3

Introduction to Visual Basic Programming

Objectives

- To write simple programs in Visual Basic.
- To become familiar with fundamental data types.
- To understand computer memory concepts.
- To be able to use arithmetic operators.
- To understand the precedence of arithmetic operators.
- To be able to write simple decision-making statements.

"Where shall I begin, please your majesty?" she asked. "Begin at the beginning," the king said, very gravely, "and go on till you come to the end; then stop."
Lewis Carroll

It is a capital mistake to theorize before one has data.
Arthur Conan Doyle

. . . the wisest prophets make sure of the event first.
Horace Walpole

An actor entering through the door, you've got nothing. But if he enters through the window, you've got a situation.
Billy Wilder

You shall see them on a beautiful quarto page, where a neat rivulet of text shall meander through a meadow or margin.
Richard Brinsley Sheridan

Exit, pursued by a bear.
William Shakespeare

Outline

3.1 Introduction
3.2 Visual Programming and Event-Driven Programming
3.3 A Simple Program: Printing a Line of Text on the Form
3.4 Another Simple Program: Adding **Integer**s
3.5 Memory Concepts
3.6 Arithmetic
3.7 Operator Precedence
3.8 Decision Making: Comparison Operators

Summary • Terminology • Common Programming Errors • Good Programming Practices • Testing and Debugging Tip • Software Engineering Observation • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

3.1 Introduction

The Visual Basic language facilitates a structured and disciplined approach to computer program design. In this chapter we introduce Visual Basic programming and present several examples that illustrate many important features. Each example is carefully analyzed one statement at a time. In Chapters 4 and 5 we present an introduction to structured programming.

3.2 Visual Programming and Event-Driven Programming

With visual programming, the programmer has the ability to create graphical user interfaces (GUIs) by pointing and clicking with the mouse. Visual programming eliminates the need for the programmer to write code that generates the form, code for all the form's properties, code for form placement on the screen, code to create and place a **Label** on the form, code to change foreground and background colors, etc. All of this code is provided as part of the project. The programmer does not need to be an expert Windows programmer to create functional Windows programs. The programmer creates the GUI and writes code to describe what happens when the user interacts (clicks, presses a key, double-clicks, etc.) with the GUI. These notifications, called *events*, are passed into the program by Microsoft's Windows operating system.

Programming the code that responds to these events is called *event-driven programming*. With event-driven programs, the user dictates the order of program execution—not the programmer. Instead of the program "driving" the user, the user "drives" the program. With the user in control, using a computer becomes a much more user-friendly process. Consider, for example, a web browser. When opened, the web browser may or may not load a page by default. After the browser is loaded, it just "sits there" with nothing else happening. The browser will stay in this *event monitoring* state (i.e., listening for events) indefinitely. If the user presses a button, the browser then performs some action, but as soon as the browser is done performing the action it returns to the event monitoring state. Thus, user actions determine browser activity.

Event procedures are Visual Basic procedures that respond to events and are automatically generated by the Visual Basic. The programmer adds code to respond to specific

events. Only events that are relevant to a program need be coded. In the next section we demonstrate how to locate event procedures and add code to respond to events.

3.3 A Simple Program: Printing a Line of Text on the Form

Consider a simple program that prints a line of text on the form. The GUI contains two buttons, **Print** and **Exit**, and is shown in the left picture of Fig. 3.1. The right picture of Fig. 3.1 shows the result after **Print** is pressed many times.

Figure 3.2 lists the *object* (i.e., form, **`CommandButton`**, etc.) and some property settings. We have only listed the properties we changed. We also provide a brief property description.We refer to **`CommandButton`**s simply as *button*s.

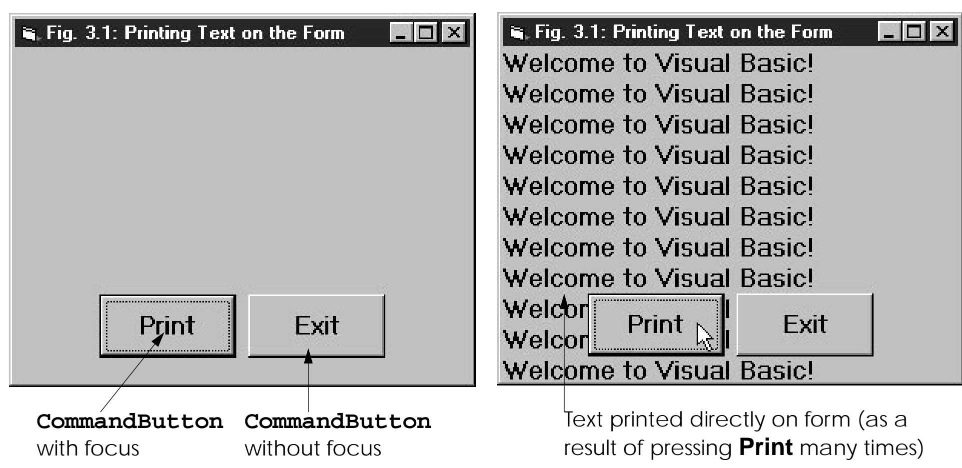


Fig. 3.1 Program that prints on the form.

