu item's subscript within the control array. |
| Shortcut | This is a dropdown list of Ctrl+keystroke access keys that you can add to any pull-down menu item. |
| Window List | Specifies whether the menu item applies to an advanced application's MDI (Multiple Document Interface) document. The menus that you create for this book don't require the use of MDI features. |

Perhaps the most important command keys on the Menu Design window when you add pull-down menu items are the four arrow command buttons. The left and right arrow command buttons indicate which items go with which menu bar command. In other words, if four items in the lower window are indented to the right and appear directly beneath the File menu bar item, those four indented items will appear on the File pull-down menu. The left arrow removes the indention and the right arrow adds the indention. The up and down arrow keys move menu items up and down the list of menu items, rearranging the order if you need to do so.

The arrow keys make a lot of sense when you see them used. Follow these steps to create the File pull-down menu bar's submenu:

1. Move the lower window's highlight line to the &Edit menu bar item. Click the Insert command button. You always insert *before* an item, so to add items to the File menu, you must *insert* before the Edit menu bar item in the lower window.
2. Click the right arrow command button. Visual Basic adds ellipsis, showing that the newly inserted item will be indented under the File option.
3. Move the focus to the Caption prompt and type **&New**.
4. Press Tab to move the focus to the Name prompt and type **mnuFileNew**.

5. Click Next and then Insert to insert another item beneath the New item. Your Menu Design window should look like the one in Figure 19.5.

**Figure 19.5. In the process of adding to the File menu.**

6. Move the focus to the Caption prompt and type **&Open**. Press Tab and enter the Name value of mnuFileOpen. Rather than add the next item, click the Shortcut dropdown list and select Ctrl+O from the list. When the user now displays the File pull-down menu, Ctrl+O will appear as the shortcut key next to the File Open menu item.
7. Click Next and then Insert to make room for the next item. Add the Close caption with the name mnuFileClose. Click Next and then Insert to insert another item beneath the Close item. It's now time to add the separator bar.

Separator bars help you break individual pull-down menus into separate sections. Although several commands appear on most Windows applications' File pull-down menus, these commands don't all perform the same kind of functions. Some relate to files, some relate to printing, and the Exit command always appears on the File menu as well. The separator bars help distinguish groups of different items from each other on the pull-down menus.

All separator bars have the same Caption property that is nothing more than a hyphen, -. You must give every separator bar a different name. Usually, the name of the separator bars on the File menu are mnuFileBar1, mnuFileBar2, and so on. You must add the separator bars using the right arrow's command button so that they indent properly beneath their pull-down menus. Follow these steps to add the single separator bar for this MYMENU.MAK's File pull-down menu:

1. Type - (the hyphen) for the Caption and press Tab.
2. Type **mnuFuleBar1** for the Name.

There's one more item to add. You know enough to add the Exit command to the File menu. After adding Exit, your Menu Design window should look like the one shown in Figure 19.6.

**Figure 19.6. The completed File pull-down menu commands.**

As an extra menu feature, add one checkmarked item to the View pull-down menu. Add an indented Highlighted item with the name mnuViewHighlighted. Click the Checked check box. The View Highlighted item will initially be checked when the user selects the View Highlighted from the menu.

Now that you've completed the menu (as far as we're taking it here), click the OK command button. When the Menu Design window disappears, you'll see the application's Form window with the menu bar across the top of the screen.

***Review:*** Adding menus to your applications requires only that you master the Menu Design window. Menus are nothing more than advanced controls with property values that you set using the Menu Design window. Most menu items require that you specify a Caption and Name property as well as indent the item properly under its menu bar command. Optionally, a menu item might contain a shortcut access keystroke or a check mark next to the item.

# Connecting Menus to Event Procedures

***Concept:*** Once you've built your menu, you need to tie each menu command to your application. To respond to menu selections, you need to write Click event procedures that you want Visual Basic to execute when the user selects a menu command.

Visual Basic generates a Click event when the user selects a menu command. The name of the menu command, combined with Click, provides the name of the event procedure. Therefore, the File Exit menu item will generate the execution of the event procedure named mnuFileExit_Click().

Adding the mnuFileExit_Click() event procedure requires only that you select that menu command during the program's development. At the Form window, click the File menu bar command. Visual Basic displays the File pull-down menu. Even though you're not running the program but are working on the program from the Form window, the File menu appears to show you what happens when the user selects File at runtime.

Click the Exit item on the File pull-down menu. As soon as you click Exit, Visual Basic opens the Code window to a new event procedure named mnuFileExit_Click(), as shown in Figure 19.7.

**Figure 19.7. Visual Basic opens the Code window when you click a menu item.**

This event procedure is simple. When the user selects File Exit, you want the application to terminate. Therefore, insert the End statement to the body of the mnuFileExit_Click() procedure and close the procedure by double-clicking its control button.

Although this unit doesn't take the time to complete the menu bar's pull-down menus or add event procedure code to every menu item, perhaps it would also be good to hook up the File Open menu command to display the file-selection frame that appears when the user selects the Display a File command button on the form. Rather than duplicate the Display a File command button's code inside the mnuFileOpen_Click() procedure, you can execute the command button's Click event by adding the following line to the body of the mnuFileOpen_Click() procedure:

```
cmdSel_Click ' Execute the Display a file
event
```

As you can see, adding event procedures requires little more than clicking the menu item and adding the body of the procedure that appears.

***Stop and Type:*** Add yet another event procedure to the View Highlighted menu option by clicking its menu item and adding the code so that the procedure looks like that in Listing 19.1.

***Review:*** After building your menu, you must tie code to the various menu items by writing Click event procedures that will execute when the user runs the application and selects from the menu. If any menu command duplicates the functionality of other controls, such as command buttons, don't copy the command button's code into the body of the menu event procedure. Instead, simply execute that

command button's event procedure from the menu item's event procedure.

**Listing 19.1. Code that boldfaces the displayed file.**

```
1: Sub mnuViewHighlighted_Click ()

2: ' Determines if the file being displayed

3: ' appears Boldfaced or not

4: mnuViewHighlighted.Checked = Not (mnuViewHighlighted.Checked)

5: txtFile.FontBold =
mnuViewHighlighted.Checked

6: End Sub
```

***Output:*** Figure 19.8 shows a textual data file being displayed without a boldface font. Before you added the View Highlighted menu item, the file always appeared in a boldface font.

**Figure 19.8. When unchecked, the View Highlighted menu item turns off the file display's boldface font.**

***Analysis:*** When the user first runs the program, the View Highlighted item will be checked, meaning that the file that the user displays will appear in the boldfaced font.

Listing 19.1 uses the Not operator to set the FontBold property of the text box that displays the file in the application. Line 4 resets the check mark on the View Highlighted menu item. If the item is checked, line 4 sets the value to the opposite value using Not. The check mark will then disappear and won't be displayed if the user displays the View pull-down menu once again. Line 5 sets the FontBold property of the file's text box to the same True or False condition that the Checked item is now set to.

## Add an Extra Touch to Help

***Concept:*** Using the Chr$(), you can add a little-known trick to right justify the Help menu bar item.

Some applications display a menu bar similar to the one that you added to MYMENU.MAK, except that those applications right justify the Help menu item to appear at the far right edge of the menu bar, as shown in Figure 19.9.

**Figure 19.9. Using a trick, you can right justify the Help menu bar item.**

If you add an ASCII 8, the backspace control character, to a menu bar's Caption property, that menu bar item will appear at the right of the menu bar. Why the backspace control character? Nobody really knows!

You can't easily add the backspace during the application's design, but you can add the backspace character using Chr$() during the program's start-up code. All you have to do is concatenate the Help menu bar item's caption to the Chr$(8) character. You must concatenate the Chr$(8) at runtime. Perhaps the best location to add the Chr$(8) is in the Form_Load() procedure (the procedure that executes before the user sees the form). Listing 19.2 contains the code needed to right justify the Help menu bar option.

**Listing 19.2. Code that right justifies the Help menu bar item.**

```
1: Sub Form_Load ()


2: ' Right-justify the Help menu bar item

3: mnuHelp.Caption = Chr$(8) & mnuHelp.Caption

4: End Sub
```

This unit doesn't do anything with the Help option because adding online help to an application is beyond the scope of this book. You can concatenate any menu bar command to the Chr$(8) if you want that command to appear right justified.

---

**Tip:** If you concatenate the Chr$(8) character to more than one menu bar command, each one will be right justified against the right edge of the menu bar.

---

***Review:*** By concatenating Chr$(8) at runtime, you can right justify menu bar commands against the right edge of the menu bar. Don't overdo the right justification, however. If you right justify any menu bar command, you should probably right justify only the Help command.

One of this book's disks contains the MYMENU.MAK application in its current state under the filename MYMENU2.MAK (as well as the form named MYMENU2.FRM). Therefore, if you had trouble with one of the menus or event procedures described here, you can load and study the MYMENU2.MAK application to see where you went wrong.

This unit ends here without adding event procedures to all of the menu items because you follow these same steps to add event procedures to the remaining items. You now understand how to add the menu bar, submenu items, and event procedure code to your applications, and you're well on your way to creating Windows programs that rival those of the pros!

# Homework

## General Knowledge

1. What Visual Basic tool do you use to design menus?
2. True or false: The Menu Design window enables you to specify property values for your application's menus.
3. True or false: You can add Ctrl+keystroke shortcut access keys to your menu bar commands.
4. True or false: You can add Alt+keystroke shortcut access keys to your menu bar commands.
5. What is the Name prefix that you should use for all menu items?
6. What is a *submenu*?
7. Which command buttons in the Menu Design window enable you to rearrange menu options?
8. Which command buttons in the Menu Design window enable you to indent menu items to indicate that those items are part of a pull-down menu?
9. Which menu property enables you to add check marks next to menu items?
10. Which menu property enables you to hide menu items from the user at various times?
11. True or false: Menu commands can be part of a control array.
12. What Caption property do all separator bars require?
13. What is the event name that all menu selections trigger?
14. What character right justifies menu items?
15. True or false: You can right justify more than one menu bar command.

## Find the Bug

1. Frank wants to right justify his View menu item. Frank attempts to type this into the menu item's Caption property: Chr$(8) & &View. Explain to Frank how to add the backspace character properly.

## Write Code That...

1. What would you name a menu item with the caption Split that appears when the user selects from the Window menu bar?
2. What name would you give to two separator bars located on the View pull-down menu?

3. Describe how you could gray out the menu items in MYMENU.MAK that have no event procedure code (such as File New) at this time.

## Extra Credit

Create a blank application with the following menu bar items: A, B, C, D, and E. Write code in the Form_Load() procedure to right justify the E item. On each pull-down menu, add the following items: 1, 2, 3, 4, and 5. Insert separator bars between the 2 and 3 and the 4 and 5. After you complete this, run the program to make sure that your menu items all appear as expected. After you complete this simple Menu Design window exercise, you will have mastered the basics of adding menu commands.

# Visual Basic in 12 Easy Lessons

---



---

# Lesson 10, Unit 20

## Time's Up

### What You'll Learn

- Introducing the timer control
- Using the timer control
- Events that occur infrequently
- The timer is not an alarm

This lesson describes the timer control. Unlike most other controls, the user never sees the timer control, even though you place the timer control on the form just as you place other controls on the form. The timer control keeps track of passing time intervals, and automatically triggers its own events when a preset time period goes by.

Unlike the other controls that respond to events, the timer control both triggers events *and* responds to events through the event procedures that you write. The timer control's property settings determine when the timer control generates events.

---

**Note:** The timer control has nothing to do with the Timer() function that you mastered in Lesson 7, "Functions and Dates." The timer control is a control that appears on the Toolbox window, whereas Timer() is a function that you add to your programs inside the Code window.

---

## Introducing the Timer Control

***Concept:*** The timer control sits invisibly on the user's form, waiting for a specified time interval to pass.

Figure 20.1 shows where the timer control appears on the toolbox. The timer control looks like a stopwatch. When you place the timer control on the form, the timer control looks like a stopwatch there also. At least, to *you*, the programmer, the timer control looks like a stopwatch; remember that the user can't see the timer control when the user runs the application.

**Figure 20.1. The timer control is the Toolbox window's stopwatch.**

You can place the timer control on the form just as you place any other control. You can do the following:

- Double-click the timer control so that it appears in the middle of the form. You then can drag the control to the location where you want the control to reside.
- Click and drag the timer control to the position on the form where you want the control to reside.

When you place the timer control, place the control out of the way of other controls. Keep in mind that the user won't see the timer control, so you can put the timer out of the way to keep it from interfering with the areas of the form where you'll place the other controls.

Not only is the placement of the timer control unimportant, but so is its size. No matter how you try to resize the timer control, Visual Basic refuses to change the size. The size of the control doesn't matter anyway because the user won't see the control.

Figure 20.2 shows the Form window after you place the timer control in the window's lower-left corner.

**Figure 20.2. The form with a single timer control.**

---

**Note:** Any application can have multiple timer controls, just as any application can have multiple instances of the other controls.

---

Table 20.1 contains a list of the timer property values available in the Property window for setting up the control. There aren't many timer control properties. If you think about the properties of the other controls, you may remember that most of the properties are designed to aid in the color and size selection of the controls (such as BackColor, Width, and so on). The timer control doesn't need all those extra properties

because the user won't see the timer control and those properties would be wasted.

---

*Definition:* A *millisecond* is one thousandth of a second.

---

**Table 20.1. The timer control properties.**

| *Property* | *Description* |
|---|---|
| Enabled | Specifies whether the timer control can respond to events. |
| Index | Indicates the subscript of the array if you store the timer control in a control array. |
| Interval | Determines the timer interval, in milliseconds, when you want a timer event to occur. The Interval value can range from 1 to 65535. |
| Left | Specifies the number of twips from the left edge of the Form window where the timer control resides. |
| Name | Contains the name of the control. Unless you change the name, Visual Basic assigns the default names of Timer1, Timer2, and so on as you add timer controls. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the option button. |
| Top | Specifies the number of twips from the top edge of the Form window where the timer control resides. |

Generally, the most important timer control property is the Interval property. The other controls are common ones used by most of the other controls, and you're already familiar with most of them. The Interval property is especially important for determining the operation of the timer event procedures that you write for the timer control.

---

**Warning:** Remember that the largest positive integer that you can store using the Integer data type is 32767. Therefore, if you initialize the timer's Interval property using a variable, be sure that you use a Long Integer variable or one of the floating-point variables (Single or Double) to hold the initial value.

---

Before studying what code to put in timer control event procedures, you need to understand that there is only one event available for the timer control. Not only are there just a handful of timer control properties, there is only a single kind of event procedure that you can write for the timer control.

Timer is the event reserved for timer controls. Therefore, if you name a timer control tmrAlarm, the event for that timer control will be called tmrAlarm_Timer(). To open that event, you can double-click the time control that you place on a form, or select the Event from the Code window's Proc dropdown list to open the Timer event procedure.

---

**Note:** Visual Basic distinguishes between the Timer event procedure from the built-in Timer() function by the context of how you use the Timer procedure or function.

---

Windows imposes a limit on the number of timer controls that you can add to an application at any one time. If you attempt to add more timer controls than Windows allows at the time, Visual Basic displays the error message shown in Figure 20.3. Windows can support only a limited number of timer controls (usually, Windows can handle at least 15 timer controls within a single application). Windows wouldn't be able to keep track of all of the time intervals properly, owing to timing conflicts and slowdowns that occur, if you were to use too many timer controls.

**Figure 20.3. The error message that you'll receive if you use too many timers.**

***Review:*** The timer control has few properties and even fewer events. There is no need to specify timer control size and color properties because the user can't see the timer control and these events would be meaningless.

## Using the Timer Control

***Concept:*** The Interval property determines how much time elapses between timer-generated events. Whereas other controls generate events in response to the user's input, the timer sits in the background, generating events on its own at preselected time intervals.

Use the timer control to execute an event procedure at a preselected time interval. When you place a timer control on a form, the timer's Interval property contains the number of milliseconds that pass before the timer's Timer() event procedure executes.

Suppose that you place a timer event on the control and set the timer's Interval property to 10000. If you name the timer event tmrSecond, the tmrSecond_Timer() event procedure executes every 10,000 milliseconds, or once every second.

***Stop and Type:*** Figure 20.4 shows the CLOCK.MAK application that you can load and run from this book's companion disks. The CLOCK.MAK application updates the time every second. The option button determines whether you hear a beep every second that the clock updates. The timer control is named tmrClock. Listing 20.1 contains the code for the tmrClock_Timer() event procedure that updates the time on the digital display, and beeps.

**Figure 20.4. Watch the time update every second.**

***Review:*** Initialize the Interval property with a value that represents the number of milliseconds that must pass between timer events. The Timer event procedure will then automatically execute every time that the Interval's time passes.

**Listing 20.1. A digital clock's Timer event procedure code.**

```
1: Sub tmrClock_Timer ()

2: '
Beep every second if the check box

3: ' is set. Also, update the value of the clock

4: '

5: ' This event procedure executes every second

6: ' (or every 1,000 milliseconds) because the

7: ' timer control's Interval value is 1000

8: If (chkBeep.Value)
Then

9: Beep

10: End If ' Don't beep if option button not set

11: ' Update the time inside the label

12: lblClock.Caption = Time$

13: End Sub
```

***Analysis:*** Line 1 shows that the name of the application's timer control is tmrClock. Therefore, that control's single event procedure is named tmrClock_Timer(). The timer control's Interval property is set to 1000 (milliseconds), so tmrClock_Timer() executes once every second.

Lines 8 through 10 check the check mark box to see whether the user wants the second beep or not. If the check box is checked, and therefore its Values property will be True, line 9 executes.

Line 12 then updates the value of the large clock label in the center of the form. The Time$ function accesses the computer's clock every second because this event procedure executes every second.

Figure 20.5 shows CLOCK.MAK's Form window during the program's development. Notice the timer

control sitting in the lower-left corner of the screen. This timer control doesn't appear when the user runs the program, but the timer control's involvement in the program is crucial because the timer control triggers the only event procedure in the entire application.

**Figure 20.5. The timer control doesn't display in the running program.**

## Events That Occur Infrequently

***Concept:*** The Interval property holds a maximum value of 65535. 65535 allows for a time interval of only a little more than a minute. If you write timer events that need to execute less frequently than every minute, you need to add code that keeps the timer event from running every time the timer event executes.

Suppose, for example, that you wanted to warn the user to select File Save every five minutes. Anytime that you need something to happen once each time period, whatever that time period may be, use the timer control. To display the File Save message box every five minutes, you could do the following:

1. Place a timer control on the application's form.
2. Determine how many milliseconds there are in five minutes. There are 1,000 milliseconds in each second. There are 60,000 milliseconds in each minute. Therefore, there are 300,000 milliseconds every five minutes.
3. The number of milliseconds exceeds 65,535 (the maximum allowable value for the Interval property) so you'll have to divide the time period into more manageable units of milliseconds, seconds, or one minute intervals.
   Five minutes is easily divided into five one-minute time intervals. One minute is equal to 60,000 milliseconds. Therefore, you'll have to set the Interval property to 60000 and control the File Save message box display so that the message box appears only once out of every five times that the event procedure executes.

***Stop and Type:*** Listing 20.2 contains a timer event procedure that executes a message box statement every five minutes. The timer event executes every minute, owing to the timer's Interval value of 60000. One simple way to accomplish displaying the message box every five minutes is to define a module variable named MinCount that counts the minutes that go by and executes the message box statement every time that MinCount reaches 5. Notice that, at the top of Listing 20.2, you'll find the (general) declarations procedure that defines a module-level variable named MinCount. MinCount won't lose its former value between event procedure executions because MinCount is a module variable and isn't locally defined inside tmrMinute_Timer().

***Review:*** When you need to execute code that must occur in time intervals greater than milliseconds, seconds, or one-minute intervals, define a module-level variable to keep track of multiple occurrences of second or minute time intervals.

**Listing 20.2. Executing a timer event every five minutes.**

```
1: ' Inside the (general) procedure

2: ' Define a module variable to keep

3: ' track of one-minute time intervals

4: Option Explicit

5: Dim MinCount As Integer ' Begins at zero

6:

7: Sub tmrMinute_Timer ()

8: ' Ensures that a message
box appears only every

9: ' five minutes despite the fact that this

10: ' event procedure executes every passing minute

11: MinCount = MinCount + 1 ' Raise minute count

12: If (MinCount = 5) Then

13: MsgBox "It's time to save your file!"


14: MinCount = 0 ' Reset the minute count

15: End If

16: End Sub
```

***Analysis:*** Line 5 defines the module variable named MinCount. All defined variables begin with initial values of zero. Remember that the timer named tmrMinute that executes the Timer subroutine procedure in this example would contain an Interval value of 60000, so the tmrMinute_Timer() event procedure executes every minute that goes by.

Once the minute count reaches 5, five minutes will have gone by since the MinCount variable was defined or since the last time that this event procedure executed. Therefore, line 13 displays the message box and line 14 resets the minute count to zero again so that the counting towards five minutes will begin to happen again. If, however, the minute count has not reached 5, the If condition is false and the subroutine terminates without displaying the message box.

## The Timer Is Not an Alarm

You can't set a timer control to trigger an event at a specific single period in time. The timer control does *not* work like an alarm that goes off at a specific point in time.

***Concept:*** The timer control executes its Timer event procedure every time that the defined Interval passes. The timer control doesn't act as an alarm, however, executing an event procedure at a preset time. You can program the Timer event procedure, however, to simulate an alarm clock or a calendar reminder.

The timer control triggers a repeated event over and over. When you write a timer event procedure, the code inside the procedure must be able to handle its own execution every time an event occurs. If you *do* want to execute a routine once at a preset time, you'll have to write the event procedure to execute repeatedly (once each preset time interval); the code then must check to see whether the PC's time equals that of the preset time that you're waiting on.

***Stop and Type:*** Listing 20.3 contains the code needed to turn CLOCK.MAK into a wake-up alarm program. You can find the alarm code on this book's disk under the project filename ALARM.MAK file. When you run the program, you'll see an extra command button labeled Set Alarm. When you click Set Alarm, an input box appears asking for a time to wake up. When you enter a correct time (the program loops until the you enter a correct time or choose Cancel), the set alarm time appears to the right of the command button as shown in Figure 20.6 and a message box alarm goes off when that time is reached.

***Review:*** When writing an alarm or calendar program, remember that the Timer event will always execute continuously, at least every 65,535 milliseconds because 65535 is the largest value that you can store in a timer control's Interval property. When the Timer event executes, your code can check the current time (or date if you write a date-checking calendar program) to see whether an alarm time has been reached.