Object	Property	Property setting	Description
form	**`Name`**	**`frmWelcome`**	Identifies the form.
	`Caption`	**`Fig. 3.1: Printing Text on the Form`**	Form title bar display.
	`Font`	**`MS Sans Serif Bold 12`** pt	Font for display on the form.
Print button	**`Name`**	**`cmdPrint`**	Identifies **Print** button.
	`Caption`	**`Print`**	Text that appears on button.
	`Font`	**`MS Sans Serif Bold 12`** pt	**`Caption`** text font.
	`TabIndex`	**`0`**	Tab order number.
Exit button	**`Name`**	**`cmdExit`**	Identifies **Exit** button.

Fig. 3.2 Object property settings (part 1 of 2).

Object	Property	Property setting	Description
	Caption	**Exit**	Text that appears on button.
	Font	**MS Sans Serif Bold 12** pt	**Caption** text font.
	TabIndex	**1**	Tab order number.

Fig. 3.2 Object property settings (part 2 of 2).

Good Programming Practice 3.1

*Prefix the name of **CommandButton**s with **cmd**. This allows easy identification of **CommandButton**s.*

The **Properties** window contains the ***Object box*** that determines which object's properties are displayed (Fig. 3.3). The **Object box** lists the form and all objects on the form. A selected object's properties are displayed in the **Properties** window.

The ***TabIndex*** property determines which control gets the *focus* (i.e., becomes the active control) when the *Tab* key is pressed at runtime. The control with a **TabIndex** value of **0** gets the initial focus. Pressing the *Tab* key at runtime transfers the focus to the control with a **TabIndex** of **1**. Eventually, if the *Tab* key is pressed enough times, the focus is transferred back to the control with a **TabIndex** of **0**. The focus for each control is displayed differently. For buttons, the one with the focus has a darker border around it and a dotted inner square on its face as shown in Fig. 3.1. Some controls, such as **Label**s, have a **TabIndex** property but are not capable of receiving the focus. In this situation, the next control (based upon **TabIndex** values) capable of receiving the focus gets it. By default, a control receives a **TabIndex** property based on the order in which it is added to the form. The first control added gets **0**, the next control added gets **1**, etc. A control's **TabIndex** property can be changed in the **Properties** window.

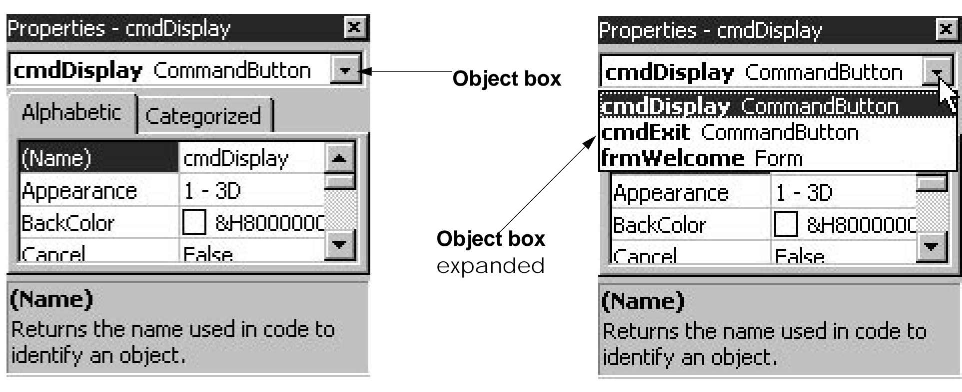


Fig. 3.3 **Properties** window.

We now switch over from the visual programming side to the event-driven programming side. If our program is going to print on the form, we must write code to accomplish this. With GUI and event-driven programming, the user decides when text is printed on the form by pressing **Print**. Each time **Print** is pressed, our program must respond by printing to the form. When the button is pressed does not matter; the fact that the button is pressed matters. Code must be written for the **Print** button's event procedure that receives this clicking (i.e., pressing) event.

When pressed, the **End** button terminates the program. Code must be written for the **End** button's event procedure that receives this clicking event. This event procedure for **End** is completely separate from the event procedure for **Print**. Separate event procedures make sense, because each button needs to respond differently.

Code is written in the ***Code*** window (Fig. 3.4). The **Code** window is displayed by either clicking the **Properties** window's **View Code** button or by double-clicking an object. The **View Code** button is disabled unless the form is visible. Figure 3.4 is the result of double-clicking the **Print** button at design time.