**Listing 20.3. Code that sets an alarm and waits to wake up.**

```
1: ' Define a module variable used in two

2: ' different procedures. The module variable

3: ' cannot lose its value between runs.
```

```
4: Option Explicit

5: Dim TimeSet
As Variant

6:

7: Sub cmdSet_Click ()

8: ' Ask the user for an alarm time to set

9: ' Keep looping until the user:

10: ' Enters a valid time or

11: ' clicks the Cancel command button

12: Do

13: TimeSet = InputBox("Enter a 24-hour time to wake
up", "Time")

14: If (TimeSet = "") Then ' Check for Cancel

15: Exit Sub ' User pressed Cancel

16: End If

17: Loop Until IsDate(TimeSet) ' Keep asking if invalid

18: ' If here, the user entered a value that can

19: ' be
converted to a valid time value. Now,

20: ' convert the Variant data to a date/time type
```

```
21: TimeSet = CVDate(TimeSet)

22: ' Display the wake-up time in a label

23: lblSet.Caption = "Alarm @ " & TimeSet

24: End Sub

25:

26: Sub
tmrClock_Timer ()

27: ' Sound an alarm if the set time has been hit

28: ' just now or perhaps bypassed by a fraction

29: Dim ctr As Integer

30:

31: If (chkBeep.Value) Then ' Second beep if needed

32: Beep

33: End If

34: lblClock.Caption = Time$ '
Update the screen's clock

35:

36: ' Sound the alarm if the wake up time has been

37: ' passed (also, make sure a time has been set)

38: If (Time >= TimeSet) And (lblSet.Caption <> "No alarm") Then
```

```
39: For ctr = 1 To 50

40: Beep '
Wake up!!!

41: Next ctr

42: MsgBox "W-a-k-e U-p-!-!-!", MB_ICONEXCLAMATION, "Alarm"

43: lblSet.Caption = "No alarm" ' To keep alarm from resounding

44: End If

45: End Sub
```

***Output:*** Figure 20.6 shows what happens when the user sets a time that's about to trigger an alarm.

**Figure 20.6. An alarm is ready to go off.**

When the alarm sounds, the user hears a wake-up series of beeps and sees the message box shown in Figure 20.7.

**Figure 20.7. It's time to wake the user.**

***Analysis:*** Line 1 through 5 contains the (general) code that defines a module variable named TimeSet that will hold the user's wake-up time. The reason that you often must define module variables when dealing with the Timer event, instead of using an argument list to pass values, is because Visual Basic generates the Timer event and there is no way to request that Visual Basic generate a Timer event procedure that receives or sends arguments. Therefore, any variable that you use inside a Timer event procedure must be a module or global variable if you want to initialize that variable elsewhere and use that variable inside the Timer procedure.

Line 7 begins the procedure that requests the wake-up alarm time from the user. The subroutine contains a loop in lines 12 through 17 that asks the user for a time, and keeps asking until the user enters a valid time or until the user presses the Cancel command button.

Line 21 then converts the user's variant variable that contains a valid time (thanks to the IsDate() function call on line 17, which ensured that the user entered a valid time) to a date format that you can use to compare against the Time$() function in the Timer() event procedure later. Line 23 changes the default caption of No alarm next to the Set Alarm command button to the newly set wake-up time.

---

**Caution:** If the user enters only a date with no time, line 17's IfDate() function passes the date along.

There is no way to check for an individual time using a built-in function.

---

Line 26 begins the Timer event procedure. Lines 31 through 33 check to see whether the second beeping should continue (the user would have selected the check box on the form if the beep is desired). Line 34 updates the large clock display on the screen every second without fail.

Line 38 checks to see whether it's time for the alarm to sound. Owing to imperfections in Windows, it's possible for the Timer event procedure to skip a second once in a while, perhaps because the user switched to another Windows program temporarily, which sometimes causes a timer control to miss a beat. Therefore, line 38 checks to see whether the current time is later than or equal to the set time using the >= operator. Also, line 38 checks to see whether an alarm is even set.

If the alarm time has been reached and an alarm is set, a For loop in lines 39 through 41 beeps the speaker for a series of 50 beeps, and then line 42 displays a message box to awaken the user. Line 43 resets the caption now that an alarm just sounded, so the program won't keep sounding the alarm every subsequent second that follows.

# Homework

## General Knowledge

1. What control keeps track of elapsed time?
2. What is the most important property of the timer control?
3. Why doesn't the timer control contain any color or size properties?
4. What does the user see in the upper-left corner of a executing program's form when you place a timer control in the upper-left corner of a form?
5. True or false: The user triggers timer events.
6. What time value is measured by the timer control's Interval property?
7. True or false: An application can have, at most, one timer control.
8. What is a *millisecond*?
9. How much time does ten thousand milliseconds measure?
10. How much time does sixty thousand milliseconds measure?
11. What Name prefix do programmers usually assign to the timer control?
12. What problem can you run into if you attempt to set a timer's Interval property with an integer variable?
13. What is the name of the only event procedure available for time controls?
14. True or false: You can set the timer's Interval property to trigger an event at a certain preset time.
15. Why would Visual Basic sometimes be error prone if you were to test the Time$() value against an exact time match?

## Write Code That...

1. Modify the code in project 7 so that the time display updates every second. The update should occur no matter which time zone the user selects.

## Extra Credit

Write an application with three command buttons labeled 10 Seconds, 30 Seconds, 45 Seconds. When the user clicks one of the command buttons, display a seconds countdown for the appropriate number of seconds. When the countdown finishes, beep five times to tell the user that the countdown completed. You can use either three separate timer controls or modify one timer's interval value depending on the command button selected by the user.

# Visual Basic in 12 Easy Lessons



# Project 10

## Making Programs "Real World"

### Stop & Type

This lesson taught you how to design and add menus to your Visual Basic applications. The Menu Design window enables you to specify the property values for menu bar commands and pull-down menu items. By using the Menu Design window, you can add shortcut access keystrokes and menu separator bars, and specify special menu features such as checkmarked menu items. Generating program code that responds to menu events is no more difficult than writing event procedures that respond to command button clicks.

In addition to creating menus, you also learned how to work with the timer control. As you saw in the previous unit, the timer control is a control that triggers its own events. Depending on the value of the timer control's Interval property, the timer control generates an event every few milliseconds. The code that you add to the timer's event procedure determines how to handle the event. You can set alarms and perform certain screen and control update actions every time that the timer generates an event, if you want.

In this lesson, you saw the following:

- How to create menus for your Visual Basic applications

- Why the Menu Design window is so much more appropriate for specifying menu property values than the Property window
- How to right justify menu items if you want to change the menu bar a bit
- What the timer control does to generate events
- When to respond to a timer's event and when to ignore the event

## The Program's Description

Figure P10.1 shows the PROJEC10.MAK opening Form window. This project presents a simple number- and letter-guessing that contains a menu bar with three items. The program contains three event procedures that are tied to the menu's commands.

**Figure P10.1. Project 10's application begins with a simple form.**

The Menu Design window is nothing more than an advanced Property window for the menu. The menu events to which your code will respond are click events. Therefore, there is nothing really new with the code used in this program. The program generates a random number or letter when the user selects File New Game. If the Option Use Alphabetic Letters menu option is checked, the program will generate a random letter from A through Z. If the Option Use Alphabetic Letters menu option isn't checked, the program will generate a random number from 1 to 100. A small label at the bottom of the guess screen reminds the user to guess a number or letter when the guessing begins.

## Needed: Program Randomness

Because there is nothing new required for code that responds to menu events, this lesson's project introduces a new command and function to generate the random value and make the project more interesting. The random number generator is the only part of Visual Basic's powerful language that does *not* do the same thing every time you execute the same code.

The Rnd() function, when used without an argument, generates a random number between 0 and 1. If the user wants to guess a number, however, the program needs a random number from 1 to 100, and if the user wants to guess a letter, the program needs a random number from 65 to 90 to represent Chr$() arguments for the letters A through Z. The program uses the following formula to generate a random number from a low value to a high value:

```
Int((LowValue + HighValue + 1) * Rnd + LowValue
```

Therefore, here is the assignment statement that generates a random number from 65 to 90, and stores the character value of that generated ASCII number in a variable named Letter:

```
Letter = Chr$(Int(26) * Rnd + 65) ' A through Z
```

There is only one problem with the Rnd function. The function generates the same set of random values every time you run a program that uses Rnd, unless you first execute the following statement:

```
Randomize Timer
```

The Randomize command requires a single value. That value *seeds* the random number generator by providing a value for the first generated random number. If you were to use the same seed value, Rnd wouldn't change from program run to program run. A guessing game that always produced the same set of values to guess wouldn't remain interesting. By using Timer to seed the internal random number generator, you can ensure that Randomize rarely works with the same value twice, because the randomizing seed value is the internal clock setting at the time that the Randomize statement executes.

## The Help Menu

You'll notice that the Help menu is right justified. The Form_Load() event procedure concatenates a Chr$(8) backspace character to the menu bar's Help command. As you learned in Unit 19, the backspace character right justifies menu bar commands.

Figure P10.1 shows the help message and command button that appear when the user selects the Help command. The Help command's event procedure, shown in Listing P10.1, sets the Visible property of the command button and help label to True so that the user can see the help message. When the user clicks the command button, the cmdHelp_Click() event procedure (also shown in Listing P10.1) executes to hide the help label and command button.

**Figure P10.2. The Help menu command produces a program overview for the user.**

**Listing P10.1. Respond to the Help command and turn off help when requested.**

```
1: Sub mnuHelp_Click ()

2: ' Show the help message and the help command button

3: lblHelp.Visible = True

4: cmdHelp.Visible = True

5: End Sub
```

```
6:

7: Sub cmdHelp_Click ()

8: ' Hide the help message and the help command button

9: lblHelp.Visible = False

10: cmdHelp.Visible = False

12: End Sub
```

## Descriptions

1: The code for the event procedure that responds to the Help menu bar command must be a click procedure.

2: A remark that explains the procedure.

3: Turn on the large label that appears in the center of the screen.

---

3: The help is a label that normally remains invisible.

---

4: Turn on the command button that the user can use to hide the help information.

5: Terminate the procedure.

6: A blank line separates the procedures.

7: The code for the event procedure that responds to the command button that turns off Help's information.

8: A remark that explains the procedure.

9: Turn off the large label that appears in the center of the screen.

10: Turn off the command button that the user can use to hide the help information.

11: Terminate the procedure.

## Studying the Rest of the Code

Listing P10.2 contains the rest of the program, including the Form_Load() event procedure that right justifies the Help command and seeds the random number generator so that subsequent calls to Rnd will produce a different set of values between program runs. Figure P10.3 shows the running program during the user's guessing of a letter. Notice that a label appears at the bottom of the screen during the progress of the game that tells the user whether letters or numbers are to be guessed.

**Figure P10.3. The user must guess a letter from A to Z.**

**Listing P10.2. The code for the guessing-game project listing.**

```
1: Sub Form_Load ()

2: ' Begin with a number-guessing game

3: ' Turn off the checkmark
from the

4: ' Options Use Alphabetic Characters menu

5: mnuOptionUseAlpha.Checked = False

6: ' Right-justify the Help menu command

7: mnuHelp.Caption = Chr$(8) + mnuHelp.Caption

8: ' Spin the internal randomizing wheel

9: Randomize Timer

10: End Sub


11:
```

```
12: Sub mnuFileExit_Click ()

13: End

14: End Sub

15:

16: Sub mnuFileNewGame_Click ()

17: ' Call appropriate subroutine procedure depending

18: ' on value of Option Use Alphabetic Characters

19:

20: ' Turn off opening label if it's still
visible

21: If (lblOpening.Visible = True) Then

22: lblOpening.Visible = False

23: End If

24:

25: If (mnuOptionUseAlpha.Checked) Then

26: Call GuessLetter

27: Else

28: Call GuessNumber

29: End If
```

```
30: ' Turn back on opening label

31:
lblOpening.Visible = True

32: End Sub

33:

34: Sub mnuOptionUseAlpha_Click ()

35: ' Reverse the state of the checkmark

36: mnuOptionUseAlpha.Checked = Not (mnuOptionUseAlpha.Checked)

37: End Sub

38:

39: Sub GuessNumber ()

40: ' User guesses a
number

41: Dim Guess As Variant

42: Dim Number, GuessNum As Integer

43:

44: ' Update the instruction label

45: lblInstruct.Caption = "Guess a number from 1 to 100"

46: lblInstruct.Visible = True

47: ' Find value from 1 to 100
```

```
48: Number =
Int(100 * Rnd + 1)

49:

50: Do

51: Guess = InputBox("What's your guess?", "Guess")

52: If (Guess = "") Then ' Check for Cancel

53: ' Turn off instruction label

54: lblInstruct.Visible = False

55: Exit Sub ' User pressed
Cancel

56: End If

57: GuessNum = Guess ' Convert to number

58: If (GuessNum > Number) Then

59: MsgBox "Your number is too high...", , "Wrong"

60: Else

61: If (GuessNum < Number) Then

62: MsgBox "Your number is too
low...", , "Wrong"

63: End If

64: End If
```

```
65: Loop Until (GuessNum = Number)

66: Beep

67: MsgBox "You guessed " & Guess & "! Correct!!!", , "Lucky"

68: ' Turn off instruction label

69:
lblInstruct.Visible = False

70: End Sub

71:

72: Sub GuessLetter ()

73: ' User guesses a letter

74: Dim Letter, GuessLet As String

75:

76: ' Update the instruction label

77: lblInstruct.Caption = "Guess a letter from A to Z"

78:
lblInstruct.Visible = True

79: ' Find value from ASCII 65 - 90 ('A' to 'Z')

80: Letter = Chr$(Int(26) * Rnd + 65) ' A through Z

81: Do
```

```
82: GuessLet = InputBox("What's your guess?", "Guess")

83: If (GuessLet = "") Then '
Check for Cancel

84: ' Turn off instruction label

85: lblInstruct.Visible = False

86: Exit Sub ' User pressed Cancel

87: End If

88: ' Convert user's letter to uppercase

89: GuessLet = UCase(GuessLet)

90: If (GuessLet > Letter) Then

91: MsgBox
"Your letter is too high...", , "Wrong"

92: Else

93: If (GuessLet < Letter) Then

94: MsgBox "Your letter is too low...", , "Wrong"

95: End If

96: End If

97: Loop Until (GuessLet = Letter)

98:

99: ' Here is
```

```
user guessed number

100: Beep

101: MsgBox "You guessed " & GuessLet & "! Correct!!!", , "Lucky"

102:

103: ' Turn off instruction label

104: lblInstruct.Visible = False

105: End Sub
```

## Descriptions

1: The code for the opening event procedure that executes right before the user sees the form.

2: A remark explains the procedure.

3: The remark continues.

4: The remark continues.

5: Set the guess to letters.

6: A remark explains the code.

7: Right justify the Help menu option.

8: A remark explains the code.

9: Use the internal clock to seed the random number generator.

---

9: Be sure to seed the random number generator so that truly random values appear.

---

10: Terminate the procedure.

11: A blank line separates the procedures.

12: The code for the File Exit menu command.

13: Terminate the program's execution.

14: Terminate the procedure.

15: A blank line separates the procedure.

16: The code for the File New Game menu command.

17: A remark explains the procedure.

18: The remark continues.

19: A blank line helps separate parts of the program.

20: A remark that explains the procedure.

21: Turn off the display of the Press File New Game label if it's on.

22: Hide the label.

23: Terminate the If.

24: A blank line helps separate parts of the program.

25: See whether the user wants to guess a letter.

26: Execute the subroutine procedure that enables the user to guess a letter.

27: If the user wants to guess a number...

28: Execute the subroutine procedure that enables the user to guess a number.

29: Terminate the If.

30: A remark explains the code.

31: Now that the user has finished the game, turn the Select File New Game label.

32: Terminate the procedure.

33: A blank line helps separate parts of the program.

34: The code for the Option use Alphabetic Letters menu command.

35: A remark helps explain the procedure.

---

35: Check or uncheck the menu command when clicked.

---

36: Turn the menu command's check mark either on or off.

37: Terminate the procedure.

38: A blank line helps separate the procedures.

39: The subroutine procedure called when the user wants to guess a number.

40: A remark explains the procedure.

41: Define a variable to hold the InputBox() function's return value.

42: Define variables to hold the user's guess and the random number generated for the guessing game.

43: A blank line helps separate parts of the procedure.

44: A remark explains the code.

45: Change the label that appears at the bottom of the screen during the user's guessing.

46: Displays the guessing label.

47: A remark explains the code.

48: Generate a random number from 1 to 100.

---

48: Unless you convert the value fall within a different range, Visual Basic generates a number from 0 to 1.

---

49: A blank line helps separate parts of the code.

50: Begin the guessing loop.

51: Get the user's guess.

52: Check to see whether the user pressed the Cancel command button at the display of the input box.

53: A remark explains the code.

54: Turn off the guessing label.

55: Terminate the guessing subroutine early.

56: Terminate the If.

57: Convert the user's Variant answer to an Integer.

58: Check whether the user's guess was too high.

59: If the user's guess was too high, tells the user.

60: The user's guess was not too high.

61: Check whether the user's guess was too low.

62: If the user's guess was too low, tells the user.

63: Terminate the If.

64: Terminate the If.

65: Keep asking the user for a guess until the user guesses the correct answer.

66: Audibly tells the user that the guess was correct.

67: Show the user that the guess was correct.

68: A remark helps explain the code.

69: Hide the guessing instructional label.

70: Terminate the procedure.

71: A blank line helps separate procedures.

72: The subroutine procedure called when the user wants to guess a letter.

73: A remark explains the procedure.

74: Define a variable to hold the user's guess and the random letter.

75: A blank line helps separate parts of the procedure.

76: A remark explains the code.

77: Change the label that appears at the bottom of the screen during the user's guessing.

78: Display the guessing label.

79: A remark explains the code.

80: Generate a random letter from A to Z.

81: Begin the guessing loop.

82: Get the user's guess.

83: Check to see whether the user pressed the Cancel command button at the display of the input box.

84: A remark explains the code.

85: Turn off the guessing label.

86: Terminate the guessing subroutine early.

87: Terminate the If.

88: A remark explains the code.

89: Convert the user's guess to uppercase if the letter is not already uppercase.

---

89: The UCase$() function keeps the user's guessing range within the uppercase letters.

---

90: Check whether the user's guess was too high.

91: If the user's guess was too high, tells the user.

92: The user's guess was not too high.

93: Check whether the user's guess was too low.

94: If the user's guess was too low, tells the user.

95: Terminate the If.

96: Terminate the If.

97: Keep asking the user for a guess until the user guesses the correct answer.

98: A blank line helps separate parts of the code.

99: A remark helps explain the code.

100: Audibly tells the user that the guess was correct.

101: Show the user that the guess was correct.

102: A blank line helps separate parts of the code.

103: A remark helps explain the code.

104: Hide the guessing instructional label.

105: Terminate the procedure.

## Close the Application

You can now exit the application and exit Visual Basic. The next lesson teaches you how to send text to the printer and really have fun by drawing graphics on the user's form.

# Visual Basic in 12 Easy Lessons



# Lesson 11, Unit 21

## Using the Printer

### What You'll Learn

- A word of warning

- Printing with Windows
- Telling the user you will print
- Introducing the Printer object
- The Print method
- Initiating the print
- Page breaks

This lesson describes how you can integrate the printer into Visual Basic applications that you write. Visual Basic communicates with the Windows printer driver so that you can send text and even graphics to the printer.

Unlike most of Visual Basic's features, however, sending output to the printer can be a tedious process. Surprisingly, one of Visual Basic's weaknesses is also its strength: Printing requires that you send a fairly long list of instructions to your printer that describes exactly the way the output is to look. As easily as Visual Basic allows you to add and manage controls, one would have thought that the printing could be made easier. Nevertheless, despite the tedium sometimes associated with the printer, you'll see that you can control every aspect of printing, including the font of individual characters that your application sends to the printer. Therefore, the tedious control needed for the printer provides pinpoint accuracy that allows you to control all printing details.

## A Word of Warning

***Concept:*** The Visual Basic Primer system that comes with this book doesn't support the use of the Print command on the File menu. You'll still be able to print reports from your applications, however.

If you attempt to select Print from the File menu, the Visual Basic Primer system displays the error message box shown in Figure 21.1. This book's version of Visual Basic doesn't support the print commands that you would normally find on Visual Basic's File menu.

**Figure 21.1. The File Print command is unavailable.**

In case you upgrade to a different version of Visual Basic, you should know that the File menu's Print command usually allows you to produce printed copies of the code within an application. In addition, you would normally be able to print an application's forms graphically as well as produce printed descriptions of forms. Such output could provide documentation that you could file away and use as a starting point if you modify the application at a later date.

***Review:*** If you attempt to select the Print command from the File menu, the Visual Basic Primer system produces a message box explaining that the feature is unavailable. Although you can't print form and code documentation, you will be able to produce text and graphic reports from the Visual Basic applications that you write by applying the properties and methods discussed in the rest of this unit.

## Printing With Windows

***Concept:*** When your application sends output to the printer, Windows intercepts those printer commands.

Rather than send output directly to the printer attached to your computer, Visual Basic actually sends printed output to the Windows Print Manager.

---

*Definition:* The *Windows Print Manager* controls all printed output in Windows.

---

The Windows Print Manager determines how all printed output from all Windows programs eventually appears. Therefore, when your Visual Basic application attempts to send printed output directly to the printer, the Windows Print Manager intercepts those commands, and might change the output before the printer ever sees the output.

The Windows Print Manager knows how to communicate with any printer supported by Windows. There are hundreds of different kinds of printers now recognized by Windows, and most of these printers require specialized commands. If every program that you bought had to provide support for every kind of printer that you or your users might own, programs would require even more disk space than they already do. In addition, programs would cost more because each software developer would have to spend the time writing the program to produce output on every kind of printer available.

Rather than require that every software developer support all printers, the Windows Print Manager requires that every software developer support only *one* kind of printed output: the kind required by the Windows Print Manager. If the applications that you write need to produce printed output, Visual Basic produces that output in a form required by the Windows Print Manager. Figure 21.2 shows that Visual Basic applications send output directly to the Windows Print Manager. The Windows Print Manager *then* converts that output into the individual commands needed by whatever printer is attached to the system.

**Figure 21.2. The Windows Print Manager collects all program output and manages individual printers.**

Suppose that you had both a laser printer and a dot matrix printer attached to your computer. Without the Windows Print Manager, you would need to provide two sets of printer commands for every Visual Basic application you write. With the Windows Print Manager, you'll need to provide only one generic set of printed output commands. Before running the application, you can use commands available in the Windows Print Manager to select one of your two printers. When you run the program, Windows will convert the Visual Basic output into commands needed by whatever printer is selected.