The code shown in Fig. 3.4 is generated by Visual Basic. The line

```
Private Sub cmdDisplay_Click()
```

begins the event procedure definition and is called the *procedure definition header*. The event procedure's name is **`cmdDisplay_Click`** (the parentheses **`()`** are necessary for syntax purposes). Visual Basic creates the event procedure name by appending the *event type* (**`Click`**) to the property **`Name`** with an underscore (_) added. ***`Private Sub`*** marks the beginning of the procedure. The ***`End Sub`*** statement marks the end of the procedure. Code that the programmer wants executed when **Print** is pressed is placed between the procedure definition header and the end of the procedure (i.e., **`End Sub`**). Figure 3.5 shows the **Code** window with code. We will discuss the code momentarily.

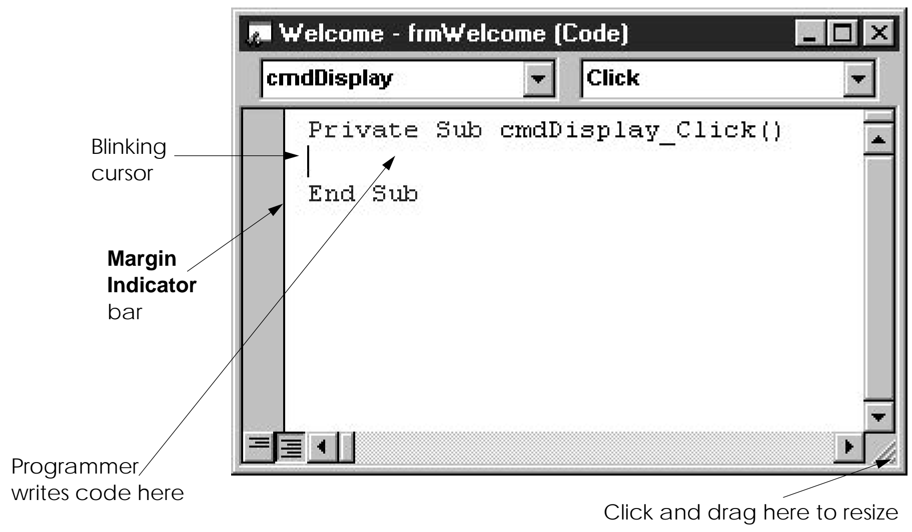


Fig. 3.4 **Code** window.

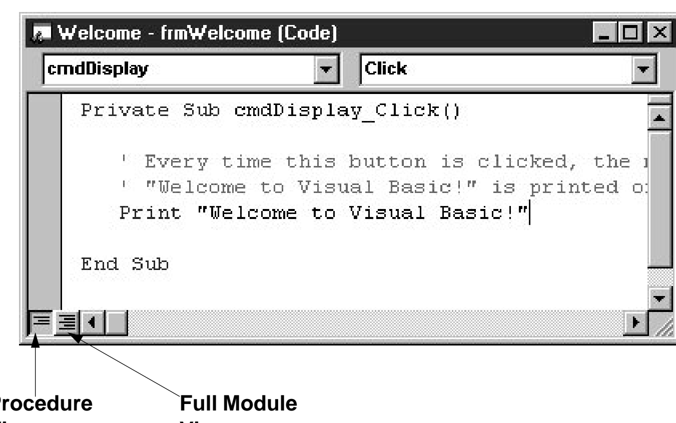


Fig. 3.5 **Code** window displaying code.

Figure 3.6 labels two buttons ***Procedure View*** and ***Full Module View***. **Procedure View** lists only one procedure at a time. **Full Module View** lists the complete code for the whole *module* (the form in this example) as shown in Fig. 3.6. The ***Procedure Separator*** separates one procedure from another. The default is **Full Module View**. We pressed the **Procedure View** button in Fig. 3.5. Any object's code can be accessed with the **Code** window's **Object box** and ***Procedure box***. The **Object box** lists the form and all objects associated with the form. The **Procedure box** lists the procedures associated with the object displayed in the **Object box**.

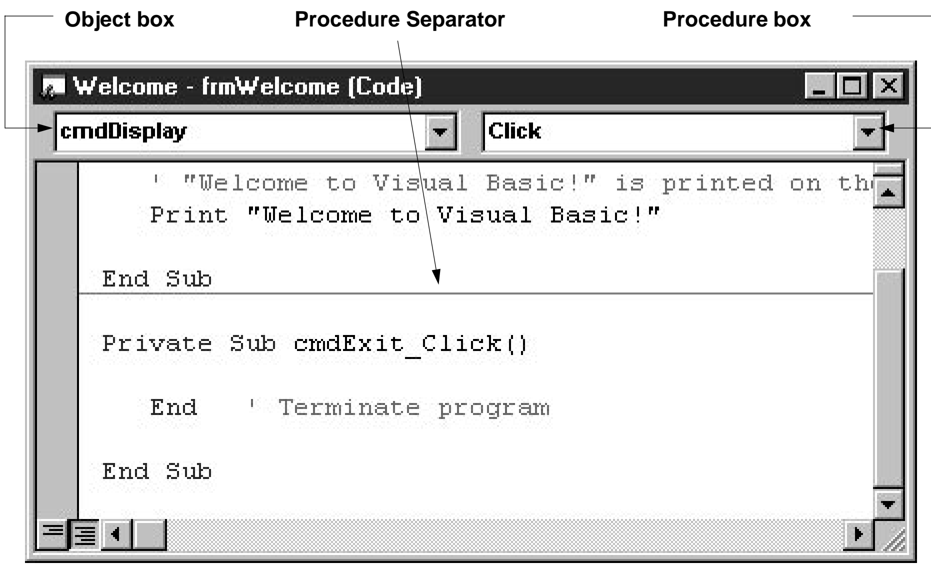


Fig. 3.6 **Code** window with **Full Module View** selected.