***Review:*** The Windows Print Manager simplifies communication with all the various printers. Your Visual Basic application needs only to send output to the Windows Print Manager no matter what kind of printer that output will eventually be directed to. The Windows Print Manager knows how to communicate with all Windows-supported printers, and converts your Visual Basic application's output to the chosen printer's required format.

## Tell the User You Will Print

***Concept:*** Users could be caught unaware if your application begins printing without first warning the user that the printer must be ready.

---

*Definition: Online* means the printer is ready for printing.

---

Always remind the user to turn on the printer, make sure that the printer has paper, and ensure that the printer is online. If the user's printer is not first turned on and ready with an ample paper supply, the user will receive a Windows Print Manager error message similar to the one shown in Figure 21.3.

**Figure 21.3. The Windows Print Manager senses that something isn't right.**

***Stop and Type:*** The function procedure in Listing 21.1 provides you with a useful MsgBox() call that you might want to incorporate into your own programs before printing.

***Review:*** Warn the user that printing is about to take place to make sure that the user puts paper in the printer and turns the printer online.

**Listing 21.1. Warns the user before printing.**

```
1: Function PrReady()

2: ' Make sure the user is ready to print

3: Dim IsReady As Integer

4: IsReady = MsgBox("Make sure the printer is ready", 1, "Printer Check")

5: If (IsReady = 2) Then

6: PrReady = 0 ' A
Cancel press returns a False value

7: End If

8: PrReady = 1 ' User pressed OK so return True

9: End Function
```

***Output:*** Figure 21.4 shows the message box presented by Listing 21.1.

**Figure 21.4. The user now knows to prepare the printer for printing.**

***Analysis:*** After the user reads the message and responds to the message box in line 4, the procedure's return value determines whether the user wants to see the output (assume that the user has properly prepared the

printer for printing) or cancel the printing. The return value of zero (meaning false) or one (meaning true) can be checked as follows from another procedure that prints based on the user's response:

```
If PrReady() Then ' If function is true

 Call PrintRoutine

End If
```

## Introducing the *Printer* Object

***Concept:*** Visual Basic applications send all printed output to a special Visual Basic object called the Printer object. The Printer object supports several property values and methods with which you determine the look of the printed output.

The keyword Printer specifies the printer object to which your applications will direct all output. There is no Printer control on the Toolbox window. All access to Printer must take place using Visual Basic code.

---

**Note:** The commands that your application sends to the Printer object are generic Windows printer commands. The Windows Print Manager converts those generic commands to a specific printer's commands. You, therefore, worry about what you want printed and let the Windows Print Manager worry about *how* the output gets produced.

---

Throughout this book, when you learned a new object, such as the command button control, you learned about the properties that relate to that object. Before using the Printer object, you should see the properties available for the Printer object so that you'll know what kinds of things you can do with printed output from within Visual Basic. All of the Printer object's properties are listed in Table 21.1.

---

*Definition:* A *pixel* is the smallest addressable point on the screen or printer.

---

**Table 21.1. The Printer object's properties.**

| *Property* | *Description* |
|---|---|
| CurrentX | Holds the horizontal print column, from the upper-left corner of the page, measured either in twips or the scale defined by the Scale properties. |

| | |
|---|---|
| CurrentY | Holds the vertical print row, from the upper-left corner of the page, measured either in twips or the scale defined by Scale properties. |
| DrawMode | Determines the appearance of graphics that you draw on the printer. |
| DrawStyle | Specifies the style of any graphical lines that your application draws. |
| DrawWidth | Specifies the width of lines drawn, from 1 (the default) to 32767 pixels. |
| FillColor | Specifies the color of printed shapes. Determines the shading density for noncolor printed output. |
| FillStyle | Contains the style pattern of printed shapes. |
| FontBold | Contains either True or False to determine whether subsequent printed output will be boldfaced. |
| FontCount | Specifies the current printer's number of installed fonts. |
| FontItalic | Holds either True or False to determine whether subsequent output will be italicized. |
| FontName | Holds the name of the current font being used for output. |
| Fonts | Contains a table of values that act as if they were stored in a control array. Fonts(0) to Fonts(FontCount - 1) holds the names of all installed fonts on the target computer. |
| FontSize | Holds the size, in points, of the current font. |
| FontStrikeThru | Holds either True or False to determine whether subsequent output will be printed with a strikethru line. |
| FontTransparent | Holds either True or False to determine whether subsequent output will be transparent. |
| FontUnderline | Holds either True or False to determine whether subsequent output will be underlined. |
| ForeColor | Specifies the foreground color of printed text and graphics. (The paper determines the background color.) |
| hDC | A Windows device context handle for advanced Windows procedure calls. |
| Height | Holds the height, in twips, of the current printed page. |
| Page | Contains the page number currently being printed and updated automatically by Visual Basic. |
| ScaleHeight | Specifies how many ScaleMode units high that each graphic will be upon output. |
| ScaleLeft | Specifies how many ScaleMode units from the left of the page where subsequent printed output appears. |
| ScaleMode | Sets the unit of measurement for all subsequent printed output that appears. |
| ScaleTop | Specifies how many ScaleMode units from the top of the page where all subsequent printed output appears. |
| ScaleWidth | Specifies how many ScaleMode units wide that each graphic will consist of upon printed output. |
| TwipsPerPixelX | Specifies the number of screen twips that each printer's dot (called a pixel) height consumes. |
| TwipsPerPixelY | Specifies the number of screen twips that each printer's dot, or pixel, width consumes. |
| Width | Holds the size of the page width (measured in twips). |

There are lots of Printer properties, as shown in Table 21.1. Luckily, you'll use only a few of the properties for most of your printing needs. The font-related Printer properties take care of just about almost all of your printing jobs that are textual in nature.

**Note:** The graphics-related Printer properties and methods aren't covered in this unit. Once you master graphics in the next unit, you'll be more prepared to understand the graphics-related Printer properties. Most of the Printer's properties are reserved for controlling extremely advanced graphics output. For typical applications, you'll rarely bother to specify any properties because the default values work well for normal reporting requirements.

---

Unlike most of Visual Basic's control objects, the Printer object's methods are much more important than the property values. Table 21.2 contains a complete list of the methods supported by Visual Basic's Printer object.

**Table 21.2. The Printer object's methods.**

| *Method* | *Description* |
|---|---|
| Circle | Draws a circle, ellipse, or arc on the printer |
| EndDoc | Releases the current document, in full, to the Print Manager for output |
| Line | Draws lines and boxes on the page |
| NewPage | Sends a page break to the printed output so that subsequent output appears on the next page |
| Print | Prints numeric and text data on the printer |
| PSet | Draws a graphical point on the printed output |
| Scale | Determines the scale used for measuring output |
| TextHeight | Determines the full height of text given in the scale set with Scale |
| TextWidth | Determines the full width of text given in the scale set with Scale |

---

**Tip:** By far the most widely used Printer methods are the Print, EndDoc, and NewPage methods. Once you master these three methods, you'll rarely need to use any other methods.

---

***Review:*** There are several properties and methods available for the Printer object. Unless you need to send graphics to the printer using advanced graphics capabilities, you'll need only two or three of the properties and methods to fulfill all your printing needs.

## The *Print* Method

***Concept:*** The Printer object's Print method handles almost all printed output. Print supports several different formats. With Print, you can print messages, variables, constants, and expressions on the printer.

The Print method is, by far, the most commonly used printing method in Visual Basic. By mastering the Print method, you will have mastered the single most important printing method that you can master.

---

**Note:** Remember that a method is nothing more than a command that you apply to a particular object. For example, the AddItem method is a method that you've seen used throughout this book to add items to list and combo box controls.

---

Here is the format of the Print method:

```
[Printer.]Print [Spc(n) | Tab(n)] Expression [; | ,]
```

The format makes Print look a lot more confusing that it really is, but the portion of the Print method that appears to the right of Print takes some explanation. The next several sections explain the various options available for the Print method.

## Printing Constants

The Print method easily prints string and numeric constants. To print a string or numeric constant, place the constant to the right of the Print method. The following methods send the numbers 1, 2, and 3 to the printed output:

```
printer.Print 1

Printer.Print 2

Printer.Print 3
```

When execution hits these three lines of code, Visual Basic sends 1, 2, and 3 to the Printer object with each number appearing on subsequent lines. Every Print method sends a carriage return and line feed sequence to the printer. A lone Print method on a line by itself such as this one:

```
printer.Print
```

sends a blank line to the printer.

---

**Note:** Visual Basic always adds a space before all positive numeric values printed on the page. The space is where an invisible plus sign appears.

---

The following Print method sends two lines of text to the Printer object:

```
printer.Print "Visual Basic makes writing programs"

Printer.Print "for Windows
easy."
```

When the Windows Print Manager gets these two lines of output, the following appears on the printer's paper:

```
Visual Basic makes writing programs

for Windows easy.
```

## Printing Variables and Controls

In addition to constants, the Print method prints the contents of variables and controls. The following initializes a string and integer variable and then prints the contents of the variables on the printer:

```
FirstName = "Charley"

Age = 24

Printer.Print FirstName

Printer.Print Age
```

Here is the output produced by these Print methods:

```
Charley

 24
```

---

**Note:** Remember that Visual Basic won't send anything to the Printer object until the code that contains Print executes. You would insert Print methods at appropriate places in the code's procedures where printed output is required. For example, if there is a command button labeled Print Report, that command button's Click

procedure would contain Print methods.

---

## Printing Expressions

If you could print only individual strings, numeric constants, and variables, Print would be extremely limiting. Of course, Print is not that limited. You can combine constants, variables, and expressions to the right of Print methods to produce more complex printed output. The following Print method prints 31:

```
printer.Print 25 + (3 * 2)
```

The expression can contain both variables, controls, and constants, like this:

```
printer.Print
Factor * lblWeight.Caption + 10
```

If you want to send special characters to the printer, you can do that by using the Chr$() function. The following expression produces a message that includes embedded quotation marks inside the printed string:

```
printer.Print "She said, " & Chr$(34) & "I do." & Chr$(34)
```

When execution reaches the former Print method, this is what the Print Manager sends to the printer:

```
She said, "I do."
```

---

**Note:** You wouldn't be able to print the quotation marks without the Chr$() function. Usually, Visual Basic suppresses quotation marks when printing string constants.

---

## Printing Multiple Values

---

*Definition:* A *print zone* occurs every 14 columns on the page.

When you need to print several values on one line, you can do so by separating those values with semicolons and commas. The semicolon forces subsequent values to appear right next to each other in the output. The comma forces values to appear in the next print zone.

The following two messages print on different lines:

```
printer.Print "The sales were "

Printer.Print 4345.67
```

By using the semicolon, you can force these values to print next to each other:

```
printer.Print "The sales were "; 4345.67
```

The semicolon also acts to keep automatic carriage return and line feeds from taking place. The following Print method ends with a trailing semicolon:

```
printer.Print "The company name is ";
```

The trailing semicolon keeps the printer's print head at the end of the message for subsequent output. Therefore, the subsequent Print statement shown next, no matter how much later in the code the Print appears, would print its output right next to the previous Print's output:

```
printer.Print lblComName.Caption ' Complete the line
```

The semicolon is nice for printing multiple values of different data types of the same line. The following Print prints all of its data on the same line of output:

```
printer.Print "Sales:"; totSales;
"Region:"; RegNum
```

The comma is still sometimes used to force subsequent values to print in the next print zone. The following Print prints names every 14 spaces on the printed line:

```
printer.Print DivName1, DivName2, DivName3, DivName4
```

No matter how long or short each division name is, the next division name will print in the next print zone.

The previous Print might produce output similar to the following:

```
North NorthWest South SouthWest
```

---

**Tip:** When you print lists of numbers or short strings, the comma allows you to easily align each column.

---

## Utilize the Fonts

Most Windows-compatible printers support a variety of fonts. The font-related properties are often useful for printing titles and other special output messages in special font sizes and styles.

You can add special effects to your printed text by setting the font modifying properties from Table 21.1. For example, the following code first puts the printer in a boldfaced, italicized, 72-point font (a print size of one full inch), and then prints a message:

```
printer.FontBold = True

Printer.FontItalic = True

Printer.FontSize = 72

Printer.Print "I'm learning Visual Basic!"
```

---

**Note:** The font properties affect *subsequent* output. Therefore, if you print several lines of text and then change the font size, the text that you've already printed remains unaffected. Visual Basic prints only the subsequent output with the new font.

---

## Better Spacing with *Spc()* and *Tab()*

The Print method supports the use of the embedded Spc() and Tab() functions to give you additional control over your program's output. Spc() produces a variable number of spaces in the output as determined by the

argument to Spc(). The following Print prints a total of ten spaces between the first name and the last:

```
printer.Print FirstName; Spc(10); LastName
```

The argument that you send to the embedded Tab() function determines in which column the next printed character appears. In the following Print, the date appears in the 50th column on the page:

```
printer.Print Spc(50); DateGenerated
```

As these examples show, if you print values before or after the Spc() and Tab() functions, you separate the functions from the surrounding printed values using the semicolon.

---

**Tip:** Spc() and Tab() give you more control over spacing than the comma and semicolon allow.

---

***Stop and Type:*** Listing 21.2 contains some code that computes and prints two housing pricing and taxation values.

***Review:*** Use Spc() and Tab() to control the printer's output spacing.

**Listing 21.2. Using Spc() and Tab().**

```
1: Tax1 = TaxRate * HouseVal1

2: Tax2 = TaxRate * HouseVal2

3:

4: TotalVal = HouseVal1 + HouseVal2

5: TotTaxes = TaxRate * TotalVal

6:

7: Printer.Print "House Value"; Tab(20); "Tax"

8: Printer.Print
Format$(HouseVal1, "Currency");
```

```
9: Printer.Print Tab(20); Format$(Tax1, "Currency")

10: Printer.Print Format$(HouseVal2, "Currency");

11: Printer.Print Tab(20); Format$(Tax2, "Currency")

12:

13: Printer.Print '
Prints a blank line

14:

15: Printer.Print "Total tax:"; Spc(5); Format$(TotTaxes, "Currency")
```

***Output:*** Here is a sample of what you may see on the paper after Listing 21.2 executes:

```
House Value Tax

$76,578.23 $9,189.39

$102,123.67 $12,254.81

Total tax: $21,444.20
```

***Analysis:*** The Tab(20) function calls in lines 7, 9, and 11 ensure that the second column that contains the tax information is aligned. Also, notice that the trailing semicolons on lines 8 and 10 allow you to continue the Print methods on subsequent lines without squeezing long Print method values onto the same line.

Line 13 prints a blank line. Line 15 uses the Spc() function to insert five spaces between the title and the total amount of tax.

## Initiating the Print

***Concept:*** The physical printing doesn't begin until all output is released to the Print Manager, or until your application issues an EndDoc method.

As you send Print methods to the Print Manager via the Printer object, the Print Manager builds the page or pages of output but doesn't release that output until you issue an EndDoc method. EndDoc tells the Print

Manager, "I'm done sending output to you, you can print now."

Without EndDoc, Windows would collect all of an application's output and not print any of the output until the application terminates. If you were to write an application that the user runs throughout the day and that prints invoices as customers make purchases, you would need to issue an EndDoc method at the end of each invoice-printing procedure if you wanted each invoice to print at that time.

***Stop and Type:*** Listing 21.3 prints a message on the printer and then signals to the Print Manager that the output is ready to go to paper. Without the EndDoc, the Print Manager would hold the output until the application containing the code terminates.

***Review:*** The EndDoc method tells the Print Manager to release all printed output.

**Listing 21.3. The EndDoc tells the Print Manager to release the output.**

```
1: Printer.Print "Invoice #"; invNum

2: Printer.Print "Customer:"; cust(CCnt); Tab(20); "Final Sales"

3: Printer.Print "Amount of sale:"; Tab(20); Format$(SaleAmt,
"Currency")

4: Printer.Print "Tax:"; Tab(20); Format$(tax, "Currency")

5: Printer.Print

6: Printer.Print "Total:"; Tab(20); Format$(TotalSale, "Currency")

7:

8: ' Release the job for actual printing


9: Printer.EndDoc
```

***Analysis:*** The program containing Listing 21.3's code might continue to run and process other sets of data. The EndDoc method triggered by line 9 ensures that the output built in the preceding Print methods all gets sent to the physical printer immediately. If other Print methods appear later in the program, the Print Manager will begin building the output all over again, releasing that subsequent output only for an EndDoc procedure or when the application ends.

# Page Breaks

***Concept:*** When printing to the printer, you must be careful to print at the top of a new page when you want the output to advance one page. The NewPage method forces the printer to eject the current page and begin subsequent output on the next new page.

The Windows Print Manager ensures that each printed page properly breaks at the end of a physical page. Therefore, if the printer's page length is 66 lines and you print 67 lines, the 67th line will appear at the top of the second page of output.

There are times, however, when you need to print less than a full page on the printer. You can release that incomplete page for printing using the NewPage method (from Table 21.1). To use NewPage, simple apply the NewPage method to the Printer object like this:

```
printer.NewPage
```

---

**Note:** Remember that you actually print to the Windows Print Manager and that your application's output methods don't directly control a physical printer. Therefore, NewPage tells the Print Manager to go the a new page when the Print Manager gets to that location in the output.

---

You've got to remember that you're working with printers that support many fonts and font sizes. You can always determine, in advance, how many lines of output will fit on a single page as long as you first check the value of the following formula:

```
numLinesPerPage = Printer.Height / Printer.TextHeight("X")
```

As explained in Table 21.1, the Height property determines the height, in twips, of the page. The TextHeight property determines the full height of a printed character (including *leading*, which is the area directly above and below characters). TextHeight measures the height in twips if you haven't changed the scale using the ScaleMode property.

For printed reports, you'll rarely use the ScaleMode method. If you have the need to change the scale of measurement, however, you'll have to change the scale back to twips before calculating the number of output lines per page, like this:

```
printer.ScaleMode = TWIPS
```

The ScaleMode accepts values defined in Table 21.3. As long as you add the CONSTANT.TXT file to your application's Property window, you can use the named constants in place of the numeric values if you want to change the scale measurement.

**Table 21.3. The ScaleMode values.**

| *Value* | *Named Constant* | *Description* |
|---|---|---|
| 0 | USER | A user-defined value |
| 1 | TWIPS | Measured in twips (the default) |
| 2 | POINTS | Measured in points |
| 3 | PIXELS | Measured in pixels (the smallest unit addressable by your printer) |
| 4 | CHARACTERS | Measured in characters (120 by 240 twips) |
| 5 | INCHES | Measured in inches |
| 6 | MILLIMETERS | Measured in millimeters |
| 7 | CENTIMETERS | Measured in centimeters |

***Stop and Type:*** Listing 21.4 contains code that prints two messages, one per page of printed output.

***Review:*** At any point during your application's printing, you can issue an NewPage method to force the printer to eject the current page and begin printing at the top of the next page.

**Listing 21.4. The NewPage forces the printer to eject the current page.**

```
1:
Printer.Print "The Report begins on the next page..."

2: Printer.NewPage ' Go to top of new page

3: Printer.Print "The Campaign Platform"
```

***Analysis:*** Line 2 ejects the printer even if the printer has not yet printed a full page.

## Homework

### General Knowledge

1. What File command does the Visual Basic Primer system not support?
2. What job does the Windows Print Manager play in printing your application's output?
3. Why doesn't your program need to understand every printer's individual command?
4. What does *online* mean?
5. How can you ensure that the user prepares the printer before you print reports from a Visual Basic application?

6. What is the Printer object?
7. True or false: The Printer object supports properties and methods just as many of the controls do.
8. What is a *pixel*?
9. What is the most commonly used Printer method?
10. True or false: You can print variables and constants but not expressions with Print.
11. True or false: There is no difference between using several embedded Tab(14) functions and using commas to separate columns of output in a Print method.
12. Printing three strings separated by commas causes the strings to appear in columns 1, 15, and 29. Why do you think that printing three positive numbers, separated by commas, causes the numbers to print in columns 2, 16, and 30?
13. What is a *print zone*?
14. True or false: A Print that isn't terminated with a trailing semicolon always prints a carriage return and line feed on the printer.
15. How can you print quotation marks onto paper?
16. True or false: If the last thing an application does before terminating is print a report, there is no need to specify the EndDoc method at the end of the printing.
17. What is meant by *leading*?

## Find the Bug

1. Poor Caroline isn't having luck getting a report to print properly. Caroline learned in her early days of computing that a printed page has 66 lines of text. Now that Caroline has upgraded to Windows and is practicing to become a top-notch Visual Basic programmer, Caroline's 66-line per printed page doesn't seem to be true all of the time. Explain to Caroline what she must do to fix her problem.
2. Mike is working on two programs. For some reason, Mike can't figure out why these two different output lines don't produce exactly the same output. Help Mike solve this dilemma.
   Printer.Print Tab(10); "Visual Basic"
   Printer.Print Spc(10); "Visual Basic"

## What's the Output?

1. What's the output from the following two Print methods?
   Printer.Print "Line 1";
   Printer.Print "Line 2"
2. What does the following Print method produce on the printer when the Print Manager releases the line to be printed?
   Printer.Print "The Spanish N is "; Chr$(164)

## Write Code That...

1. Write a line of Visual Basic code that prints the word America in column 57.

## Extra Credit

Write a program that prints the ASCII characters (print only the values from ASCII 32 to 255) on a piece of paper when the user presses a command button.

# Visual Basic in 12 Easy Lessons



# Lesson 11, Unit 22

## Glitzy Graphics

### What You'll Learn

- The line and shape controls
- Mastering the line control
- Mastering the shape control
- If you have graphics files...

Most Visual Basic programmers enjoy learning and using the material presented in this lesson. This lesson searches for that artist deep within every programmer! This lesson explores the graphics controls available to you as a Visual Basic programmer.

Perhaps you won't be drawing pretty pictures on every application's form that you design, but knowing how to draw lines and circles often allows you to spruce up a form by separating controls and enclosing labels within colored ovals to highlight important information.

# The Line and Shape Controls

***Concept:*** The line and shape controls work together to draw lines, boxes, and all kinds of circular figures on the form. By placing the controls and setting appropriate properties, you'll be adding flair to applications.

Figure 22.1 shows the location of the line and shape controls on the Toolbox window. The properties of each control that you place on your form determine exactly what kind of image the control becomes.

**Figure 22.1. The line and shape controls provide Visual Basic's artistic tools.**

Here are the primary graphic images that you can draw with the line and shape controls:

- Lines
- Rectangles
- Squares
- Ovals
- Circles
- Rounded rectangles
- Rounded squares

Figure 22.2 shows each of these seven primary images. By combining these fundamental geometric images and setting appropriate color and size properties, you can draw virtually anything you need to draw on the form.

**Figure 22.2. The seven fundamental images that you can draw.**

Use the line control for drawing lines of various widths, lengths, and patterns. The shape control handles the drawing for all the other fundamental shapes.

***Review:*** The line and shape controls are the primary drawing controls. There are seven fundamental geometric shapes that you can draw. By specifying various properties, you can control how those shapes appear on the form.

# Mastering the Line Control

***Concept:*** The line control contains properties that specify the width and length of lines that you draw. In addition, you can change the pattern of each line that you draw.

Table 22.1 lists the property values for the line control. The BorderStyle property requires its own table for explanation. Table 22.2 contains the values that you can specify for the BorderStyle. The BorderStyle property determines the pattern that Visual Basic uses to draw the line. By specifying various BorderStyle values, you can vary the line pattern. If you assign a BorderStyle property at runtime, you either can specify a number that represents the BorderStyle, or use one of the constants defined in the

CONSTANT.TXT file.

**Table 22.1. The line control properties.**

| *Property* | *Description* |
|---|---|
| BorderColor | Specifies a hexadecimal Windows color value that determines the color of the line. |
| BorderStyle | Contains one of seven values that specifies the pattern of the drawn line. See Table 22.2 for available BorderStyle values. The default value is 1-Solid. The BorderStyle has no effect on lines with a BorderWidth greater than one twip. |
| BorderWidth | Specifies the size, in twips, that the line takes. |
| DrawMode | An advanced style that determines how the bit patterns of the line interact with the surrounding form's bit appearance. The default value, 13 - Copy Pen, works well for virtually all Visual Basic applications. |
| Index | Specifies the subscript of the line control if the line control is part of a control array. |
| Name | Contains the name of the line control. Visual Basic names the line controls Line1, Line2, and so on, unless you rename the line controls. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the line control. |
| Visible | Holds True or False, indicating whether the user can see the line control. |
| X1 | Contains the number of twips from the left of the Form window to the start of the line. |
| X2 | Contains the number of twips from the left of the Form window to the end of the line. |
| Y1 | Contains the number of twips from the top of the Form window to the left starting point of the line. |
| Y2 | Contains the number of twips from the top of the Form window to the lower ending point of the line. |

**Table 22.2. The line control's BorderStyle values.**

| *Value* | *CONSTANT.TXT Name* | *Description* |
|---|---|---|
| 0-Transparent | TRANSPARENT | Background comes through the line |
| 1-Solid | SOLID | The line is a solid line |
| 2-Dash | DASH | The line is comprised of dashes |
| 3-Dot | DOT | The line is comprised of dots |
| 4-Dash-Dot | DASH_DOT | The line is comprised of a series of dashes followed by dots |
| 5-Dash-Dot-Dot | DASH_DOT_DOT | The line is comprised of a series of one dash followed by two dots |
| 6-Inside Solid | INSIDE_SOLID | Same as 1-Solid for lines |

Figure 22.3 shows how various BorderStyle settings affect the lines that you draw. The BorderStyle

determines how a series of dashes and dots comprise the line's pattern. (Is this Morse code we're speaking here?)

**Figure 22.3. Affects of the BorderStyle properties.**

To draw a line, double-click the line control on the toolbox. A line appears in the center of the form with two handles on each end. To move the line to a different location, drag the center of the line with the mouse. To lengthen or shorten the line, drag either handle on the line. You can raise and lower either end of the line by dragging either end's handle with the mouse.

**Tip:** After you position the line with the mouse in the approximate location you need the line to appear, you can fine-tune the line's size and location by setting the various property values. If you're a patient programmer, you can even animate the lines by changing the X1, X2, Y1, and Y2 property settings repeatedly through code.

***Stop and Type:*** Figure 22.4 contains the Form window that might be used by a local hardware store for its regular program screen. The entire house was drawn using a series of lines. You can find this form inside the MOMPOP.MAK application on this book's disk.

---

**Note:** The only function connected to this form is the End statement inside the cmdExit_Click() command button's event procedure.

---

**Figure 22.4. Lines add eye-catching glamour to a dull form.**

***Review:*** By using the line control and specifying various property values, you can draw shapes on the form's background.

## Mastering the Shape Control

***Concept:*** The shape control gives you the ability to draw six different kinds of figures on the form. The various shading and color properties help you distinguish one shape from another.

Table 22.3 contains the properties available for the shape control. The most important property is the Shape property. The Shape property gives a shape one of the six fundamental shapes.

**Table 22.3. The shape control properties.**

| *Property* | *Description* |
|---|---|
| BackColor | Specifies a hexadecimal Windows color value that determines the background color of the shape. |
| BackStyle | Contains either 0-Transparent (the default) or 1-Opaque that determines whether the background of the form appears through the shape or if the shape hides whatever it covers. |

| | |
|---|---|
| BorderColor | Specifies a hexadecimal Windows color value that determines the color of the shape's bordering edges. |
| BorderStyle | Contains one of seven values that specifies the pattern of the shape's border. Table 22.2's line control's BorderStyle values provide the shape's BorderStyle possible values as well. The default value is 1-Solid. The BorderStyle has no effect on shapes with a BorderWidth greater than one twip. |
| BorderWidth | Specifies the size, in twips, that the shape's outline takes. |
| DrawMode | An advanced style that determines how the bit patterns of the shape interact with the surrounding form's bit appearance. The default value, 13 - Copy Pen, works well for virtually all Visual Basic applications. |
| FillColor | Specifies a hexadecimal Windows color value that determines the color of the shape's interior lines. |
| FillStyle | Contains one of eight values that specifies the pattern of lines with which Visual Basic paints the interior of the shape. Table 22.4 contains the possible values for the shape's FillStyle. The default FillStyle value is 0-Solid. |
| Height | Specifies the number of twips high that the shape takes (from the highest point to the lowest point in the shape). |
| Index | Specifies the subscript of the shape control if the shape control is part of a control array. |
| Left | Specifies the number of twips from the form's left edge to the shape's leftmost edge. |
| Name | Contains the name of the shape control. Visual Basic names the line controls Shape1, Shape2, and so on, unless you rename the shape controls. |
| Shape | Contains one of six values that specifies the type of shape that the shape control takes on. Table 22.5 contains the possible values for the shape's Shape property. The default Shape property is 0-Rectangle. |
| Tag | Unused by Visual Basic This is for the programmer's use for an identifying comment applied to the shape control. |
| Top | Specifies the number of twips from the form's top edge to the shape's highest edge. |
| Visible | Holds True or False, indicating whether the user can see the shape control. |
| Width | Specifies the number of twips wide that the shape takes (at the widest axis). |

Table 22.4 contains the possible values for the shape control's FillStyle property. Figure 22.5 shows you the various fill patterns that a shape can contain.

**Figure 22.5. The FillStyle determines how the shapes' interiors appear.**

**Table 22.4. The shape control's FillStyle values.**

| *Value* | *CONSTANT.TXT Name* | *Description* |
|---|---|---|
| 0-Solid | SOLID | Solid color fill with no pattern |
| 1-Transparent | TRANSPARENT | The shape appears as an outline only |
| 2-Horizontal Line | HORIZONTAL_LINE | Horizontal lines fill the shape |
| 3-Vertical Line | VERTICAL_LINE | Vertical lines fill the shape |

| | | |
|---|---|---|
| 4-Upward Diagonal | UPWARD_DIAGONAL | Upward diagonal lines fill the shape |
| 5-Downward Diagonal | DOWNWARD_DIAGONAL | Downward diagonal lines fill the shape |
| 6-Cross | CROSS | Crosshairs fill the shape |
| 7-Diagonal Cross | DIAGONAL_CROSS | Diagonal crosshairs fill the shape |

Table 22.5 contains the possible values for the shape control's Shape property. Figure 22.2 showed you the various shapes that the shape controls can take on. Therefore, when you want to place a square on a form, you'll place the shape control on the form and set the Shape property to 1-Square.

**Table 22.5. The shape control's Shape properties.**

| *Value* | *CONSTANT.TXT Name* | *Description* |
|---|---|---|
| 0-Rectangle | SHAPE_RECTANGLE | A rectangle |
| 1-Square | SHAPE_SQUARE | A square |
| 2-Oval | SHAPE_OVAL | An oval |
| 3-Circle | SHAPE_CIRCLE | A circle |
| 4-Rounded Rectangle | SHAPE_ROUNDED_RECTANGLE | A rectangle with rounded corners |
| 5-Rounded Square | SHAPE_ROUNDED_SQUARE | A square with rounded corners |

***Stop and Type:*** Figure 22.6 provides a simple shape application that changes the shape as the user presses the command button. You can load and run the SHAPECH.MAK application on this book's disk if you want to study the application.

**Figure 22.6. Changing the shape with the click of a button.**

***Review:*** The shape control provides six of the seven fundamental geometric shapes that you can place on a form. The Shape property determines the outline of the shape.

## If You Have Graphics Files...

***Concept:*** Visual Basic supports two additional graphics controls that don't draw graphics but with which you can place icons and graphic bitmap images that you may have on the disk.

Figure 22.7 shows the location of two additional graphics-related controls called the picture box control and the image control. These two controls are virtually identical and enable you to load disk images onto your application's form.

---

**Note:** The picture box control provides a few advanced features that you can tap into if you ever write *MDI* (*multiple document interface*) applications. The image box control is more efficient and displays images faster than the picture box.

---

**Figure 22.7. The picture box and image control on the toolbox.**

Here are the three kinds of graphics files that these controls can load from the disk and display on a form:

- *Bitmap files* that end with the .BMP or .DIB filename extensions
- *Icon files* that end with the .ICO filename extension
- *Metafiles* that end with the .WMF filename extension

Where do you get these graphic files? None come with the Visual Basic Primer system (graphic images consume lots of disk space). There are several sources available. If you use the Microsoft Paintbrush program supplied with Windows, you can create bitmap files. Icon files are compact graphics files that contain the picture icons that you see in a Windows Program Manager group. Several various Windows applications store graphic images in the metafile format.

Table 22.6 contains the vital and common properties available for both the picture box and image controls. The most important value is the Picture property that contains the complete filename and pathname to the picture displayed inside the control.

**Table 22.6. The picture box and image controls' properties.**

| *Property* | *Description* |
|---|---|
| BorderStyle | Describes the style of the border around the image. If set to 0-None (the default), no border appears around the image. If 1-Fixed Single, a dark border line appears around the image. |
| Height | Contains the height, in twips, of the image. (See Stretch.) |
| Left | Holds the number of twips from the left edge of the Form window to the left edge of the image. |
| Name | Contains the name of the picture box or image. Visual Basic names the picture box controls Picture1, Picture2, and so on, and Visual Basic names the image control Image1, Image2, and so on unless you rename the controls. |
| Picture | Contains a complete path and filename to the image that Visual Basic will display in the control at runtime. No image appears inside the control until the user runs the program. |
| Stretch | (For image controls only.) If set to False, (the default) the control resizes automatically to fit the size of the graphic image. No matter what size you specify for the image control, the control will resize to the exact size of the disk file's graphic image as if to *shrinkwrap* around the image. If set to True, the image resizes to fit the shape of the control. |
| Width | The width, in twips, of the control. (See Stretch.) |

When you want to put a graphic image on your form, you only need to place an image (or picture box) control on the form and specify a filename for the image that you want the control to display. Open a new form and follow these steps to create an application with an image:

1. Double-click the image control to place the control in the middle of the form.
2. Press F4 to display the Property window.
3. Click the Picture property. You'll see an ellipsis appear where you normally enter property values.
4. If you click the ellipsis, Visual Basic opens a Load Picture dialog box like the one shown in Figure

22.8. In the dialog box, you specify any file, including the path, that holds the image you want to display on your form.
The Picture property continues to hold the pathname and filename of the image that you place on the form. At runtime, Visual Basic will display the image in the place of the control.

**Figure 22.8. The Load Picture dialog box for filling in the Picture property.**

You also can change a graphic file that appears in an image or picture box control using Visual Basic code. The LoadPicture function loads specific graphic files into graphic controls to display those graphics on the form. The following statement changes the picture file used for an image control named imgFace:

```
imgFace.Picture = LoadPicture("D:\FIGS\FACE.BMP")
```

You can replace pictures during runtime by loading different files using the LoadPicture() function.

**Tip:** By using a null string, "", for the LoadPicture() argument, you can erase a picture from a picture box or image control.

***Review:*** This section wraps up the discussion on combining graphics with Visual Basic programs. Not only does Visual Basic provide the tools needed to draw your own pictures, but Visual Basic also enables you to load images from disk files onto your own forms.

# Homework

## General Knowledge

1. What two controls enable you to draw geometric shapes on a form?
2. How many different shapes does the shape control produce?
3. Which property value determines the pattern of drawn lines?
4. Which property value determines the shape of objects drawn with the shape control?
5. What two controls enable you to load graphic file images onto a form?
6. Which graphic image control is the most efficient of the two controls?
7. What three kinds of graphics files can Visual Basic display on a form?
8. Which property forces a graphic image to shrinkwrap around the picture box or image control?
9. Which control property determines the path and filename of the graphic image to display?
10. What is the name of the Visual Basic function that loads graphic file images at runtime?

## Find the Bug

1. Jean has drawn a thick line that measures 15 twips. Jean wants to use the line to separate a large form message from the rest of the controls on her form. The trouble that Jean can't seem to overcome is that she wants the thick line to appear as a series of dashes. Visual Basic refuses to honor her BorderStyle request and always displays a solid line. Explain to Jean what her problem is.
2. Can you think of a roundabout way to accomplish Jean's thick, dashed line?

## What's the Output?

1. Gordon wrote a program that contains the following statement:
   imgPainting.Picture = LoadPicture("")
   What appears on the image control when Gordon's application calls the LoadPicture() function?

## Write Code That...

1. Suppose that you want to draw a rectangle with a blue border, red diagonal lines, and a green interior. Describe the shape control's properties that you would have to set to a blue, red, and green color.

## Extra Credit

Write a program that draws a large green happy face in the center of the form. Add a command button that the user can click to blink one of the happy face's eyes.

# Visual Basic in 12 Easy Lessons



# Project 11

## Spruce Up your Programs

### Stop & Type

This lesson taught you how to use the printer for text reports and use the screen for fancy graphics. Although business and scientific data processing requires printed reports, this unit's real glamour appeared in Unit 22 when you learned all about the line and shape controls.

The *visual* in Visual Basic becomes most apparent when you add graphics that capture the user's eye. Perhaps the control that provides more different formats than any other is the shape control, which enables you to draw squares, circles, and ovals of all shapes, sizes, and colors. This unit takes the shape control to the max by demonstrating a fancy form that blows up in your face with more than 150 sets of colored circles, squares, and ovals.

In this lesson, you saw the following:

- When to access the Printer object
- How to use the Print method to produce text on the printed page
- What shapes are available from the shape control
- How to draw lines on the form using the line control
- When to use the picture box and image controls to place file graphics on a form

#### The Program's Description

Figure P11.1 shows the PROJEC11.MAK opening Form window. Wow! When you load and run the program, the program generates all kinds of circles on the form. The entire collection of circles, as well as the subsequent graphic collections that appear after the circles, all are different owing to the program's use of the Rnd() function introduced in project 10.

**Figure P11.1. Project 11 runs circles around other applications.**

As you click the Next Shape command button, the program replaces the circles with a similar mosaic of squares. Pressing Next Shape again produces a series of ovals. You can continue cycling through the circles, squares, and ovals as often as you want until you press the Exit command button to terminate the program.

The program contains a control array with 160 shape controls named shpShape. Figure P11.2 shows the form at design time. The location of the shape controls at design time is irrelevant to the program because the code relies heavily on the Rnd() function to display the shapes in random screen locations, with random colors, sizes, and fill patterns.

**Figure P11.2. project 11's design mode shows an interesting placement of a control array containing shapes.**

The program depends so much on the Rnd() function that a special function subroutine is included that converts Rnd() into a random number, from 1 to whatever argument is passed to the function. The function procedure's name is RndNum(); you'll see the function in the code description that appears in Listing P11.1.

**Listing P11.1. The code that generates all the graphics on the form.**

```
1: Option Explicit

2:

3: Sub Form_Activate ()

4: ' Get
things started by displaying circles

5: Call Circles

6: End Sub

7:

8: Sub Circles ()

9: ' Draws several circles of different
```

```
10: ' colors and sizes on the form

11: Dim Count As Integer

12: lblShape.Caption = "Circles"

13: For Count = 0
To 159 ' 160 total shapes

14: shpShape(Count).Shape = SHAPE_CIRCLE

15: shpShape(Count).Visible = False

16: shpShape(Count).Top = RndNum(3000)

17: shpShape(Count).Width = RndNum(5000)

18: shpShape(Count).Height = RndNum(3000)

19: shpShape(Count).Left
= RndNum(3000)

20: shpShape(Count).BackColor = QBColor(RndNum(16) - 1) ' Adjust for zero

21: shpShape(Count).FillColor = QBColor(RndNum(16) - 1) ' Adjust for zero

22: shpShape(Count).Visible = True

23: shpShape(Count).FillStyle = RndNum(8) - 1 ' Adjust
for zero

24: Next Count

25: End Sub

26:
```

```
27: Sub Ovals ()

28: ' Draws several ovals of different

29: ' colors and sizes on the form

30: Dim Count As Integer

31: lblShape.Caption = "Ovals"

32: For Count = 0 To 159 ' 160 total shapes

33:
shpShape(Count).Shape = SHAPE_OVAL

34: shpShape(Count).Visible = False

35: shpShape(Count).Top = RndNum(3000)

36: shpShape(Count).Width = RndNum(4000)

37: shpShape(Count).Height = RndNum(3000)

38: shpShape(Count).Left = RndNum(3000)

39:
shpShape(Count).BackColor = QBColor(RndNum(16) - 1) ' Adjust for zero

40: shpShape(Count).FillColor = QBColor(RndNum(16) - 1) ' Adjust for zero

41: shpShape(Count).Visible = True

42: shpShape(Count).FillStyle = RndNum(8) - 1 ' Adjust for zero

43: Next
Count

44: End Sub
```

```
45:

46: Sub Squares ()

47: ' Draws several squares of different

48: ' colors and sizes on the form

49: Dim Count As Integer

50: lblShape.Caption = "Squares"

51: For Count = 0 To 159 ' 160 total shapes

52:
shpShape(Count).Shape = SHAPE_SQUARE

53: shpShape(Count).Visible = False

54: shpShape(Count).Top = RndNum(3000)

55: shpShape(Count).Width = RndNum(5000)

56: shpShape(Count).Height = RndNum(3000)

57: shpShape(Count).Left = RndNum(3000)

58:
shpShape(Count).BackColor = QBColor(RndNum(16) - 1) ' Adjust for zero

59: shpShape(Count).FillColor = QBColor(RndNum(16) - 1) ' Adjust for zero

60: shpShape(Count).Visible = True

61: shpShape(Count).FillStyle = RndNum(8) - 1 ' Adjust for zero

62: Next
```

```
Count

63: End Sub

64:

65: Sub cmdShape_Click ()

66: ' Check the caption of the description

67: ' label to see what shapes to draw next

68: Select Case Left$(lblShape.Caption, 1)

69: Case "C":

70: Call Squares

71: Case "S"

72:
Call Ovals

73: Case "O":

74: Call Circles

75: End Select

76: End Sub

77:

78: Sub cmdExit_Click ()

79: End

80: End Sub
```

```
81:

82: Function RndNum (n)

83: ' Returns a random number from 1 to n

84: RndNum = Int(n * Rnd + 1)

85: End
Function
```

## Descriptions

1: Requires that all variables be defined before their use.

2: A blank line that helps separate parts of the program.

3: The code for the event procedure that executes as soon as the user sees the form for the first time.

4: A remark that explains the procedure.

---

4: Always remember to generate an initial form.

---

5: Execute the subroutine procedure that draws the circles.

6: Terminate the procedure.

7: A blank line separates procedures.

8: The subroutine procedure that draws circles.

9: A remark that explains the procedure.

10: The remark continues.

11: Define a variable that will control the loop.

12: Set the caption on the description label.

13: Step through all 160 elements of the control array.

---

13: There are 160 elements in the shape control array to display.

---

14: Set the shape of each control to a circle.

15: Hide the control, temporarily, until the properties are set.

16: Set an appropriate placement from the top of the form.

17: Set an appropriate width.

18: Set an appropriate height.

19: Set an appropriate placement from the left edge of the form.

20: Set the background color by calling the built-in QBColor() function and passing a value from 0 to 15.

21: Set the fill color by calling the built-in QBColor() function and passing a value from 0 to 15.

22: Display the control now that the size and placement properties are set.

23: Fill the shape with a fill number from 0 to 8.

24: Prepare for the next shape in the control array.

25: Terminate the procedure.

26: A blank line separates the procedures.

27: The subroutine procedure that draws ovals.

28: A remark that explains the procedure.

29: The remark continues.

30: Define a variable that will control the loop.

31: Set the caption on the description label.

32: Step through all 160 elements of the control array.

33: Set the shape of each control to a oval.

34: Hide the control, temporarily, until the properties are set.

35: Set an appropriate placement from the top of the form.

36: Set an appropriate width.

37: Set an appropriate height.

38: Set an appropriate placement from the left edge of the form.

39: Set the background color by calling the built-in QBColor() function and passing a value from 0 to 15.

---

39: The QBColor() function accepts a value from 0 to 15 that represents a different color.

40: Set the fill color by calling the built-in QBColor() function and passing a value from 0 to 15.

41: Display the control now that the size and placement properties are set.

42: Fill the shape with a fill number from 0 to 8.

43: Prepare for the next shape in the control array.

44: Terminate the procedure.

45: A blank line separates the procedures.

46: The subroutine procedure that draws squares.

47: A remark that explains the procedure.

48: The remark continues.

49: Define a variable that will control the loop.

50: Set the caption on the description label.

51: Step through all 160 elements of the control array.

52: Set the shape of each control to a square.

---

52: The named constants in CONSTANT.TXT eliminate the need to remember numeric property equivalents.

---

53: Hide the control, temporarily, until the properties are set.

54: Set an appropriate placement from the top of the form.

55: Set an appropriate width.

56: Set an appropriate height.

57: Set an appropriate placement from the left edge of the form.

58: Set the background color by calling the built-in QBColor() function and passing a value from 0 to 15.

59: Set the fill color by calling the built-in QBColor() function and passing a value from 0 to 15.

60: Display the control now that the size and placement properties are set.

61: Fill the shape with a fill number from 0 to 8.

62: Prepare for the next shape in the control array.

63: Terminate the procedure.

64: A blank line separates the procedures.

65: A procedure that executes to display the next set of shapes when the user clicks the cmdShape command button.

66: A remark that explains the procedure.

67: The remark continues.