The program code is shown in Fig. 3.7. The line numbers to the left of the code are not part of the code but are placed there for reference purposes.

Procedure **cmdDisplay_Click** executes when button **Print** is pressed. The lines

```
' Every time this button is clicked, the message
' "Welcome to Visual Basic!" is printed on the form
```

are *comments*. Programmers insert comments to document programs and improve program readability. Comments also help other people read and understand your program code. Comments do not cause the computer to perform any action when a program is run. A comment can begin with either **'** or ***Rem*** (short for "remark") and is a *single-line comment* that terminates at the end of the current line. Most programmers use the single-quote style.

Good Programming Practice 3.2

Comments written to the right of a statement should be preceded by several spaces to enhance program readability.

Good Programming Practice 3.3

Visual Basic statements can be long. You might prefer to write comments above the line(s) of code you are documenting.

Good Programming Practice 3.4

Precede comments that occupy a single line with a blank line. The blank line makes the comment stand out and improves program readability.

The line

```
Print "Welcome to Visual Basic!"
```

prints the text "**Welcome to Visual Basic!**" on the form using the ***Print*** *method*. Each time this *statement* executes, the text is displayed on the next line. Method **Print** is a feature of the Visual Basic language and is unrelated to **cmdDisplay**'s **Caption** (**Print**).

Good Programming Practice 3.5

Indent statements inside the bodies of event procedures. We recommend three spaces of indentation. Indenting statements increases program readability.

```
1  Private Sub cmdDisplay_Click()
2
3     ' Every time this button is clicked, the message
4     ' "Welcome to Visual Basic!" is printed on the form
5     Print "Welcome to Visual Basic!"
6
7  End Sub
8
9  Private Sub cmdExit_Click()
10
11    End    ' Terminate program
12
13 End Sub
```

Fig. 3.7 Program code.

Drawing directly on the form using **`Print`** is not the best way of displaying information, especially if the form contains controls. As is shown in Fig. 3.1, a control can hide text that is displayed with **`Print`**. This problem is solved by displaying the text in a control. We demonstrate this in the next example.

The only statement in the **`cmdExit_Click`** event procedure is

```
End    ' Terminate program
```

The ***`End`*** statement terminates program execution (i.e., places the IDE in design mode). Note the comment's placement in the statement.

Software Engineering Observation 3.1

*Even though multiple **`End`** statements are permitted, use only one. Normal program termination should occur in only one place.*

When the user types a line of code and presses the *Enter* key, Visual Basic responds either by generating a *syntax error* (also called a *compile error*) or by changing the colors on the line. Colors may or may not change depending on what the user types.

A syntax error is a violation of the language syntax (i.e., a statement is not written correctly). Syntax errors occur when statements are missing information, when statements have extra information, when names are misspelled, etc. When a syntax error occurs, a *dialog* like Fig. 3.8 is displayed. Note that some syntax errors are not generated until the programmer attempts to enter run mode.

Testing and Debugging Tip 3.1

As Visual Basic processes the line you typed, it may find one or more syntax errors. Visual Basic will display an error message indicating what the problem is and where on the line the problem is occurring.

If a statement does not generate syntax errors when the *Enter* key is pressed, a coloring scheme (called *syntax color highlighting*) is imposed on the line of code. Comments are changed to green. The event procedure names remain black. Words recognized by Visual Basic (called *keywords* or *reserved words*) are changed to blue. Keywords (i.e., **`Private`**, **`Sub`**, **`End`**, **`Print`**, etc.) cannot be used for anything other than for the feature they represent. In addition to syntax color highlighting, Visual Basic may convert some lowercase letters to uppercase, and vice versa.

Common Programming Error 3.1

Using a keyword as a variable name is a syntax error.

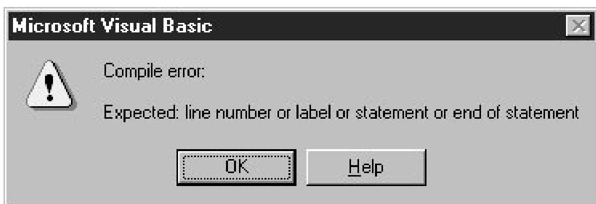


Fig. 3.8 Syntax error dialog.

Testing and Debugging Tip 3.2

Syntax color highlighting helps the programmer avoid using keywords accidentally.