68: Determine the next shape by looking at the label that describes the previously displayed shape.

---

68: The code looks at the first character of the label to decide what shape to display next.

---

69: If the label contains Circles...

70: Change the display to random squares.

71: If the label contains Squares...

72: Change the display to random ovals.

73: If the label contains Ovals...

74: Change the display to random circles.

75: Terminate the Select statement.

76: Terminate the procedure.

77: A blank line separates the procedures.

78: A procedure that ends the program when the user clicks the cmdExit command button.

79: Terminate the program's execution.

80: Terminate the procedure.

81: A blank line separates the procedures.

82: A function procedure that accepts an integer argument and that returns a random number from 1 to the argument.

83: A remark explains the function.

84: Set the function's return value to the generated random number.

---

84: A function procedure keeps you from having to repeat the same code throughout the program.

---

85: Terminate the procedure.

## Close the Application

You now can exit the application and exit Visual Basic. The next lesson teaches you how to use the remaining controls and Visual Basic's online debugging tool to help locate and correct errors within code.

# Visual Basic in 12 Easy Lessons



# Lesson 12, Unit 23

## The Scroll Bars, Grid, and Mouse

### What You'll Learn

- Scrolling the scroll bars
- Preparing for the grid control
- Using the grid control
- Monitoring the mouse cursor
- Capturing mouse clicks and movements

This unit wraps up the remaining controls that come with the Visual Basic Primer system by showing you ways to access and control special scroll bars and grid controls, as well as how to interpret the user's mouse movements and clicks. As with all the controls, understanding how to work with scroll bars and the grid requires that you master the properties related to those controls.

---

**Note:** The grid control requires a little preparation. In Lesson 2, you removed the grid control from the AUTOLOAD.MAK default project. Therefore, the grid control won't appear on your toolbox until you add the control once again.

---

# Scrolling the Scroll Bars

***Concept:*** The scroll bars give the user the ability to control changing values. Rather than type specific values, the user can move the scroll bars with the mouse to specify relative positions within a range of values.

Figure 23.1 shows you the location of the two scroll bar controls on the Toolbox window. There is a horizontal scroll bar and a vertical scroll bar control. In addition, you'll see two shapes whose width and height properties are being adjusted by the user's clicking in the scroll bar.

**Figure 23.1a. The scroll bars and their descriptions.**

**Figure 23.1b. The scroll bars and their descriptions.**

Table 23.1 contains a list of the scroll bar properties. The most unique and important property values for a scroll bar are the LargeChange, Max, Min, and SmallChange.

**Table 23.1. The scroll bar properties.**

| *Property* | *Description* |
|---|---|
| DragIcon | Specifies the icon that appears when the user drags the scroll bar around on the form. (You'll only rarely allow the user to move a scroll bar, so the Drag... property settings aren't usually relevant.) |
| DragMode | Contains either 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the scroll bar but that you, through code, can initiate the dragging if needed. |
| Enabled | If set to True (the default), the scroll bar can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| Height | Contains the height, in twips, of the scroll bar. |
| HelpContextID | If you add advanced, context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| Index | If the scroll bar is part of a control array, the Index property provides the numeric subscript for each particular scroll bar. |
| LargeChange | Specifies the amount that the scroll bar changes when the user clicks within the scroll bar's shaft area. |
| Left | Holds the number of twips from the left edge of the Form window to the left edge of the scroll bar. |
| Max | Indicates the maximum number of units that the scroll bar value represents at its highest setting. The range is from 1 to 32767 (the default). |
| Min | Indicates the minimum number of units the scroll bar value represents at its lowest setting. The range is from 1 (the default) to 32767. |
| MousePointer | Contains the shape that the mouse cursor changes to if the user moves the mouse cursor over the scroll bar. The possible values are from 0 to 12, and represent a range of different shapes that the mouse cursor can take on. |
| Name | Contains the name of the control. By default, Visual Basic generates the names VScroll1, VScroll2, and so on (for vertical scroll bars), and HScroll1, HScroll2, and so on (for horizontal scroll bars) as you add subsequent scroll bars to the form. |
| SmallChange | Specifies the amount that the scroll bar changes when the user clicks an arrow at either end of the scroll bar. |

| | |
|---|---|
| TabIndex | Determines that the focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, the user can press Tab to move the focus to this scroll bar. If False, the scroll bar can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the scroll bar. |
| Top | Holds the number of twips from the top edge of a scrollbar to the top of the form. |
| Value | Contains the unit of measurement currently represented by the position of the scroll bar. |
| Visible | Contains either True or False, indicating whether the user can see (and, therefore, use) the scroll bar. |
| Width | Holds the number of twips wide that the scroll bar consumes. |

**Tip:** Prefix the names of your horizontal scroll bars with the hsb prefix and your vertical scroll bars with the vsb prefix so that you can easily distinguish them from each other.

When you place a scroll bar on a form, you must decide at that time what range of values the scroll bar is to represent. The scroll bar's full range can extend from 1 to 32767. Set the Min property to the lowest value represented by the scroll bar. Set the Max property to the highest value represented by the scroll bar.

When the user eventually uses the scroll bar, the scroll bar arrows control small movements in the scroll bar's value determined by the SmallChange property. Clicking the empty shaft on either side of the scroll box produces a positive or negative change in the value represented by the LargeChange property. The user can drag the scroll bar itself to any position within the scroll bar shaft to jump to a specific location instead of changing the value gradually.

Suppose, for example, that a horizontal scroll bar was to represent a range of whole dollar amounts from $5 to $100. When the user clicks the scroll arrows, the scroll bar's value is to change by one dollar. When the user clicks the empty shaft on either side of the scroll box, the scroll bar's value is to change by five dollars. Here are the property values that you would set that determine how Visual Basic interprets each click of the scroll bar:

Min: 5

Max: 100

SmallChange: 1

LargeChange: 5

The physical size of the scroll bar has no bearing on the scroll bar's returned values when the user selects from the scroll bar. Adjust the scroll bars on your form so that the scroll bars are wide enough or tall enough to look appropriate sizes for the items that they represent.

***Stop and Type:*** Listing 23.1 contains the SCROLL.MAK code that you can load and run to adjust the circle and bar sizes that you saw in Figure 23.1.

***Review:*** There are two scroll bars, a horizontal scroll bar and a vertical scroll bar, that give the user the ability to select from a range of possible values without having to enter individual values.

**Listing 23.1. The code for the SCROLL.MAK application.**

```
1: Option Explicit

2:

3: Sub Form_Load ()
```

```
4: ' Set
initial scroll bar values

5: hsbBar.Value = 1800 ' Circle's default width

6: vsbBar.Value = 1800 ' Bar's default height

7: End Sub

8:

9: Sub hsbBar_Change ()

10: ' As user clicks the scroll bar,

11: ' the width of the circle adjusts

12:
shpCircle.Width = hsbBar.Value

13: End Sub

14:

15: Sub vsbBar_Change ()

16: ' As user clicks the scroll bar,

17: ' the height of the bar adjusts

18: shpBar.Height = vsbBar.Value

19: End Sub
```

***Analysis:*** Here are the vital horizontal scroll bar properties that were set during the design of the form:

Min: 50

Max: 2100

SmallChange: 50

LargeChange: 100

The shape control that contains the circle (named shpCircle) has its Width property set to 2100, so the largest that the circle could appear within the control was 2100 twips. Hence the use of 2100 for the Max property.

Here are the vital vertical scroll bar properties that were set during the design of the form:

Min: 50

Max: 2300

SmallChange: 50

LargeChange: 100

The shape control that contains the bar (named shpBar) has its Height property set to 2300, so the tallest that the bar could appear within the control was 2300 twips. Hence the use of 2300 for the Max property.

The two Min properties of 50 keep both the circle and bar from shrinking entirely when the user minimizes either control.

Lines 5 and 6 set the initial values for the circle's width and the bar's height to 1800, so both shapes are fairly large to begin with. Actually, the lines set the initial runtime values for both the scroll bars, which, in turn, generates the Change() event procedures that follow in the code.

Line 12 ensures that the circle's width increases or decreases, depending on the value of the horizontal scroll bar's Value property. Line 18 ensures that the bar's height increases or decreases, depending on the value of the vertical scroll bar's Value property.

## Prepare for the Grid Control

***Concept:*** Before you can use the grid control, you must add the grid control to your AUTOLOAD.MAK's Toolbox window.

Lesson 2 removed the grid control from the Toolbox window. The grid control is a special custom control that wasn't part of Visual Basic's original toolbox. All custom controls reside in files on your disk with the filename extension of VBX. All applications that use the grid control must list the GRID.VBX file in the Project window.

When you don't need the grid control, your applications will load more quickly and consume less disk space if you remove the GRID.VBX file from the application's Project window. Until now, the lessons in this book didn't need the grid control; therefore, Lesson 2 walked you through the steps to remove the grid control's customer GRID.VBX file from the AUTOLOAD.MAK's Project window.

If you want to work with the grid control, you must add the GRID.VBX file to any application that requires the grid control. You can decide now whether you want to add GRID.VBX to AUTOLOAD.MAK so that all subsequent projects will contain the grid control or just add the GRID.VBX file to individual projects that use the grid control.

The following steps explain how to add the grid control to AUTOLOAD.MAK. (If you want to remove the control after you complete this unit, you can do so by removing the control as explained in the second unit of Lesson 2.) To add the grid control to AUTOLOAD.MAK:

1. Load the AUTOLOAD.MAK project.
2. Select File Add File. Visual Basic displays the Add File dialog box.
3. Select the file named GRID.VBX. The file should reside in your VBPRIMER directory.
4. Press Enter or click the OK command button to close the dialog box. Visual Basic updates the Toolbox window so that the window contains the custom grid control as shown in Figure 23.3.
5. Save the AUTOLOAD.MAK project.

**Figure 23.2. The location of the grid control on the toolbox.**

---

**Note:** All new projects that you now create will contain the grid control in the Toolbox window and the GRID.VBX listing in the Project window.

---

***Review:*** Before learning how to use the grid control, you must load the GRID.VBX custom control file into your project's

Project window. By loading the file into AUTOLOAD.MAK's project, all subsequent projects will contain the grid control until you remove the custom control file.

## Using the Grid Control

***Concept:*** The grid control produces a table of rows and columns in which you can display text, numeric values, and even graphics.

When you must display several values at once, the grid control is one of the handiest controls to use. Although labels are great for messages and individual data items, and scrolling list boxes are fine for lists of values from which the user can select, the grid gives your application a two-dimensional table display of data.

Suppose that you just started a lawn fertilization company, and you begin with an initial route of eight customers. Each customer requires five annual fertilizations. That's a total of 40 applications that you need to track for the upcoming year. Each yard size differs and each application requires different amounts and kinds of fertilization mixtures.

Thinking about the eight customer's five applications, you realize that a table with eight rows and five columns would be the perfect place for placing pricing data in a computer application. After you build a Visual Basic application that contains the values, you could easily modify the table or store the values in a disk file. As you add additional customers, you could easily expand the size of the grid through the program's code.

---

*Definition:* A *cell* is one row and column intersection of a grid.

---

Before looking at an actual demonstration of a grid that tracks this lawn care business, take a few moments to study Figure 23.3. The figure shows the layout of cells on a grid that might appear on a form. The grid's size is determined by the grid's properties, which you can set or adjust using the mouse when you place the grid control on the form.

**Figure 23.3. The grid control produces a table of rows and columns.**

**Tip:** The grid doesn't have be large enough to display all of its data. If the grid's size isn't large enough to hold all the grid's cells, Visual Basic adds scroll bars so that the user can scroll through the grid.

The figure's shaded row and column is known as a *fixed row* and *fixed column*. The grid's property value settings determine whether you want to set aside a fixed row or column. When you request one or more fixed rows or one or more fixed columns (or a combination of both), Visual Basic keeps those rows and columns from scrolling when the user scrolls through the grid's values.

As the user views values inside the grid, the user can scroll through the grid, selecting (with the mouse or keyboard) one or more cells in the grid. Through programming, you can update or copy selected cell values to other grid controls, to variables, or to data files.

Table 23.2 contains the property values available for a grid control. Although you've seen several of the properties for other controls, the grid does contain properties unique to the grid control that determine the dimensions of the grid as well as the number of fixed rows and columns.

**Table 23.2. The grid control's properties.**

| *Property* | *Description* |
|---|---|
| About... | Clicking this About property opens a description dialog box that displays information about the grid control. Most custom controls come with these About dialog boxes that detail copyright information about the custom control. The About property is available only during program design time and does nothing at runtime. |

| | |
|---|---|
| BackColor | Specifies the grid control's background color, chosen as a hexadecimal color code or from the color palette. The BackColor describes the nonfixed row and column cell colors. All fixed row and column cells are gray. |
| BorderStyle | Specifies that a border appears around the grid if set to 1-Fixed Single (the default). No border appears if set to 0-None. |
| Cols | Holds the number of columns in the grid. |
| DragIcon | Contains the icon that appears when the user drags the grid around on the form. (You'll only rarely allow the user to move a grid, so the Drag... property settings aren't usually relevant.) |
| DragMode | Holds either 1 for manual mouse dragging requirements (the user can press and hold the mouse button while dragging the control) or 0 (the default) for automatic mouse dragging, meaning that the user can't drag the grid but that you, through code, can initiate the dragging if needed. |
| Enabled | Determines whether the grid can respond to events. If set to True (the default), the grid can respond to events. Otherwise, Visual Basic halts event processing for that particular control. |
| FillStyle | If set to 0-Single (the default), a value is to be assigned only to a single selected cell, and if 1-Repeat, a value is to be assigned to a range of selected cells. |
| FixedCols | Holds the number of fixed columns. The FixedCols value must be at least two fewer than the Cols value. |
| FixedRows | Holds the number of fixed rows. The FixedRows value must be at least two fewer than the Rows value. |
| FontBold | Holds True (the default) if the grid values are to display in boldfaced characters; False otherwise. |
| FontItalic | Holds True (the default) if the grid values are to display in italicized characters; False otherwise. |
| FontName | Contains the name of the grid values' font styles. Typically, you'll use the name of a Windows TrueType font. |
| FontSize | Holds the size, in points, of the font used for the grid values. |
| FontStrikethru | Holds True (the default) if the grid values are to display in strikethru letters (characters with dashes through them); False otherwise. |
| FontUnderline | Holds True (the default) if the grid values are to display in underlined letters; False otherwise. |
| ForeColor | Specifies the color of the characters in the grid values. |
| GridLines | Holds True (the default) if the grid is to control separating row and column lines; False otherwise. |
| Height | Holds the height, in twips, of the grid. |
| HelpContextID | If you add advanced, context-sensitive help to your application, the HelpContextID provides the identifying number for the help text. |
| Highlight | Holds True (the default) or False to determine whether selected cell or cells are to appear highlighted. |
| Index | If the grid is part of a control array, the Index property provides the numeric subscript for each particular grid control. |
| Left | Contains the number of twips from the left edge of the Form window to the left edge of the grid. |
| Name | Contains the name of the control. By default, Visual Basic generates the names Grid1, Grid2, and so on as you add subsequent grids to the form. |
| Rows | Holds the number of rows in the grid. |

| | |
|---|---|
| ScrollBars | Holds 0-None, 1-Horizontal, 2-Vertical, or 3-Both (the default) to describe the scroll bars that appear in the grid if the grid requires more row and column space than the grid size allows. |
| TabIndex | Determines that the focus tab order begins at 0 and increments every time you add a new control. You can change the focus order by changing the controls' TabIndex to other values. No two controls on the same form can have the same TabIndex value. |
| TabStop | If True, determines whether the user can press Tab to move the focus to this grid. If False, the grid can't receive the focus. |
| Tag | Unused by Visual Basic. This is for the programmer's use for an identifying comment applied to the grid. |
| Top | Holds the number of twips from the top edge of a grid to the top of the form. |
| Visible | Holds True or False, indicating whether the user can see (and, therefore, use) the grid. |
| Width | Holds the number of twips wide that the grid consumes. |

---

***Warning:*** The grid control enables your user to view and scroll through values but doesn't allow the user to enter new values into the grid. Your program can update the grid, but the user can't.

---

In addition to the property values that you can set at program design time (those listed in Table 23.2), there are additional property values that your program can add and modify at runtime. Table 23.3 lists the runtime properties available to your code.

**Table 23.3. The grid control's runtime properties.**

| *Property* | *Description* |
|---|---|
| Col | Contains the column number, counting from 0, of the currently selected cell. |
| ColAlignment | Contains 0 (the default) for left justification of the cell values, 1 for right justification, and 2 for center alignment of the cell values. This property applies only to the cells in nonfixed columns. |
| ColWidth | Holds the width, in twips, of an individual column. |
| FixedAlignment | Contains 0 (the default) for leftjustification of the cell values, 1 for right justification, and 2 for center alignment of the cell values. This property applies only to the cells in fixed columns. |
| HighLight | Holds True or False to indicate whether or not the user selected a cell or a range of cells. |
| Picture | At runtime, you can assign the LoadPicture() procedure (see the previous unit) to display a graphic image in a selected cell. |
| Row | Contains the row number, counting from 0, of the currently selected cell. |
| RowHeight | Holds the height, in twips, of an individual row. |
| SelEndCol | Holds the rightmost column of a selected range. |
| SelEndRow | Holds the bottom row of a selected range. |
| SelStartCol | Holds the leftmost column of a selected range. |
| SelEndCol | Holds the top row of a selected range. |
| Text | Holds the value of any given cell. |

The two most frequently used runtime property values are the Row and Col properties. These two properties determine which cell you're currently formatting and which cell you want to assign text to. Before assigning a grid's cell a value, you must set the Row and Col values to the row and column that intersect the cell to assign to.

---

**Note:** Many of the property values in Table 23.3 make sense only when the user runs the program because, until then, no cells are selected. The project application at the end of this lesson demonstrates how to use these properties to initialize a grid and respond to the user's selected range of cells.

---

***Review:*** The grid control is useful for displaying tables of data values for the user. The grid control is a custom control located inside the GRID.VBX file that you must add to any application that uses the grid control.

## Monitoring the Mouse Cursor

***Concept:*** Your program can respond to mouse movements and the user's mouse clicks through event procedures. The mouse object supports property values and methods that you can use to monitor the mouse. One important property that you may want to control during the user's mouse movement is the MousePointer property, which determines how the mouse looks when the user moves the mouse over a control.

If you need to, you can monitor the user's mouse movements and clicks. As you've seen throughout this book, Visual Basic takes care of monitoring most of the important mouse functions. For example, you already know that if the user clicks a mouse over a command button, Visual Basic ensures that the command button's Click event procedure automatically executes without your having to worry about looking for the mouse click.

One of the most common ways that a program works with the mouse is to change the appearance of the mouse cursor. In Windows, the term *cursor* is technically used solely for the mouse cursor, whereas the term *caret* is used for the text cursor. Despite Microsoft's original design and the technical manuals that promote these "accurate" names, most users and programmers refer to the mouse cursor simply as the *mouse cursor* and the text cursor as the *text cursor*. This chapter will continue to use the vernacular.

Table 23.4 contains a list of every mouse cursor shape that you can display. Most controls contain the MousePointer property. For example, command buttons contain a MousePointer property. The value of each control's MousePointer property, described in Table 23.4, determines what shape the mouse cursor takes on when the user moves the mouse over that control.

**Table 23.4. The thirteen mouse cursor values.**

| *Value* | *Description* |
|---|---|
| 0-Default | The cursor assumes the control's default mouse cursor shape. Each control has its own default mouse cursor shape. Most controls use the common mouse cursor arrow for the default shape. |
| 1-Arrow | The typical arrow mouse pointer. (Generally, this is the same shape as most controls' default mouse cursor shape.) |
| 2-Cross | A crosshair pointer. |
| 3-I-Beam | The vertical mouse cursor most often used as a text cursor. |
| 4-Icon | A small black square within another square. |
| 5-Size | The sizing cursor that looks like a plus sign with arrows pointing in the four directions. |
| 6-Size NE SW | A diagonal arrow pointing northeast and southwest. |
| 7-Size N S | A vertical arrow pointing north and south. |
| 8-NW SE | A diagonal arrow pointing northwest and southeast. |
| 9-W E | A vertical arrow pointing west and east. |
| 10-Up Arrow | An arrow pointing straight up. |
| 11-Hourglass | The hourglass shape. |
| 12-No Drop | The familiar roadsign "No" circle with a slash through it. |

Depending on the application, you may need to keep the user from clicking on a control. For example, you may want to ignore

all clicks of a command button that prints a report of daily activities until after a 5:00 p.m. closing time. Your application could respond to the command button's click after 5:00 p.m. and return without responding if the time is before 5:00 p.m.

During the time that the command button is to be ignored, you could set the command button's MousePointer shape to the 12-No Drop mouse cursor. Therefore, the user's mouse cursor changes to the no drop shape whenever the user moves the mouse cursor over that particular command button. The mouse cursor could remain the standard arrow pointer for the other controls. After 5:00 p.m., inside the click procedure, you could respond to the command button click by printing the report and also set the MousePointer property to 1-Default (or 1-Arrow, which is the same shape for command buttons because the default mouse cursor is the arrow for command button controls).

Table 23.5 lists the CONSTANT.TXT file's named constants that you can use to set the MousePointer property values.

**Table 23.5. The CONSTANT.TXT's named MousePointer property values.**

| *Property* | *Description* |
|---|---|
| DEFAULT | The control's default mouse cursor shape. |
| ARROW | The typical mouse cursor arrow. |
| CROSSHAIR | A crosshair. |
| IBEAM | An I-Beam (the typical text cursor) |
| ICON_POINTER | An icon square within a square |
| SIZE_POINTER | The sizing cursor that looks like a plus sign with arrows pointing in the four directions. |
| SIZE_NE_SW | A diagonal arrow pointing northeast and southwest. |
| SIZE_N_S | A vertical arrow pointing north and south. |
| SIZE_NW_SE | A diagonal arrow pointing northwest and southeast. |
| SIZE_W_E | A vertical arrow pointing west and east. |
| UP_ARROW | The straight up arrow. |
| HOURGLASS | The hourglass waiting shape. |
| NO_DROP | The "No" circle with a line through it. |

***Stop and Type:*** Listing 23.2 contains the code found in this book's MOUSECH.MAK application. The program demonstrates how the mouse cursor can change depending on the value of an option button. The program produces the before 5:00 p.m. and after 5:00 p.m. results described earlier in this section.

***Review:*** Most controls support the MousePointer property that determines the shape that the mouse takes on when the user moves the mouse over another control.