The colors used for comments, keywords, etc. can be set using the ***Editor Format*** *tab* in the ***Options*** dialog (from the **Tools** menu). The **Option** dialog displaying the **Editor Format** tab is shown in Fig. 3.9.

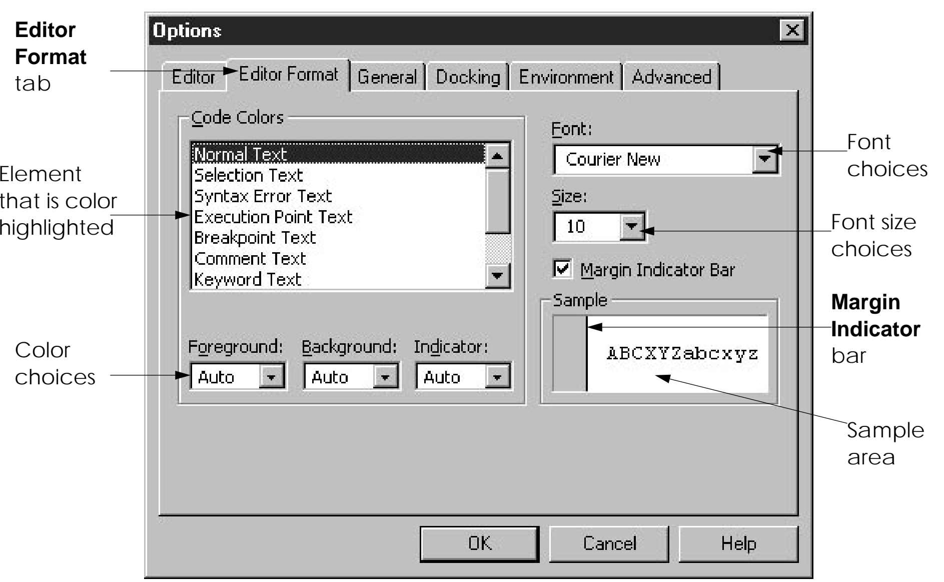


Fig. 3.9 **Options** dialog displaying **Editor Format** tag.

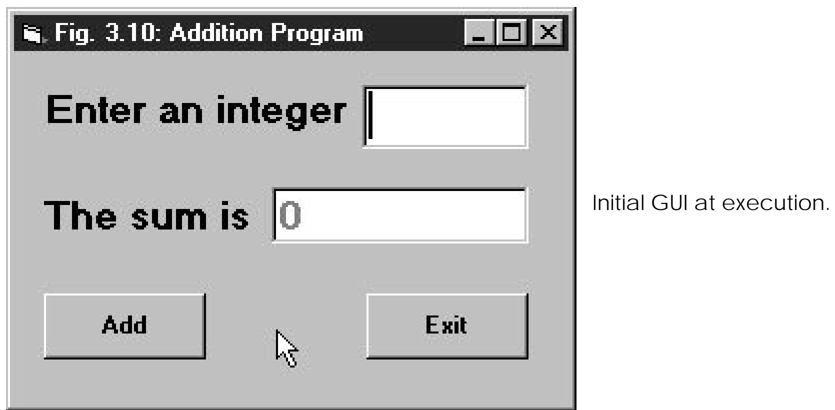
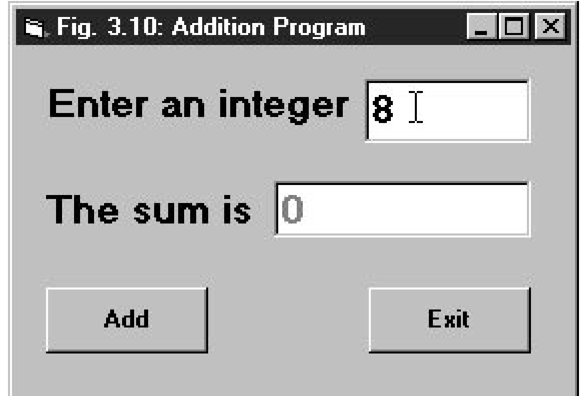
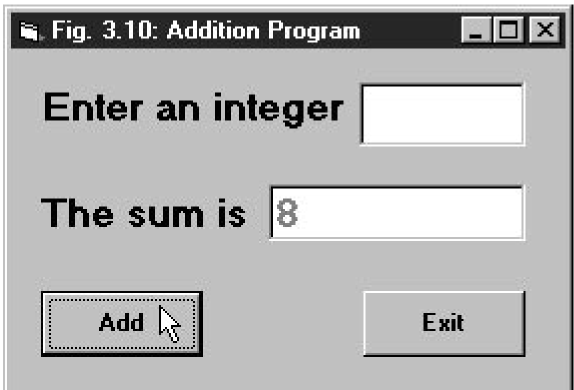


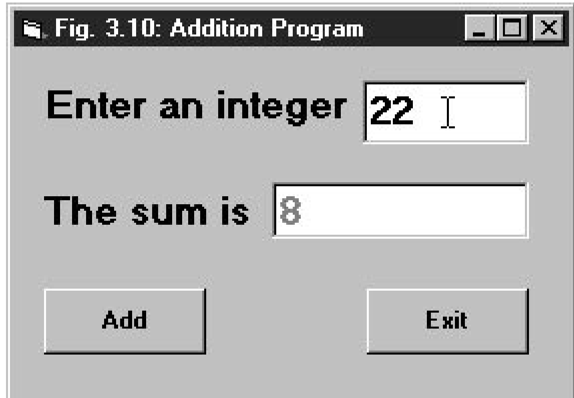
Fig. 3.10 Program that adds **`Integer`**s (part 1 of 3).



GUI after user has entered **8** in the first `TextBox`.

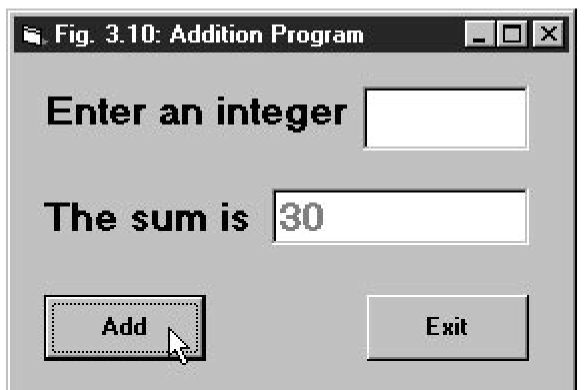


GUI after user has pressed **Add**. The value 8 is added to the sum and the sum is displayed in the second `TextBox`. The first `TextBox` is cleared.



GUI after user has entered **22** in the first `TextBox`.

Fig. 3.10 Program that adds `Integer`s (part 2 of 3).



GUI after user has pressed **Add**. The value 22 is added to the sum and the sum is displayed in the second **TextBox**. The first **TextBox** is cleared.

Fig. 3.10 Program that adds **Integer**s (part 3 of 3).

3.4 Another Simple Program: Adding Integers

Our next program obtains **Integer**s from the user, computes their sum and displays the result. The GUI consists of two **Label**s, two **TextBox**es and two buttons as shown in Fig. 3.10. The object properties are listed in Fig. 3.11 and the program is shown in Fig. 3.12.

Object	Icon	Property	Property setting	Property description
form		**Name**	**frmAddition**	Identifies the form.
		Caption	**Fig. 3.10: Addition Program**	Form title bar display.
Add button		**Name**	**cmdAdd**	Identifies **Add** button.
		Caption	**Add**	Text that appears on button.
Exit button		**Name**	**cmdExit**	Identifies **Exit** button.
		Caption	**Exit**	Text that appears on button.
Label	A	**Name**	**lblSum**	Identifies the **Label**.
		Caption	**The sum is**	Text **Label** displays.
Label	A	**Name**	**lblPrompt**	Identifies the **Label**.
		Caption	**Enter an integer**	Text **Label** displays.

Fig. 3.11 Object properties (part 1 of 2).

Object	Icon	Property	Property setting	Property description
TextBox		**Name**	**txtSum**	Identifies **TextBox**.
		Font	**MS San Serif Bold 14** pt	Font for **TextBox**.
		Text	**0**	Text that is displayed.
		Enabled	**False**	Enabled/disabled.
TextBox		**Name**	**txtInput**	Identifies **TextBox**.
		Font	**MS San Serif Bold 14** pt	Font for **TextBox**.
		MaxLength	**5**	Maximum length of character input.
		TabIndex	**0**	Tab order.
		Text	(empty)	Text that is displayed.

Fig. 3.11 Object properties (part 2 of 2).

```
 1  Dim sum As Integer                    ' Declare an Integer
 2
 3  Private Sub cmdAdd_Click()
 4     sum = sum + txtInput.Text          ' Add to sum
 5     txtInput.Text = ""                 ' Clear TextBox
 6     txtSum.Text = sum                  ' Display sum in TextBox
 7  End Sub
 8
 9  Private Sub cmdExit_Click()
10     End                                ' Terminate execution
11  End Sub
```

Fig. 3.12 Program code.

Good Programming Practice 3.6

Prefix the name of ***TextBox****es with* ***txt*** *to allow easy identification of* ***TextBox****es.*

The ***TextBox*** control is introduced in this example. This is the primary control for obtaining user input. **TextBox**es can also be used to display text. In our program one **TextBox** accepts input from the user and the other outputs the sum.

Like other controls, **TextBox**es have many properties. ***Text*** is the most commonly used **TextBox** property. The **Text** property stores the text for the **TextBox**. **TextBox**es have their ***Enabled*** *property* set to **True** by default. If the **Enabled** property is set to **False**, the user cannot interact with the **TextBox** and any text displayed in the **TextBox** is grayed. Object **txtSum** has its **Enabled** property set to **False**. Note that the text representing the sum appears gray, indicating that it is disabled.