**Listing 23.2. Code that changes the shape of the mouse depending on the value of option buttons.**

```
1: Sub optAfter_Click ()

2: ' After 5:00 pm so fix mouse over command

3:
cmdReport.MousePointer = ARROW

4: End Sub

5:
```

```
6: Sub optBefore_Click ()

7: ' Before 5:00 pm so negate mouse over command

8: cmdReport.MousePointer = NO_DROP

9: lblReport.Visible = False

10: End Sub

11:

12: Sub cmdExit_Click ()

13: End

14: End Sub


15:

16: Sub cmdReport_Click ()

17: ' Beep if it's before 5:00

18: If optBefore.Value = True Then

19: Beep

20: lblReport.Visible = False

21: Else

22: lblReport.Visible = True

23: End If

24: End Sub
```

***Output:*** Figure 23.4 shows what happens to the shape of the mouse cursor when the user moves the mouse cursor over the command button before 5:00 p.m.

**Figure 23.4. The mouse cursor tells the user that the command button won't respond.**

***Analysis:*** When the program begins, the first option button named optBefore is set to True. Therefore, the application assumes

that the time is before 5:00 p.m. and the command button's MousePointer value is set to NO_DROP as, shown in Figure 23.4.

---

**Note:** The program could use the Time$() function to check the actual time to see whether the command button can be active or not. However, the option buttons offer you an easier method for practicing with the two mouse cursors; you don't have to wait until 5:00 p.m. to see the difference.

---

As soon as the user clicks the After 5:00 p.m. option button named optAfter, the optAfter_Click() event procedure changes the command button's MousePointer property to ARROW in line 3.

When the user clicks the Report command button, the command button's cmdReport_Click() event procedure executes (line 16). If the Before 5:00 pm option button is set, line 19 beeps. If the After 5:00 pm option button is set, a small label appears beneath the command button to inform you that the report can now be printed. (The label, named lblReport, is set to False when the program first begins.)

## Capturing Mouse Clicks and Movements

***Concept:*** The mouse supports three events that return movement and clicking information to the user through the form of event procedures. Unlike most event procedures, the mouse event procedures use arguments that return information such as the location of the mouse and the button that was clicked.

Table 23.6 lists the events most often supported by applications that deal with the mouse. These events return events for which you can write event procedures to respond to the user's mouse commands.

**Table 23.6. The events most often associated with the mouse.**

| *Event* | *Description* |
|---|---|
| DblClick | Generated when the user double-clicks a mouse button |
| MouseDown | Generated when the user presses a mouse button |
| MouseMove | Generated when the user moves the mouse |
| MouseUp | Generated when the user lets up on a mouse button |

The MouseDown, MouseMove, and MouseUp event procedures open with argument lists when you select them inside the Code window. Suppose that you wanted to respond to user clicks of the mouse when the user clicked the mouse over a form named frmApp. Here is the three event procedures' wrapper code that Visual Basic would open for you when you selected MouseDown, MouseMove, or MouseUp from the Code window's dropdown Proc listbox:

```
Sub frmApp_MouseDown (Button As Integer, Shift As
Integer, X As Single, Y As Single)

End Sub

Sub frmApp_MouseMove (Button As Integer, Shift As Integer, X As Single, Y As Single)

End Sub

Sub frmApp_MouseUp (Button As Integer, Shift As Integer, X As Single, Y As Single)
```

```
End Sub
```

The argument lists for these procedures are so long that you may have to horizontally scroll the Code window to see the entire argument list.

Table 23.7 lists the descriptions for these four argument values. The values returned in the arguments describe something about the user's mouse press or mouse release. Your code can check the argument values to determine what needs to be known. For example, if you want to know the location of the mouse when the user clicked the mouse on the form, the X and Y arguments describe the number of twips from the left and top edge of the form that the user clicked.

**Table 23.7. The MouseDown() and MouseUp() arguments.**

| *Argument* | *Description* |
|---|---|
| Button | Holds a number that represents the mouse button pressed. The value is 1 for the left button press, 2 for the right button press, and 4 for both or for the center button on a three-button mouse. |
| Shift | Describes the shifting key (if any) pressed at the same time that the user clicked or moved the mouse button. Shift holds 1 if the user pressed Shift during the mouse click, 2 for the Ctrl key, and 4 for the Alt key. If Shift contains a number other than 1, 2, or 4, that number is the sum of two or more key presses just described. For example, if the user presses the Alt keys at the same time that the user pressed the mouse button, the value of Shift will be 5. |
| X | Contains the horizontal twip form measurement where the user clicked or moved the mouse button. |
| Y | Contains the vertical twip form measurement where the user clicked or moved the mouse button. |

If the user double-clicks the mouse, Visual Basic generates both a MouseUp *and* a DblClick event. The MouseUp event procedure returns location and button information about the double-click (see Table 23.6), and the DblClick event should contain the code that you want executed upon the double-clicking of the mouse.

The MouseUp event occurs every time that the user lets up on a mouse button, whether the user lets up from a single click or from a double-click. Therefore, if you want to respond to a single click's MouseUp event but not to a double-click's MouseUp event, you'll have to set a module variable inside the DblClick event procedure that you can test to see which kind of release, a click or double-click, generated the MouseUp event.

---

**Note:** Depending on your computer, Visual Basic might generate a mouse movement event procedure every time that the user moves the mouse 10 to 15 twips across the form. Visual Basic doesn't generate a mouse movement event for *every* twip movement; most computers couldn't keep up with the events that would execute that fast.

---

***Stop and Type:*** Listing 23.3 contains a sample MouseUp() event procedure that demonstrates how the code can use the arguments to the procedure to learn more about the mouse movement. The code prints mouse information. You could write a similar set of mouse event procedures for the other mouse events and get a printed log of the mouse actions as you moved, clicked, and double-clicked the mouse.

***Review:*** The mouse movement and clicking event procedures need additional information when a mouse event occurs. Through the argument list to the mouse event procedures, you can determine which button the user pressed, which key the user was pressing at the time of the click or mouse movement, and exactly where on the form the mouse was when the user generated the event.

**Listing 23.3. Code that checks the mouse after a button release.**

```
1: Sub frmApp_MouseUp (Button As Integer, Shift As Integer, X As Single, Y As Single)

2: ' Display text on the printer
that describes the

3: ' mouse button press. The semicolon at the end of

4: ' the Printer.Print statements forces Visual

5: ' to keep the printer cursor on the same Print

6: ' line.

7:

8: ' Tell the user about the button pressed

9: Printer.Print
"Up: The button you released was the ";

10: Select Case Button

11: Case 1: Printer.Print "Left ";

12: Case 2: Printer.Print "Right ";

13: Case 4: Printer.Print "Middle ";

14: End Select

15: Printer.Print
"button"

16:

17: Printer.Print "The mouse was at X position: ";

18: Printer.Print X;

19: Printer.Print "and Y position: ";

20: Printer.Print Y

21: ' Print a blank line to separate for subsequent output
```

```
22: Printer.Print


23:


24: End Sub
```

***Analysis:*** Line 1 collects the mouse button's release arguments. The argument tell the event procedure which button the user just released, which key, if any, was pressed at the time, and the exact twip coordinates of the mouse when the user released the mouse.

The rest of the event procedure proceeds to print the mouse information gathered from the arguments. Keep in mind that this event procedure will execute every time the user releases either mouse button, so the printer's output will continue as the user continues clicking the mouse.

## Homework

### General Knowledge

1. How many different scroll bar controls are there?
2. How do the scroll bars eliminate the typing of specific entries?
3. What do the LargeChange and SmallChange properties indicate?
4. How does a user trigger a LargeChange change of a scroll bar's value?
5. What do the Min and Max properties indicate?
6. What are the preferred naming prefixes for scroll bar controls?
7. True or false: You can initialize a scroll bar using a variable defined as an Integer data type.
8. What must you do to add the grid control to the Toolbox window?
9. What is the name of the custom control file that holds the grid control?
10. True or false: The user can read, change, and enter new values into the grid.
11. What is a *cell*?
12. True or false: When you adjust the grid's physical size on the form, the grid's size must be large enough to display all of the grid's rows and columns.
13. When can scroll bars appear on a grid?
14. What two property values, available only at runtime, specify an individual cell that you want to change or initialize with code?
15. What are the four property values that define selected cells?
16. How many different shapes can the mouse appear as over a control?
17. Which property describes the mouse cursor's appearance?
18. Technically, what is the text cursor called?
19. Technically, what is the mouse cursor called?
20. When would the 0-Default and 1-Arrow MousePointer property values differ?
21. Which mouse cursor might be useful for reminding the user to wait a few seconds until a calculation completes?
22. How do the mouse movement events generate extra information that describe the buttons and location of the mouse?
23. True or false: When the user double-clicks a mouse button, two events actually take place.

## Find the Bug

1. Kelly wants to use a grid control to hold a 5-by-5 table of weight readings that he needs for a chemistry experiment. After Kelly initializes the table, through code, and then presses a command button, he wants the code to fix the rows and columns for the entire table so that the user can't select additional cells. Sadly, Visual Basic imposes a limit that doesn't allow Kelly the freedom to fix all rows and columns. What is the maximum number of rows and columns that can be fixed in a table?

## Write Code That...

1. Suppose that you were writing a program that needed the user to enter a temperature value from a range of 32 to 212 degrees. You want the user to enter the temperature reading using the relative position on a scroll bar. When the user clicks one of the scroll bar arrows, the scroll bar should adjust by 3 degrees. When the user clicks in the shaft on either side of the scroll bar, the scroll bar should adjust by 8 degrees. What would be appropriate Min, Max, SmallChange, and LargeChange properties?

## Extra Credit

Write a program that contains one command button labeled Change Mouse that displays a different mouse cursor every time the user clicks the button. Use a module variable to keep track of the current mouse pointer's value.

# Visual Basic in 12 Easy Lessons



# Lesson 12, Unit 24

## Debugging For Perfection

### What You'll Learn

- The debugging routine
- The debugger
- Entering break mode
- Using the Immediate window

This unit finishes the book by taking a break from the usual teaching of controls and code. As you write more and more programs, you'll have more and more success at improving your Visual Basic skills. The problem is that your program errors will increase as well. No matter how careful, almost every programmer introduces errors along with working code. Those errors, known as *bugs*, are sometimes extremely difficult to find.

Visual Basic contains several debugging tools that help you hunt, find, and correct the bugs that will appear. Visual Basic actually makes finding bugs fun. Well, perhaps you won't use the term *fun*, but the debugging tools that Visual Basic provides certainly makes locating those errors easier than finding bugs from programming languages that preceded Visual Basic.

# The Debugging Routine

***Concept:*** Several kinds of errors can occur. Some errors are easy to find and others are not so easy to find. Visual Basic's debugging tools help you zero in on some of the more difficult kinds of program errors that can creep into your code.

---

*Definition:* A *debugger* is the integrated tool that helps you find program bugs.

---

As described in Unit 1, there are two general categories of errors that you'll often run across as you use Visual Basic. The debugger can help you locate and fix these errors. You will find both *syntax errors* and *logic errors*.

A syntax error is the easiest kind of error to find. A syntax error often occurs because you misspell a command, function name, or method. A syntax error can also appear because you rearrange the grammar of a loop such as a For loop with a Step option that appears before the loop's To keyword.

As you write your program, Visual Basic will spot syntax errors for you. If you set the Options Environment Syntax Checking option to Yes, Visual Basic actually checks for syntax errors every time you press Enter at the end of any code line that contains an error. Figure 24.1 shows how Visual Basic displays a message box that warns the user of the error.

**Figure 24.1. The programmer just typed a syntax error and needs to fix the problem.**

When you run across a syntax error message box, press Enter to get rid of the message box and fix the problem. Usually, you'll immediately see the misspelled word and correct the error. If you can't locate the error right away, or if you want to come back to the problem later, Visual Basic will allow you to continue writing the rest of the program without displaying the error message again unless you change the same line and fail to correct the problem.

---

**Note:** As you can see from Figure 24.1, Visual Basic doesn't always call the error a *syntax error* but you'll know from the context of the error message that the statement's syntax (grammar or spelling) is wrong.

---

If you set the Options Environment Syntax Checking option to No, Visual Basic will ignore all syntax errors until you run the program. At runtime, Visual Basic will display the syntax error message boxes as Visual Basic runs across them.

The second kind of error you'll run across is a logic error. Logic errors are more difficult to find. Logic errors aren't errors of grammar or spelling, but are errors in action.

Logic errors appear when your program seems to understand everything you typed but produces incorrect results. For example, suppose that you wrote a program that printed paychecks and, every so often, the

program made a calculation error and doubled the amount of withholding for all employees who had been with the company more than ten years. If you were running the program and printing the checks, you would probably not notice the errors. You *would* notice the errors, however, when the employees began pounding on your office door.

Sometimes, logic errors don't appear until people begin using the program. You'll get the syntax errors out of a program right away because Visual Basic helps you find them with message boxes. People are the hunters and gatherers of logic errors. When you realize that one of your programs has a problem with a calculation, loop, or a procedure, you must return to the Code window and find the problem. In large programs, those logic errors can be difficult to trace.

Logic errors can occur because you forgot to initialize a variable properly, you didn't loop through enough table or list control values, or you reversed data types when you defined variables. Although logic errors are often hard to locate, Visual Basic's integrated debugger, described in the rest of this unit, helps you find logic errors.

***Review:*** There are two general categories of errors: syntax errors and logic errors. Visual Basic finds all syntax errors in your program. The logic errors require work on your part. The debugger, described next, helps you hunt down offending logic errors.

## The Debugger

***Concept:*** The Debug menu produces a list of all of Visual Basic's debugging commands. Using the debugging commands, you'll be able to stop a program before the execution of any line of code, look at variable and control values, and single step through the code starting at any line.

Figure 24.2 shows the Debug menu. Visual Basic's debugging tools are extremely powerful and allow you to execute programs using one of several methods that enable you to analyze various parts of the code.

**Figure 24.2. The debugging tools are available from the Debug command on the menu bar.**

---

*Definition:* Visual Basic enters the *break mode* when you halt a program during execution.

---

These menu options are all available during the break mode. These are the three modes that a Visual Basic program can be in:

- Design mode
- Runtime mode
- Break mode

Visual Basic tells you which mode is current by displaying the words design, run, or break in Visual Basic's title bar at the top of your Visual Basic screen. When you develop the program, the program is in the design mode; when you or the user runs a program, the program is in runtime mode; and when you

halt a program to use the debugger to analyze program problems, the program enters the break mode.

This unit concerns itself with the break mode. While in break mode, your program retains all variable and control values. Therefore, you can halt the program at any time and look at data values from any line of the code. By comparing the values with what you expect the values to contain, you can find where problems are taking place.

***Review:*** Of Visual Basic's three modes, you'll be using the break mode to debug programs. When a program enters the break mode, the program halts at any line in the program's execution but retains all variable and control values assigned to that point.

## Entering Break Mode

***Concept:*** There are several ways to enter the break mode. The most common way is to set a breakpoint. You also can halt a program during the program's execution by intercepting the program's execution with Ctrl+Break or by selecting Break from the Run menu at runtime.

You'll always enter the break mode from the runtime mode. Only after you begin a program's execution will the break mode be available, because only at runtime are the variables and controls initialized with values. Here are the ways that you can move from the runtime mode to the break mode:

- Press Ctrl+Break during the program's execution at the place where you want to enter the break mode. Stopping on one particular line of code is virtually impossible when using Ctrl+Break.
- Select Run Break from the menu bar.
- Click the Break toolbar button (the button with the hand icon).
- In design mode or during a break mode, set a *breakpoint* on a particular line at which you want the execution to halt. By setting a breakpoint, you can specify the exact line of code where Visual Basic is to enter the break mode.
- The debug menu's Add Watch dialog box enables you to specify a *break expression* that Visual Basic monitors and uses to halt the program's execution when the expression becomes true.
- If a runtime error occurs, such as a division by zero that is an undefined math operation, Visual Basic enters the break mode on the offending line. Sometimes, this kind of error is called a *runtime* error and fits within the logic error category.

The most accurate and common way to enter the break mode is by setting a breakpoint. To set a breakpoint, find the line where you want execution to halt at a breakpoint, and set a breakpoint at that particular line of code. The following steps walk you through setting a breakpoint:

1. Load the CHECK2.MAK project that comes with this book.
2. Press F7 to open the Code window. Visual Basic opens the code to the Form_Load() event procedure.
3. Find the following line of code in Form_Load():
   lblPoem = lblPoem & "If you ever hear differently,"
4. Move the mouse cursor to the line and click the mouse button. The text cursor appears at the mouse click's location.

5. Select Debug Toggle Breakpoint to set a breakpoint. You'll see from the menu command that F9 is the access key for this command. Also, clicking the toolbar's hand icon would also place a breakpoint on this line of code. Figure 24.3 shows how your code window should appear. Visual Basic changes the color of the line to let you know that a breakpoint will take place on that line during the program's execution.
   You can turn off a breakpoint by selecting the Debug Toggle Breakpoint (or pressing F9) once again. You can set as many breakpoints as you need throughout a program. Leave the breakpoint in place for now. By setting the breakpoint, you're requesting that Visual Basic halt the program and enter break mode when execution reaches this line of code. Close the Code window and run the program by pressing F5.

**Figure 24.3. Visual Basic highlights all breakpoints.**

As soon as you press F5 to run the program, Visual Basic executes the program. At least, Visual Basic executes the code down to the first breakpoint. As soon as Visual Basic reaches the breakpoint line, Visual Basic enters the break mode *before* executing the breakpoint line. You can verify the break mode by reading Visual Basic's title bar and seeing the word **(break)** at the top of the screen.

The CHECK2.MAK application is the program that concatenated and assigned a poem into a label's caption. The Form_Load() procedure builds the concatenated poem. The breakpoint that you set occurs in the middle of the poem-building code. Only two values are initialized at the breakpoint. The string variable Newline contains a carriage return and line feed sequence, and the control named lblPoem contains the poem's text.

At the breakpoint, the lblPoem label contains the first two lines of the poem. Remember that all controls have default properties, and the Caption property is the default property for the label. Therefore, in building the poem, the code doesn't specify that the poem's text is to enter the Caption property of the label. As you'll see, whenever you specify only the label's name, Visual Basic assumes that you want to assign or display the Caption property.

Follow these steps to see what kinds of things you can do at a breakpoint:

1. By dragging the mouse, highlight either mention of the lblPoem control on the breakpoint's line.
2. Click the toolbar button with the eyeglass icon. The eyeglass toolbar button produces the Instant Watch dialog box (which is also available from the Debug menu). Visual Basic displays the first line stored in the lblPoem's Caption property, as shown in Figure 24.4.
   At first, you may be disappointed that the Instant Watch dialog box doesn't show both lines of the label's Caption property contents. Keep in mind, though, that a label might contain a caption that is several thousand characters long. Visual Basic is able to show you only the characters up to the carriage return and line feed characters at the end of the caption's first line.
3. Click the Instant Watch dialog box's Cancel command button to close the dialog box. Visual Basic returns to the Code window's breakpoint.
4. Usually, the programmer will single step through a few lines of code after reaching a breakpoint. To step through the code one line at a time, you can choose Debug Single Step, press F8, or click the single footstep icon on the toolbar. As you single step though the code, Visual Basic highlights the next line of execution. At any point during the single-step process, you can examine variables and controls with the Instant Watch dialog box.

**Figure 24.4. Displaying the label's value.**

Eventually, the Form_Load() procedure ends and the form appears on the screen. If you were to click one of the option buttons, Visual Basic would execute that option button's event procedure and simultaneously open the Code window once again so that you could continue the single step process.

Suppose that a variable contains an incorrect value but you're not exactly sure where the error is occurring. You could set a breakpoint at every line of code that changes that variable. When you run the program, you'll look at the contents of that variable before and after each breakpoint's line of execution. If the first breakpoint seems to initialize the variable properly, you don't have to single step through the code until the next breakpoint is reached. Instead of single stepping, you can select Run Continue or press F5 to return the execution to its normal runtime (and real time) mode. When Visual Basic reaches the next breakpoint, the code halts at that next breakpoint and you can continue to examine the variable.

Either from the Debug menu or from the Instant Watch dialog box, you can request the Add Watch dialog box. The Add Watch dialog box is shown in Figure 24.5. The Add Watch dialog box allows you to type any expression in the Expression text box.

**Figure 24.5. Set up expressions that Visual Basic watches for in the Add Watch dialog box.**

The Add Watch dialog box is useful for data errors that you can't seem to locate in the code. Rather than break at every line that uses of the data value, you can add a variable or expression to the Add Watch dialog box, and Visual Basic will halt execution when that variable changes, when the expression that you enter in the Add Watch dialog box becomes true, or when Visual Basic changes the value of the expression (the Watch Type frame at the bottom of the Add Watch dialog box determines how you want to set up the watch).

For example, suppose that a variable is to maintain a count of customers, but somewhere in your code a negative value appears in the variable. If you added a watch expression such as CustCnt < 0 to the Add Watch dialog box's Expression prompt, and clicked the Break when Expression is True option button, you could then run the program and Visual Basic would enter the break mode at the exact line that caused the variable to become negative.

The toolbar buttons with the double footstep icon works almost like the single-step icon except that Visual Basic executes the breakpoint's current procedure in a single-step mode but executes all procedures *called* by the single-step procedure in their entirety.

Terminate your current debug session by selecting Run End. Visual Basic returns to the design mode.

***Review:*** The breakpoints and watch dialog boxes that you can request while debugging your code give you tremendous power in analyzing variables and watching for specific results. You can look at the contents of variables and controls to make sure that data is being initialized the way you expect. Also, the Add Watch dialog box enables you to set up expressions that Visual Basic watches for during the program's execution. If the values of those expressions ever become true or change, Visual Basic will halt the code at that line and allow you to analyze values using the Instant Watch dialog box.

# Using the Immediate Window

***Concept:*** The Immediate window enables you to look at and change variables and controls during the execution of a program.

At any breakpoint, you can select Window Debug (Ctrl+B) to request the Debug window. The Debug window is a special window in which you can directly type Visual Basic commands, and view and change variables and control values during a program's execution.

For printing, you can apply the Print method (see Unit 21) to view variables and controls. When you use Print in the Debug window, Visual Basic sends the output to the Debug window and not to the Printer object, as you saw in Unit 21. For example, suppose that you set a breakpoint during the poem's concatenation, as described in the previous section, and you pressed Ctrl+B to open the Debug window. The Debug window recognizes simple commands and methods such as Print and assignment statements.

Figure 24.6 shows what happens if you print the value of the lblPoem. Unlike the Instant Watch dialog box, the Debug window displays the entire multiple line value of the label. You can resize and move the Debug window. Although they must use the Print command instead of simply clicking a variable or control, many users prefer to display values from the Debug window instead of from the Instant Watch dialog box because the Debug window displays the entire value and contains a vertical scroll bar so that they can scroll through the values printed in the window.