The ***MaxLength*** property value limits how many characters can be entered in a **TextBox**. The default value is **0**, which means that any number of characters can be input. We set **txtInput**'s **MaxLength** value to **5**.

The first line of code resides in the *general declaration*. Statements placed in the general declaration are available to every event procedure. The general declaration can be accessed with the **Code** window's **Object box**. The statement

```
Dim sum As Integer
```

declares a variable named **sum**. A *variable* is a location in the computer's memory where a value can be stored for use by a program. A variable name is any valid *identifier*. *Variable names cannot be keywords and must begin with a letter*. The maximum length of a variable name is 255 characters containing only letters, numbers, and underscores. Visual Basic is not case-sensitive—uppercase and lowercase letters are treated the same, so **a1** and **A1** are considered identical. Keywords appear to be case-sensitive but they are not. Visual Basic automatically sets to uppercase the first letter of keywords, so typing **dim** would be changed to **Dim**.

Good Programming Practice 3.7

Begin each identifier with a lowercase letter. This will allow you to distinguish between a valid identifier and a keyword.

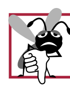
Common Programming Error 3.2

Attempting to declare a variable name that does not begin with a letter is a syntax error.

Good Programming Practice 3.8

Choosing meaningful variable names helps a program to be "self-documenting." A program becomes easier to understand simply by reading the code rather than having to read manuals or having to use excessive comments.

Keyword **Dim** *explicitly* (i.e., formally) declares variables. The clause beginning with the keyword ***As*** is part of the declaration and describes the *variable's type (i.e., what type of information can be stored)*. ***Integer*** means that the variable holds **Integer** values (i.e., whole numbers such as 8, –22, 0, 31298). **Integer**s are stored in two bytes of memory and have a range of –32767 to +32768. **Integer** variables are initialized to **0** by default. We discuss other data types in the next several chapters.



Common Programming Error 3.3

*Exceeding an **Integer**'s range is a run-time error.*

Variables can also be declared using special symbols called *type declaration characters*. For example, the declaration

```
Dim sum As Integer
```

could also be written as

```
Dim sum%
```

The *percent sign*, **%**, is the ***Integer*** *type declaration character*. Not all types have type declaration characters.

Common Programming Error 3.4

Attempting to use a type declaration character and keyword **`As`** *together is a syntax error.*

Variables can also be declared *implicitly* (without giving them a formal type) by mentioning the name. For example, consider the line

```
someVariable% = 8   ' Implicitly declare an Integer variable
```

which declares and initializes **`someVariable`**. When Visual Basic executes this line, **`someVariable`** is declared and given a value of **`8`** with *assignment operator* **`=`**. Visual Basic provides a means of forcing explicit declaration which we discuss later in this chapter.

Good Programming Practice 3.9

Explicitly declaring variables makes programs clearer.

If a variable is not given a type when its declared, its type defaults to ***`Variant`***. The **`Variant`** data type can hold any type of value (i.e., **`Integer`**s, **`Single`**s, etc.). Although the **`Variant`** type seems like a convenient type to use, it can be very tricky determining the type of the value stored. We discuss the **`Variant`** type in Chapter 4.

Common Programming Error 3.5

It is an error to assume that the **`As`** *clause in a declaration distributes to other variables on the same line. For example, writing the declaration* **`Dim x As Integer, y`** *and assuming that both* **`x`** *and* **`y`** *would be declared as* **`Integer`***s would be incorrect, when in fact the declaration would declare* **`x`** *to be an* **`Integer`** *and* **`y`** *(by default) to be a* **`Variant`***.*

Line 4

```
sum = sum + txtInput.Text
```

gets **`txtInput`**'s text and adds it to **`sum`**, storing the result in **`sum`**. To access a property, use the object's name followed by a period and the property name. Before the *addition operator*, **`+`**, adds the value input, the **`Text`** property value must be converted from a *string* (i.e., text) to an **`Integer`**. The conversion is done implicitly—no code need be written to force the conversion.

Common Programming Error 3.6

Expressions or values that cannot be implicitly converted result in run-time errors.

The previous assignment statement could have been written as

```
Let sum = sum + txtInput.Text
```

which uses keyword **`Let`**. When writing an assignment statement, keyword **`Let`** is optional. Our convention is to omit the keyword **`Let`**.

The lines

```
txtInput.Text = ""
txtSum.Text = sum
```

"clear" the characters from **txtInput** and display text in **txtSum**. The pair of double quotes, **""**, assigned to **txtInput.Text** is called an *empty string*. Assigning an empty string to **txtInput.Text** clears the **TextBox**. When **sum** (an **Integer**) is assigned to **txtSum.Text**, Visual Basic implicitly converts **sum**'s value to a string.

3.5 Memory Concepts

Variable names such as **sum** actually correspond to locations in the computer's memory. Every variable has a name, a type, a size and a value. In the addition program of Fig. 3.12, the statement

```
sum = sum + txtInput.Text
```

places into **sum**'s memory location the result of adding **sum** to **txtInput.Text**. Suppose the value of **txtInput.Text** is **"22"**. Visual Basic converts the string **"22"** to the **Integer 22** and adds it to the value contained in **sum**'s memory location. The result is then stored in **sum**'s memory location as shown in Fig. 3.13.