**Figure 24.6. In the Debug window, you can print values of variables and controls.**

---

Use the Debug Window Inside Your Code: The Debug window's scrolling and resizing features are sometimes so handy that some Visual Basic programmers prefer to send messages to the Debug window at runtime rather than use the Instant Watch dialog box. For example, if you want to see the value of certain arguments when called procedures execute, you can add the Print commands at the top of those procedures that send the argument values to the Debug window automatically as the program executes. Once you get the bugs out of the program, you can remove the Print commands so that the Debug window stays closed.
To print to the Debug window, preface the Print method with the Debug object. The following command, executed anywhere from an application's Code window, prints the values of two variables with appropriate titles in the Debug window:
Debug.Print "Age:"; ageVal, "Weight:"; weightVal
All the Print method's options, including semicolons, commas, Tab(), and the Spc() functions, work inside the Debug window just as they did for the Printer object described in Unit 21. Be careful to specify the Debug object before the Print method, however. If you omit Debug, Visual Basic prints the output directly on the form itself!

---

The Debug window recognizes assignments that you make to variables and controls. For example, suppose that you know that a certain variable wasn't initialized properly earlier in the execution, but you still want to finish the program's execution. If you need to, you can assign that variable a new value

directly within the Debug window using the assignment statement. When you resume the program's execution, either in single-step or in runtime mode, the variable, from that point in the program, will contain the value that you assigned to it.

***Review:*** The Debug window gives you more control over the display of variables and controls. The Debug window is resizable and provides you with a scroll bar so that you can see the entire contents of variables and controls as well as keep a running and scrollable tally of those values as you debug the program's execution. The Debug window gives you the ability to assign values to variables and controls as well. In the middle of a program's execution, you can enter the break mode and change the values of anything. When you subsequently resume the program's execution, the program will use those values that you assigned in the Debug window.

# Homework

## General Knowledge

1. What are the two major classifications of errors that a program can contain?
2. Which error would describe spelling and grammar errors?
3. What is a *debugger*?
4. How can you request that Visual Basic check for syntax errors as you type a program into Visual Basic's Code window?
5. True or false: Visual Basic displays a message box when a syntax error appears.
6. True or false: Visual Basic displays a message box when a logic error appears.
7. Which kind of error, syntax or logic, is the most difficult to find?
8. What three modes can a program be in?
9. Which mode is Visual Basic in when you write the program and place items on the form?
10. How can you tell which mode a program is currently in?
11. Which mode is perfect for displaying and changing variables?
12. Name two ways to enter the break mode.
13. What is a *breakpoint*?
14. How can you set a breakpoint?
15. How do you know that a breakpoint is set when you look through the Code window?
16. What happens when Visual Basic encounters a breakpoint during the execution of a program?
17. What is meant by *single stepping* through a program?
18. What is the difference between the toolbar button with the single footprint and the double footprint?
19. What is the difference between the Instant Watch and the Add Watch dialog boxes?
20. Which kind of watch is most useful for displaying variables at a program's breakpoint?

21. Which window enables you to view and change Visual Basic variables and controls at runtime?
22. Which command enables you to display data values in the Debug window?
23. Which command enables you to assign new values to variables and controls at runtime?

## Find the Bug

Jennifer wants to send several values to the Debug window so that she can keep a scrollable list of variable values as she runs a program. Here are some of the statements that Jenny is trying to use to print in the Debug window:

```
Print lblTitle.Caption
Print x, y, z
Print CompName
```

Even though Jennifer opens her Debug window during the program's execution, the window is blank. Instead of going to the Debug window, Jennifer's Print methods seem to be appearing on the form itself. Help Jennifer with her problem.

# Visual Basic in 12 Easy Lessons

---



---

# Project 12

## Getting Exotic

### Stop & Type

This lesson taught you how to use the scroll bars and the grid control, to manage the mouse, and to use Visual Basic's integrated debugger. The scroll bars give users a flexible control for entering and selecting from a range of data values. The grid enables the user to see tables of information in a spreadsheet-like format. When you need to monitor the user's mouse movements, in addition to the built-in monitoring that Visual Basic performs for all control clicking with the mouse, you can write code that responds to the user's mouse movements and clicks.

The debugger is an integrated development tool that helps you hunt down errors that appear in your code. The debugger allows you to single step through a program's code, monitor variables, and inquire into a program's data values at any point in the running of the program.

In this lesson, you saw the following:

- How to place and monitor scroll bar controls
- How to set up a grid for tables of data
- How to determine selected cells within a grid
- How to monitor the user's mouse movements
- How to use Visual Basic's integrated debugger to remove obscure bugs in your programs

## The Program's Description

Figure P12.1 shows the PROJEC12.MAK opening Form window. Much of this project comes from a description of a scroll bar application found in the first unit of this lesson. As Figure P12.1 shows, the user first sees a formatted table of values stored in a grid.

**Figure P12.1. Project 12's application begins with a simple form.**

---

**Warning:** You must make sure that the GRID.VBX customer control file for the grid control appears in your Visual Basic's directory. See Unit 23 for details on adding this file.

---

The application demonstrates how to use, change, and monitor the user's use of a grid control as well as how you can use scroll bars to change the behavior of other controls on the form.

## Studying the Grid

The grid represents the lawn fertilizer's application price table for the 8 customer's 5 lawn applications. As described in Unit 23, this lawn care company just began and needs to make 40 service calls that year. As the company adds more customers, the grid can easily be expanded by changing the grid's property values and adding the new values in the code. Once the number of customers grew by just a few more, the lawn service would want to incorporate the disk drive and store the current pricing table on the disk drive for easy maintenance and for use by other programs such as an invoicing program.

Try this: Click the grid's scroll bars to scroll around the pricing table looking at the 40 prices. Before worrying about the command buttons and horizontal scroll bars at the right of the screen, study listing P12.1 for a glimpse of the overall program's format.

**Listing P12.1. The Form_Load() procedure that controls the flow of the initial code.**

```
1:
Sub Form_Load ()

2: ' Defines the justification of the cells

3: ' as well as assigns cell titles to the

4: ' fixed row and columns and assigns initial

5: ' price values to the 40 cells.
```

```
6: Call InitScrolls ' Initialize scroll bars

7: Call CenterCells
' Center all cell alignments

8: Call SizeCells ' Specify width of cells

9: Call Titles ' Initialize column and row titles

10: Call FillCells ' Fill cells with values

11: End Sub
```

---

6: Break long modules into shorter ones.

---

---

*Definition: Structured programs* are programs broken into several small sections of code.

---

***Analysis:*** The code in Listing P12.1 demonstrates a programming approach called *structured programming* that you should incorporate into your own programs. Form_Load() isn't one huge event procedure because the Call statements in lines 6 through 10 help break up the code into smaller and more manageable code routines. All of the called code executes when the Form_Load() event procedure runs (right before the user sees the form on the screen), but the five subroutines provide a way for you to break the program into sections that let you zero in later on parts that you need to modify.

Suppose that you need to update the pricing information. If you study only the Form_Load() procedure, you can see that the FillCells subroutine fills the grid's cells with values. Therefore, you could go straight to that subroutine if you needed to change the prices.

Listing P12.2 contains the code for the routines called by the Form_Load() procedure. Study the code to familiarize yourself with your own grid's initialization requirements.

**Warning:** Not all of the code for the FillCells and Titles subroutines is listed in Listing P12.2 because the code is lengthy and repetitive. The long string of assignment statements helps to show how loading and saving the pricing information from and to the disk would be helpful once the company added many more customers. You don't want to look through and modify long strings of assignments every time that a price needs updating, you get a new customer, or you lose a customer. Using the disk for pricing, however, would probably require at least one additional program that enables you to update the disk price information directly.

**Listing P12.2. The code that the Form_Load() procedure calls to initialize the grid and other controls.**

```
1: Sub InitScrolls ()
2: ' Set both scroll bars to their maximum values
3: hscIncrease.Value = 15
4: hscDecrease.Value = 15
5: End Sub
6:
7: Sub CenterCells ()
8: ' Sets the justification of the grid's
9: ' cells to center alignment

10: Dim Apps As Integer ' 5 applications yearly
11: ' Center all values in the grid
12: For Apps = 0 To 5 ' 5 applications
13: grdLawn.Col = Apps ' Locate next column
14: ' Center the fixed cells in this column
15: grdLawn.FixedAlignment(Apps) = 2

16: Next Apps
17: For Apps = 1 To 5 ' 5 applications
18: ' Center the non-fixed cells in this column
```

```
19: grdLawn.ColAlignment(Apps) = 2

20: Next Apps

21: End Sub

22:

23: Sub SizeCells ()

24: ' Specify the width of each cell

25: Dim Apps As Integer
' 5 Application

26: For Apps = 0 To 5

27: grdLawn.ColWidth(Apps) = 1100 ' Twips

28: Next Apps

29: End Sub

30:

31: Sub Titles ()

32: ' Fill in the column titles

33: grdLawn.Row = 0 ' All titles are in row #0

34: grdLawn.Col = 1

35: grdLawn.Text =
"Pre-Emerge"

36: grdLawn.Col = 2
```

```
37: grdLawn.Text = "Deep"

38: ' Rest of column titles are filled here

39: ' Fill in the row titles

40: grdLawn.Col = 0 ' Customer titles in first column

41: grdLawn.Row = 1

42: grdLawn.Text =
"Jones"

43: grdLawn.Row = 2

44: grdLawn.Text = "Smith"

45: ' Rest of row titles are filled here

46: End Sub

47:

48: Sub FillCells ()

49: ' Fill in all 40 cells one at a time.

50: ' This is tedious!

51: '

52: ' Normally, you
would fill these cells from

53: ' a disk file or through calculations.

54: grdLawn.Row = 1
```

```
55: grdLawn.Col = 1

56: grdLawn.Text = 43.16

57: grdLawn.Col = 2

58: grdLawn.Text = 41.56

59: grdLawn.Col = 3

60: grdLawn.Text = 34.57

61: grdLawn.Col = 4


62: grdLawn.Text = 27.49

63: grdLawn.Col = 5

64: grdLawn.Text = 41.34

65: grdLawn.Row = 2

66: grdLawn.Col = 1

67: grdLawn.Text = 43.56

68: ' Rest of cells filled here

69: End Sub
```

## Descriptions

1: The code for the subroutine procedure that sets the scroll bars to their initial maximum values.

2: A remark that explains the procedure.

3: Set the Increase scroll bar to its maximum value of 15, which represents a 15 percent increase when the user clicks the Increase command button.

4: Set the Decrease scroll bar to its maximum value of 15, which represents a 15 percent decrease when the user clicks the Decrease command button.

5: Terminate the subroutine.

6: A blank line to separate the procedures.

7: The code for the subroutine procedure that sets the alignment in all the grid's cells to centered justification.

8: A remark explains the procedure.

9: The remark continues.

10: Define a variable used to loop through the number of applications.

11: A remark explains the code.

12: Step through all of the table's six columns (the first column, column 0, holds the row titles).

13: Set the column number to the For loop's value.

14: A remark explains the code.

15: Center (the value of 2 sets the property to 2-Center) the alignment property of all the fixed rows in the selected column. Only the top row is fixed in the last five columns.

---

15: Visual Basic right-justifies cell values if you don't change the justification.

---

16: Continue the loop.

17: Step through the table's last five columns (these contain the nonfixed columns) to prepare for centering.

18: A remark explains the code.

19: Center (the value of 2 sets the property to 2-Center) the alignment property of all the nonfixed rows in the selected column.

20: Continue the loop.

21: Terminate the subroutine.

22: A blank line to separate the procedures.

23: The code for the subroutine procedure that sets the width of the cells.

24: A remark explains the procedure.

25: Define a variable used to loop through the number of applications.

26: Step through each column, including the first fixed column 0.

27: Set every cell to 1,100 twips wide.

---

27: Make sure that each cell is wide enough to hold all data and titles.

---

28: Continue the loop.

29: Terminate the subroutine.

30: A blank line to separate the procedures.

31: The code for the subroutine procedure that initializes all the titles in the grid's fixed rows and columns.

32: A remark explains the procedure.

33: Set up for the first row (row number 0).

34: Set up for the second column (column number 1).

35: Assign a title.

36: Move to the next column in the same row.

37: Assign a title.

38: A remark to explain that not all the code is shown.

39: A remark explains the code.

40: Set up for the first column (column number 0).

41: Set up for the second row (row number 1).

42: Assign a title.

43: Move to the next row in the same column.

44: Assign a title.

45: A remark to explain that not all the code is shown.

46: Terminates the subroutine.

47: A blank line to separate the procedures.

48: The code for the subroutine procedure that initializes all the values in the grid's nonfixed cells.

49: A remark explains the procedure.

50: The remark continues.

51: The remark continues.

52: The remark continues.

53: The remark continues.

54: Set up for the second row (the first nonfixed row).

---

54: The Row and Col properties determine which cell gets the next value.

---

55: Set up for the second column (the first nonfixed column).

56: Assign a price value.

57: Move to the next column.

58: Assign a price value.

59: Move to the next column.

60: Assign a price value.

61: Move to the next column.

62: Assign a price value.

63: Move to the next column.

64: Assign a price value.

65: Move to the next row.

66: Move to the next column.

67: Assign a price value.

68: A remark to explain that not all code is listed here.

69: Terminate the subroutine.

## Using the Command Buttons and Scroll Bars

The application is set up to allow the user to select one or more cells with the mouse and then update the price values for only those selected cells. The user can thus try various pricing scenarios to see which pricing structure works best for specific applications. When the user clicks either command button, the selected cell or cells all increase or decrease by the amount shown on the command button at the time of the click. All unselected cells remain unaffected by the price change.

The scroll bars affect the price of the change as well as the captions on the command buttons themselves. For example, if the user lowers the value of the Decrease scroll bar, the Decrease by... command button's caption changes accordingly. The next time the user clicks the command button, the selected cells will decrease in price by the amount indicated on the scroll bar. The code in Listing P12.3 describes the interaction between the two scroll bars and their appropriate command buttons.

**Listing P12.3. Managing the price changes through scroll bars, command buttons, and selected cells.**

```
1: Sub hscDecrease_Change ()

2: ' Change the command button's caption

3: cmdDecrease.Caption = "&Decrease by" & Str$(hscDecrease.Value) & "%"

4:
End Sub

5:

6: Sub hscIncrease_Change ()

7: ' Change the command button's caption

8: cmdIncrease.Caption = "&Increase by" & Str$(hscIncrease.Value) & "%"

9: End Sub

10:

11: Sub cmdDecrease_Click ()

12: ' Decrease
selected cell values

13: ' by the decreasing scroll bar percentage

14: Dim SelRows, SelCols As Integer

15: If (grdLawn.HighLight) Then ' If true...

16: For SelRows = grdLawn.SelStartRow To grdLawn.SelEndRow

17: For SelCols = grdLawn.SelStartCol To
grdLawn.SelEndCol
```

```
18: grdLawn.Row = SelRows

19: grdLawn.Col = SelCols

20: ' Decrease the cell by scroll bar amount

21: grdLawn.Text = grdLawn.Text - (hscDecrease.Value / 100 * grdLawn.Text)

22: grdLawn.Text = Format(grdLawn.Text, "Fixed")


23: Next SelCols

24: Next SelRows

25: End If

26: End Sub

27:

28: Sub cmdIncrease_Click ()

29: ' Increase selected cell values

30: ' by the amount of the increase scroll bar's percentage

31: Dim SelRows, SelCols As Integer

32: If
(grdLawn.HighLight) Then ' If true...

33: For SelRows = grdLawn.SelStartRow To grdLawn.SelEndRow

34: For SelCols = grdLawn.SelStartCol To grdLawn.SelEndCol

35: grdLawn.Row = SelRows
```

```
36: grdLawn.Col = SelCols

37: ' Increase the cell by scroll bar
amount

38: grdLawn.Text = grdLawn.Text * (1 + hscIncrease.Value / 100)

39: grdLawn.Text = Format(grdLawn.Text, "Fixed")

40: Next SelCols

41: Next SelRows

42: End If

43: End Sub

44:

45: Sub cmdExit_Click ()

46: End

47: End
Sub
```

## Description

1: The code for the event procedure that executes when the user changes the decreasing scroll bar.

2: A remark explains the procedure.

3: Change the command button's caption to reflect the new percentage decrease.

---

3: One event procedure can change the property values of another.

---

4: Terminate the procedure.

5: A blank line separates the procedures.

6: The code for the event procedure that executes when the user changes the increasing scroll bar.

7: A remark explains the procedure.

8: Change the command button's caption to reflect the new percentage increase.

9: Terminate the procedure.

10: A blank line separates the procedures.

11: The code for the event procedure that executes when the user clicks the Decrease command button.

12: A remark that explains the event procedure.

13: The remark continues.

14: Define two variables that control the selected cells.

15: If there are highlighted cells...

---

15: There are no selected cells if no cells are highlighted.

---

16: Step through the selected rows.

17: Step through the selected columns.

18: Set the row to be changed.

19: Set the column to be changed.

20: A remark explains the code.

21: Update the cell's value by the amount of the decrease.

22: Format the cell to two decimal places.

23: Continue stepping through the selected columns.

24: Continue stepping through the selected rows.

25: Terminate the If statement.

26: Terminate the procedure.

27: A blank line separates the procedures.

28: The code for the event procedure that executes when the user clicks the Increase command button.

29: A remark that explains the event procedure.

30: The remark continues.

31: Defines two variables that control the selected cells.

32: If there are highlighted cells...

33: Step through the selected rows.

34: Step through the selected columns.

35: Set the row to be changed.

36: Set the column to be changed.

37: A remark explains the code.

38: Update the cell's value by the amount of the increase.

---

38: Does not affect any cells that are unselected.

---

39: Format the cell to two decimal places.

40: Continue stepping through the selected columns.

41: Continue stepping through the selected rows.

42: Terminate the If statement.

43: Terminate the procedure.

44: A blank line separates the procedures.

45: The code for the event procedure that executes when the user clicks the Exit command button.

46: Terminate the program.

47: Terminate the procedure.

## Close the Application

You can now exit the application and exit Visual Basic. Congratulations! You are now a Visual Basic programmer!

# Visual Basic in 12 Easy Lessons

---



---

# Answers

## Lesson 1, Unit 1

1. A program is a list of instructions that tells the computer exactly what to do.
2. Absolutely nothing. A computer is a dumb machine that cannot function without the detailed instructions of a program.
3. You can buy a program or write one yourself.
4. The programs that you write do exactly what you want them to do once you get any bugs out of them.
5. Writing your own programs takes lots of time and effort.
6. True
7. Code is a program.
8. An error that you accidentally place in the programs that you write.
9. There are syntax errors and logic errors.
10. You've just put a syntax error in the program.
11. Visual Basic finds syntax errors for you.
12. Logic errors are much harder to find than syntax errors.
13. With hardwired control panels.
14. Non-electrical people could program computers.
15. On and off states of electricity.
16. The addition of the computer keyboard introduced programming to the masses.
17. Code is the program that you and other programmers create.
18. Machine language
19. FORTRAN
20. BASIC
21. Beginner's All-purpose Symbolic Instruction Code
22. Procedural programming languages lend themselves well to text-based DOS environments.
23. Event-driven programs lend themselves well to Windows-like environments.
24. Graphical User Interface
25. Two events might be a keypress or a mouse click.
26. The user might trigger any event in any order.
27. False. Visual Basic does require some procedural-like programming for certain functions.
28. True

# Lesson 1, Unit 2

1. True
2. True
3. Add a shortcut Ctrl + Shift + V keystroke using the File Properties menu after highlighting the Visual Basic icon in the program group.
4. False
5. Select File Run, then type A:\SETUP (You can type the command in uppercase or lowercase letters).
6. The value used if you don't type a different value.
7. VBPRIMER
8. Select File Exit to exit Visual Basic and return to Windows.
9. You may lose some or all of the program you're writing.
10. The Code window, Form window, Project window, Properties window, and the Toolbox window.
11. Visual Basic uses the Form window for the application's background.
12. False
13. True. Click on the dialog control box's upper-left corner.
14. Use the access keys to more quickly execute menu commands.
15. The toolbar provides push button access to common menu commands.
16. The Visual Basic menu bar follows the same naming standard for most of its menu commands as other major Windows programs use.
17. Eleven
18. Not all menu or toolbar commands are available at all times.
19. The measurement indicators describe the size and location of elements on the Form window.
20. A twip is 1/1440th of an inch.
21. The grid provides snap-to locations that keep controls in alignment with the grid's dots when you set the appropriate Options Environment command.

# Lesson 2, Unit 3

1. You must first load the application using File Open.
2. The project file holds the application's description.
3. .MAK

4. Load the project and look in the Project window.
5. To let the user interact with and direct the program. Users click, scroll, and manage the controls as a way to respond to a program's activities.
6. The Toolbox window holds the controls that you can add to your form. (Don't you think "Control window" would be a better name?)
7. The user needs an easy way to exit a program.
8. False. The user cannot directly change text displayed in a label control.
9. The text box control.
10. You can control the font size and style of the text.
11. The user can move the text cursor forward and backward in a text box as well as use the Ins and Del keys to insert and delete text.
12. The command button control
13. False
14. True
15. The user deselects a check box by selecting the box a second time.
16. The frame control
17. True
18. True
19. False
20. True
21. The form works as the application's background and holds the controls that you place there.
22. True
23. The user can click the command button with the mouse or press the Alt+ shortcut access keystroke if the command button's label contains an underlined character. There is a third way as well (not documented in this unit): The user can press Tab until the command button is highlighted and press Enter.
24. The command button next to the combo box informs Visual Basic that there is data to be added to the list.
25. Pamela should replace the simple combo box with a dropdown combo box control.

## Lesson 2, Unit 4

1. An event might be a user keypress, mouse movement, mouse click, menu selection, or virtually anything else that a user can do to respond to a program.
2. There are too many ways for the user to interact with the visual controls on the screen. A text-based program dictates the order of the user's involvement.
3. False. Sometimes Windows captures events for its own system use.

4. Properties are attributes that differentiate the actions and behaviors of controls.
5. Event procedures
6. Here are four properties (from a huge list of many others): font size, font style, the control size, and the control color
7. True
8. True
9. The Properties window
10. The process of updating the program later.
11. The three-letter prefix describes what kind of control you're working with.
12. Users will more speedily adapt to your programs when they follow Windows standards.
13. AUTOLOAD.MAK
14. CONSTANT.TXT
15. Names given to data values that make them easier to remember.
16. False
17. Double-click the control on the Form window.
18. The first part of the event procedure is the control name, followed by an underscore, followed by the name of the event (and parentheses come after that to let you know you're working with a procedure name and not a control name).
19. Wrapper code
20. End
21. The controls and form names inside the project.
22. Usually the project and form names are the same with a .MAK and a .FRM extension respectively.
23. A form
24. A combo box
25. A command button
26. Victor, less is better! Use fewer fonts or you'll confuse the user looking at your screen.
27. A. End is a reserved word.
    B. Names cannot begin with a number.
    C. Names cannot contain special characters such as a dollar sign.
    D. Names cannot contain hyphens.

## Lesson 3, Unit 5

1. The control that has Window s current attention. You can tell which one it is by the highlighting.
2. The order in which the controls receive the focus as the user presses the Tab key.
3. TabIndex

4. The Name property.
5. False
6. True
7. CONSTANT.TXT
8. An icon is a picture.
9. A point is 1/72nd of an inch.
10. A selection of colors from which you can choose.
11. D. All of the above.
12. Enabled.
13. 0, 1, or 2 representing left-justified, centered, or right-justified.
14. False; rarely will you have to set or change more than a few properties because the default values are so common for many of the properties.
15. A carriage return character sends the text cursor to the next line on the screen.
16. Dynamic Data Exchange.
17. True due to the Visible property.
18. Left, Right, Top, Width.
19. The label control.
20. The PasswordChar property.
21. The Cancel property.
22. Use the control dropdown selection box.

## Lesson 3, Unit 6

1. Yes, the form is an object as is the other controls.
2. False, label controls have no MultiLine property.
3. A pixel is the smallest screen element available.
4. The WindowState property.
5. Chopping off part of text.
6. False
7. False, text box controls can have scroll bars.
8. Set the AutoSize property to False.
9. False.
10. False.
11. The TabIndex property controls the focus order.
12. The Load event.

13. The Load event occurs before the Activate event occurs.
14. The Code window contains a Proc dropdown list box that contains all the events for any given object.
15. Labels can never receive the focus.
16. The name of the event procedure must be txtLastName_Change().
17. The KeyPreview property determines whether the form or control gets the keystrokes.
18. The Caption property.
19. Add the access key to a label that describes the text box contents. Make sure that the TabIndex of the label is sequentially one less than the TabIndex of the text box.
20. The labels could hide too many other form objects.
21. The label would automatically expand horizontally before you have a chance to set the WordWrap property.

## Lesson 4, Unit 7

1. A named storage location in memory.
2. The Dim statement.
3. A data type is a category of data that all Visual Basic values fall into.
4. Integer, Long, Single, Double, Currency, Variant, String
5. Integer
6. Double
7. Single, Double, and Currency
8. E means Exponent.
9. Scientific notation provides a shortcut notation for a wide range of numeric values.
10. False
11. True; variables can change but they do not have to.
12. You'll help eliminate bugs that can occur when you misspell a variable name.
13. A fixed-length string can only accept a preset number of characters while variable-length strings can accept strings of changing-length as a program runs.
14. True
15. True
16. A. 7
    B. 9
    C. 16
    D. (parentheses would be required around the addition to compute a true average)
    E. 6
    F. 40

17. Merge the strings together
18. + is used for either addition or string concatenation (depending on the context of its use) and & is used strictly for concatenation.
19. Dim SqFoot As Single
20. SalesPrice = Price / Discount
    Tax = TaxRate * SalesPrice
21. Name is a reserved word and cannot be a variable name.
22. Judy cannot define more than one variable with the same name. There is an advanced exception to this rule that you'll learn in Lesson 7.
23. Larry cannot use a dollar sign or commas in constant values.

## Lesson 4, Unit 8

1. A condition is a relation that you can check for in your program.
2. False, conditional tests produce one of two results, true and false
3. False but the parentheses help clarify the code
4. <, >, <=, >=, =, <>
5. A. False
   B. True
   C. True
   D. True
   E. True
   F. False
6. The ASCII table
7. The If statement makes decisions (as well as the Select Case)
8. The If-Else statement
9. False
10. A nested If is an If statement within another If statement
11. The Select Case
12. Four: Case value, Case Is, Case To, and Case Else
13. Nothing happens and control falls to the statement following the End Select statement
14. The code in the Case Else executes before control returns to the statement following the End Select statement
15. Case To
16. Case Is
17. If (M = 3) AND (P = 4) Then
    TestIt = "Yes"

End If

18. If (d >= 3) Or (p < 9) Then
19. Select Case Age
Case Is <0: lblTitle.Text = "Age Error"
Case Is <5: lblTitle.Text = "Too young"
Case 5: lblTitle.Text = "Kindergarten"
Case 6 To 11: lblTitle.Text = "Elementary"
lblSchool.Text = "Lincoln"
Case 12 To 15: lblTitle.Text = "Intermediate"
lblSchool.Text = "Washington"
Case 16 To 18: lblTitle.Text = "High School"
lblSchool.Text = "Betsy Ross"
Case Else: lblTitle.Text = "College"
lblSchool.Text = "University"
End Select
20. There is no End Else statement.

## Lesson 5, Unit 9

1. A comment that helps explain a program's logic.
2. To help aid programmers with program maintenance.
3. False
4. True
5. Two: The Rem statement and the apostrophe remark
6. To state the programmer's name and the date the program was written
To describe in the (general) procedure the overall goal of the program
To describe at the top of all procedures the overall goal of that procedure
To explain tricky or difficult statements so that others who modify the program later can understand the lines of code without having to decipher cryptic code
7. Programmers
8. False; add remarks when you write your program so you'll be able to understand the program more easily when you modify it later.
9. False
10. Specify a type value that signifies which group of command buttons you need.
11. By specifying a msg value.
12. Modal indicates whether or not the user must respond to a message box before switching to another application.
13. True
14. False

15. To get answers
16. Strings and variants
17. Check for a null string, "", to see if the user pressed Cancel or OK.
18. Press the OK command button (or press Enter which clicks the OK command button)
19. Four
20. None
21. Use a proper type value
22. Seven
23. Seven
24. Make sure that the type argument includes the value of 4096 or the MB_SYSTEMMODAL named constant.
25. Bob can use the only the apostrophe if he wants to include a remark at the end of a line. Bob must change the code to this:
26. do until (endOfFile) ' Continue until the end

## Lesson 5, Unit 10

1. A loop is the repetition of a block of Visual Basic statements.
2. Five
3. Four
4. True
5. A loop that repeats forever (at least until the user intervenes to stop the loop)
6. Keep asking the user for the answer, using a loop, until the user enters a correct response.
7. 9
8. 10
9. 0
10. 10
11. 1
12. The For statement
13. The Do statement
14. The Do While loop continues executing while a relational test is true, whereas the Do Until loop continues executing while a relational test is false.
15. Visual Basic checks the relational test at the top of the loop using a Do While statement, whereas, Visual Basic checks the relational test at the bottom of the loop using a Do-Loop While statement.
16. An iteration is one complete loop cycle.
17. Use a negative Step value.

18. The increment's Step value must be negative.
19. The Exit Do and the Exit For statements end loops before their natural termination.
20. Dim Guess, Ans As Integer
    Guess = 34 ' The user must guess this
    Do
    Ans = InputBox("Make a guess...", "Guessing Game")
    ' Tell the user about the wrong guess
    If (Ans <> Guess) Then
    MsgBox "You guessed incorrectly. Try again."
    End If
    Loop While Ans <> Guess
21. Larry changes the variable named i inside the loop, keeping the For loop's controlling variable from ever triggering the end of the loop.
22. Kim must switch the starting and ending values like this:
    For I = 1 To 100 Step 1

## Lesson 6, Unit 11

1. Arrays let you step through large amounts of data without the need to reference multiple variable names.
2. A single item from the array's list of values
3. The array subscript is the number that references different array elements.
4. One
5. 20
6. True
7. Use the subscript
8. The Static statement defines arrays
9. 1
10. The For loop works best for array processing.
11. List and combo boxes
12. False
13. True as long as the MultiSelect property is set to True.
14. True
15. The methods initialize, count, and remove items from list and combo box controls.
16. The AddItem method
17. True
18. False

19. True
20. True
21. The Selected method
22. False
23. The Sorted property
24. Static StdAges(3) As Integer
25. The MultiSelect property
26. Sub cmdGoAway_Click()
    lstVals.Clear
    End Sub
27. The command button can trigger an event procedure that uses the AddItem method to add the new item to the combo box.
28. Carla cannot define a list box as if the list box were a variable array. Carla must use the list box control on the Toolbox window to place a list box on the form.

## Lesson 6, Unit 12

1. The option button
2. The check box
3. Option buttons imitate older radios with push button station selectors.
4. The first option button will no longer be selected.
5. True
6. The Alignment property determines if the option or check box description appears to the right or left of the button or check box.
7. The Value property
8. The Value property
9. False
10. True
11. A holder of other controls such as option buttons.
12. False
13. A special character that performs an action on the screen.
14. Concatenate a combination of carriage-return/line feed character strings within text to break the lines onto separate label lines.
15. Changes a number to its ASCII character equivalent.
16. An array of controls that are the same control type.
17. All arrays, including control arrays, have only one name.

18. By a subscript
19. Zero
20. One
21. Index
22. By displaying a message box that makes sure you want to create a control array.
23. cmdAll_DblClick (Index As Integer)
24. Opal needs to draw her application's option buttons *within* their frames, not move them into the frames.

## Lesson 7, Unit 13

1. An argument is a value that you pass to a function inside the function's parentheses.
2. False
3. True
4. False
5. There are three different integer functions because there are different ways to round numbers.
6. A. 61
   B. -62
   C. -61
   D. 422
7. False; CInt() is a true rounding function whereas Int() and Fix() always round positive arguments down to the integer equal to or lower than the argument.
8. The Asc() function is the mirror-image function to Chr$().
9. A. "ABCDEFG"
   B. "abcdefg"
10. Anything would hold the string "78.1"
11. A. S
    B. Sam
    C.
    D. ams
    E. ams
12. True, see question #18 and #19.
13. True
14. Str$() (or Str())
15. Pass the numeric value to Str$() before concatenating the value to the prompting string.
16. Send to Len() a double-precision variable and multiply the result by 250.
17. Dim ANum, ANumSq As Single

```
ANum = Val(InputBox$("Enter a number to square"))
MsgBox "The square of your number is" & Str$(Sqr(ANum))
```

18. 
```
First = InputBox$("What is your first name?")
Last = InputBox$("What is your last name?")
MsgBox "Your name has" & Str$(Len(First) + Len(Last)) & "letters"
```
19. ValAsc = Asc("P")
20. Rudy needs to convert the weight value to a long integer because he is computing a value too large for an integer result.

## Lesson 7, Unit 14

1. True
2. The time and date functions get their values from your computer's internal clock and calendar.
3. 24-hour times do not use the AM or PM designators. Add 12 to all times after 12:59 PM to get the 24-hour time.
4. True
5. False
6. 19:54
7. Date$() returns a string and Date() returns a variant data type.
8. False; Visual Basic supports only a Now() function.
9. Now() returns a value based on the 12-hour clock.
10. False
11. False; Date and Time (statements not functions) set your computer's date and time.
12. Date$()
13. Timer() returns the number of seconds since midnight as a double-precision value.
14. A byte is a character of memory.
15. The VarType 7 data type
16. TimeValue() accepts three argument values, the hour, minute, and second, and forms a time whereas Hour(), Minute(), and Second() all accept a full time value and return individual pieces of the time.
17. A logical value is the result of a relational comparison.
18. Format() and Format$()
19. Format() returns a variant data type and Format$() returns a string.
20. A thousands separator is a comma (or period for some international settings) that separates every three digits to the left of the decimal point in values more than 999.
21. The "Fixed" fixed-format string does not enclose negative values in parentheses and does not display a thousands separator in large numbers.

22. Time$ will not accept 12-hour values that contain the AM or PM designator. You can use the Right$() function to see if the user happened to enter AM or PM.

## Lesson 8, Unit 15

1. Technically, the answer is false. We're splitting hairs, however. An event procedure is a specialized form of a subroutine procedure.
2. Use the Call statement to call subroutine procedures.
3. Use the function procedure's name inside an expression or statement.
4. You can select from the View menu bar command or type Sub or Function below another procedure in the Code window.
5. If a dollar sign appears after the function name, the function returns a string.
6. A function procedure returns a value back to the calling procedure.
7. Assign the function's return value to the function name as if the function name were a variable.
8. F2
9. The programmer must supply the code for function procedures.
10. Module
11. .BAS
12. You do not yet know how to pass data between procedures.
13. Option Base and Option Explicit
14. True
15. True
16. Sub PrReport()

    End
17. Function GetValue()

    End Function
18. GetPi = 3.14159
19. There are no parentheses around the argument list so you cannot use the Call keyword.

## Lesson 8, Unit 16

1. Local
2. Module

3. Global
4. False
5. True
6. True
7. Global
8. Module
9. As global constants
10. Defined global named constants
11. Nowhere else in the program are you allowed to assign values to constants.
12. In the (general) procedure of a non-form module.
13. True; one can be local to one procedure and one local to another. You can also have a local variable that has the same name as a global variable. The local variable will be the active variable inside its procedure.
14. The values in local variables go away when the procedure ends.
15. The sending procedure
16. The receiving procedure
17. False
18. The variables that you pass are local to the calling function so the receiving function needs to know the data types of the arguments received.
19. False
20. False; you can pass several arguments to a function procedure but return only one value.
21. Two
22. Passed data may be changed by the receiving procedure and those changes remain in effect when the sending procedure regains control.
23. Passed data cannot be changed by the receiving procedure except within the boundaries of the receiving procedure.
24. Use the ByVal keyword or enclose passed arguments in parentheses.
25. The UBound() function
26. Global LastName As String
    Global Age As Integer
27. Global Const NDAYS = 7
    Global Const NMONTHS = 12
28. A. You must initialize a constant and constants cannot have data type declarations.
    B. Constants cannot have data type declarations.
    C. Always define constants using the Const keyword.
    D. You cannot initialize a global variable at the time that you define the variable.
29. Merle should never code the subscript value inside a receiving array's parentheses. Leave the parentheses blank and use the UBoumd() function inside the procedure to find the highest subscript in the array.

# Lesson 9, Unit 17

1. File data
2. The user can select from drives, paths, and filenames when you use a dialog box. The user can too easily make mistakes when entering a filename in response to an input box.
3. A set of controls that get user information.
4. Add a frame to the form that holds the file controls. The frame lets you group the file controls together without interfering with other controls such as other list boxes.
5. Set the frame's Visible property to True or False.
6. Three
7. At runtime
8. A pattern, using the * and ? filename wildcard characters, that determine which files are to be displayed in the file list box.
9. When one file control changes, that change usually affects other file controls.
10. Manage the drive list box first because more is affected by the drive list box change than by changes made to the other file controls.
11. The directory and the filename list box controls.
12. The filename list box control is the only control that needs to be updated when the user changes the directory list box.
13. The DidCancel variable contains either True or False.
14. The Change event.
15. True
16. Both ACCT* and ACCT*.* work.
17. Both *.ex? and *.ex* work.
18. drvDisk.Drive = "d:"
    dirList.Path = "d:\vbprimer\direng" ' Root directory

# Lesson 9, Unit 18

1. A collection of related data, programs, or a document.
2. To hold long-term data.
3. True as long as the files reside in separate directories or disks.
4. Open
5. Input, Output, and Append

6. the file number associates a file with a number.
7. Visual Basic creates the file.
8. Visual Basic overwrites the file.
9. Visual Basic creates the file.
10. Visual Basic adds to the end of the file.
11. Visual Basic displays an error.
12. 1 to 255
13. False
14. FILES=
15. FreeFile()
16. Close
17. Close ensures that the file is safely stored away in case a power failure occurs.
18. The statement closes all open files.
19. Write#
20. A comma
21. A pound sign
22. A quotation mark
23. Input#
24. A line from a file
25. The Eof() function tests for the end of file.
26. Line Input#
27. "Peach", 34.54, 1, "98"
28. Rusty must use the following Line Input# command:
    Line Input #1, MyRecord
29. Close 3, 19
30. Open "names.dat" for Append As #7
31. Sub ReadData ()
    ' Reads array data from a file and reports the data
    ' Assume that 200 values were read
    Static CNames(200) As String, CBalc(200) As Currency
    Static CDate(200) As Variant, CRegion(200) As Integer
    Dim NumVals As Integer ' Count of records
    Dim ctr As Integer ' For loop control

    NumVals = 1 ' Start the count
    ' Reads the file records assuming
    ' four values on each line
    Open "c:\mktg.dat" For Input As #1
    Input #1, CNames(NumVals), CBalc(NumVals), CDate(NumVals), CRegion(NumVals)

```
Do Until (Eof(1) = True)
NumVals = NumVals + 1 ' Increment counter
If (NumVals = 201) Then
MsgBox "First 200 values are now read", MB_ICONEXCLAMATION
Exit Do
End If
Input #1, CNames(NumVals), CBalc(NumVals), CDate(NumVals), CRegion(NumVals)
Loop

' When loop ends, NumVals holds one too many
NumVals = NumVals - 1

' The following loop is for reporting the data
For ctr = 1 To NumVals
' Code goes here that outputs the array
' data to the printer
'
Next ctr
Close #1

End Sub
```

## Lesson 10, Unit 19

1. The Menu Design window designs menus.
2. True
3. False
4. True
5. mnu
6. A pull-down menu that appears when you click a menu bar item.
7. The up and down arrow command buttons let you rearrange menu options.
8. The left and right arrow command buttons let you indent or remove indentions that indicate submenu items.
9. Checked
10. Visible
11. True
12. A single hyphen
13. Click
14. Chr$(8) (the backspace character) right-justifies menu bar items.

15. True
16. Frank needs to concatenate the Chr$(8) at runtime in the Form_Load() procedure.
17. mnuWindowSplit
18. mnuViewBar1 and mnuViewBar2
19. Select the Enabled property to gray out certain menu items.

## Lesson 10, Unit 20

1. The Timer control
2. The Interval property
3. The user does not see the timer control on the running program's form.
4. Nothing.
5. False; the timer control triggers its own events.
6. Milliseconds
7. False
8. One thousandth of a second
9. Ten seconds
10. One minute
11. tmr
12. The Interval property can hold values larger than integers can hold.
13. Timer()
14. False; you can only trigger a timer event to occur repeatedly
15. The Windows environment can sometimes cause a Visual Basic application to skip a time interval
16. Add a timer control to the project with an Interval property value of 1000 so the timer event occurs every second. Update the time display inside the timer control's Timer() event procedure.

## Lesson 10, Unit 21

1. Print
2. The Windows Print Manager collects all printing and routes the output to the appropriate printer.
3. The Windows Print Manager recognizes all installed printers so that individual programs do not have to.
4. Online means that the printer is ready and has paper.
5. Issue a message box that warns the user of subsequent output.

6. The Printer object is a generic Visual Basic object that collects all printed output and sends that output to the Windows Print Manager.
7. True
8. The smallest point on a printer or screen.
9. The Print method
10. False
11. False
12. The positive numbers each contain a hidden plus sign that does not print.
13. A zone is a kind of built-in tab stop that occurs every 14 columns of printed output.
14. True
15. Use Chr$(34)
16. True
17. The distance between the bottoms of the characters on one line to the bottoms of the characters on the line above.
18. Caroline is not taking into account the fact that Windows printers support different fonts and font sizes that may require calculation to find appropriate page lengths.
19. The second Print slips 10 spaces before printing, whereas the first Print begins printing in column 10.
20. Line1Line2
21. The Spanish N is ñ.
22. Printer.Print Tab(57); "America"

## Lesson 11, Unit 22

1. The line and shape controls let you draw geometric shapes on a form.
2. Six
3. BorderStyle
4. The Shape property
5. The picture box control and the image control
6. The image control is more efficient than the picture box control.
7. Bitmaps, metafiles, and icons.
8. The Shrink property
9. The Picture property
10. LoadPicture()
11. Jean cannot change the line appearance for any line thicker than one twip.
12. Jean would have to draw several smaller thick lines to create the appearance of a single dashed

line.

13. Nothing appears when the LoadPicture() function executes because of the null argument.
14. Indicate the blue border using the BorderColor property, the red diagonal line using the FillColor property, and the green interior with the BackColor property.

## Lesson 12, Unit 23

1. 2
2. The user can select from the scroll bar's relative position instead of typing specific values.
3. The amount that the scroll bar values change when the user adjusts a scroll bar's value.
4. By clicking within the shaft on either side of the scroll bar.
5. The smallest and largest values that a scroll bar can represent.
6. hsb and vsb are the preferred naming prefixes.
7. True
8. Make sure that the grid's custom control file is added to the project.
9. GRID.VBX
10. False
11. One of the individual row and column intersection locations inside a grid.
12. False
13. When the grid's size is not large enough to display the full grid.
14. Row and Col
15. SelEndCol, SelEndRow, SelStartCol, and SelEndCol
16. 12
17. MousePointer
18. The caret
19. The cursor
20. When a control's default MousePointer property is not the arrow. This typically occurs only for custom controls.
21. The hourglass
22. Through event procedure arguments
23. True
24. The number of fixed rows and fixed columns must be no more than two fewer than the total number of rows and columns. You will have to change the grid's Enabled property to False if you don't want the user to have any additional control over a grid.
25. Min: 32, Max: 212, SmallChange: 3, LargeChange: 8

# Lesson 12, Unit 24

1. Logic and syntax errors
2. Syntax error
3. A debugger is an online tool that lets you hunt down errors (bugs) in a program.
4. Use the Options Environment command to request interactive syntax checking.
5. True
6. False
7. Syntax errors are easier than logic errors to find.
8. Design mode, run mode, and break mode
9. Design mode
10. Look at Visual Basic's title bar to see the name of the current mode.
11. Break mode
12. Press Ctrl+Break, Select from the Run menu, click the break button on the toolbar, or set a breakpoint
13. A specific line that Visual Basic halts execution at during a program's run.
14. By highlighting the line of code and pressing F9, clicking on the toolbar, or by selecting from the Debug menu
15. Visual Basic highlights the breakpoint line
16. When Visual Basic reaches a breakpoint, Visual Basic halts the program's run before the breakpoint line executes.
17. Executing a program one line at a time, controllable by yourself instead of by Visual Basic.
18. The single footprint requests that Visual Basic single step through every line of code and the double footprint requests that Visual Basic single step through the breakpoint's current procedure but not single step through procedures called by the current procedure.
19. From the Instant Watch dialog box, you can look at variables and control values. From the Add Watch dialog box, you can request that Visual Basic enter a break mode when a certain expression becomes true or when a certain value is changed.
20. The Instant Watch is most handy for looking at variables during a breakpoint.
21. The Debug window
22. The Print method
23. The assignment statement
24. Jennifer must precede her Print methods with the Debug object.