Whenever a value is placed in a memory location, the value replaces the previous value in that location. The process of storing a value in a memory location is known as *destructive read-in*. The statement

```
sum = sum + txtInput.Text
```

that performs the addition involves destructive read-in. This occurs when the result of the calculation is placed into location **sum** (destroying the previous value in **sum**).

Variable **sum** is used on the right side of the assignment expression. The value contained in **sum**'s memory location must be read in order to do the addition operation. Thus, when a value is read out of a memory location, the original value is preserved and the process is *nondestructive*.

3.6 Arithmetic

Most programs perform arithmetic calculations. The *arithmetic operators* are summarized in Fig. 3.14. Note the use of various special symbols not used in algebra. The *caret* (**^**) indicates exponentiation, and the *asterisk* (*****) indicates multiplication. The ***Integer*** *division operator* (****) and the *modulus* (***Mod***) operator will be discussed shortly. Most arithmetic operators are *binary operators* because they each operate on two *operands*. For example, the expression **sum + value** contains the binary operator **+** and the two operands **sum** and **value**.

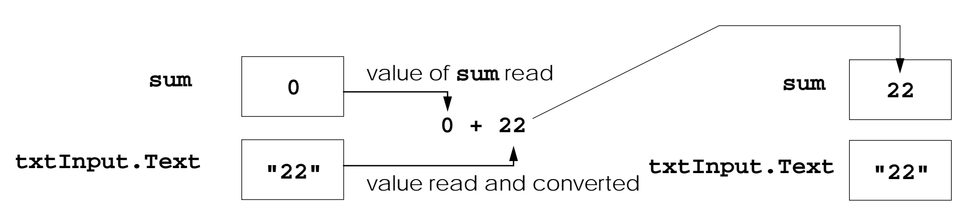


Fig. 3.13 Memory locations showing the names and values of variables.

Visual Basic operation	Arithmetic operator	Algebraic expression	Visual Basic expression
Addition	`+`	$x + y$	`x + y`
Subtraction	`-`	$z - 8$	`z - 8`
Multiplication	`*`	yb	`y * b`
Division (float)	`/`	v / u or $\frac{v}{u}$	`v / u`
Division (`Integer`)	`\`	none	`v \ u`
Exponentiation	`^`	q^p	`q ^ p`
Negation	`-`	$-e$	`-e`
Modulus	`Mod`	$q \bmod r$	`q Mod r`

Fig. 3.14 Arithmetic operators.

Visual Basic has separate operators for **Integer** division (the backslash, ****) and floating-point division (the forward slash, **/**). **Integer** division yields an **Integer** result; for example, the expression **7 \ 4** evaluates to **1**, and the expression **17 \ 5** evaluates to **3**. Note that any fractional part in **Integer** division is rounded before the division takes place. For example, the expression **7.7 \ 4** would yield **2**. The value **7.7** is rounded to **8**. The expression **7.3 \ 4** would yield **1**. The value **7.3** is rounded to **7**.

Floating-point division yields a *floating-point number* (i.e., a number with a decimal point such as 7.7). We will discuss floating-point numbers in Chapter 4.

The *modulus operator*, ***Mod***, yields the **Integer** remainder after **Integer** division. Like the **Integer** division operator, the modulus operator rounds any fractional part before performing the operation. The expression **x Mod y** yields the remainder after **x** is divided by **y**. A result of 0 indicates that **y** divides evenly into **x**. Thus, 20 **Mod** 5 yields 0, and 7 **Mod** 4 yields 3.

The *negation operator*, **-**, changes the sign of a number from positive to negative (or from negative to positive). The expression **-**8 changes the sign of 8 to negative, which yields **-**8. The negation operator is said to be a *unary operator*, because it operates on only one operand. The operand must appear to the right of the negation operator.

Arithmetic expressions must be written in *straight-line form* when entering programs into the computer. Thus, expressions such as "*a* raised to the power *b*" must be written as

```
a ^ b
```

so that all constants, variables and operators appear in a straight line. The algebraic notation

$$a^b$$

is generally not acceptable to compilers, although some special-purpose software packages do exist that support more natural notation for complex mathematical expressions.

Parentheses are used in expressions in much the same manner as in algebraic expressions. For example, to multiply **b** times the quantity **e + n** we write

```
b * (e + n)
```

3.7 Operator Precedence

Visual Basic applies the operators in arithmetic expressions in a sequence determined by the following rules of *operator precedence*, which are similar to those followed in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Thus, *parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer*. Parentheses are said to be at the "highest level of precedence." In cases of *nested* or *embedded* parentheses, the operators in the innermost pair of parentheses are applied first.
2. Exponentiation is applied next. If an expression contains several exponentiation operations, operators are applied from left to right.
3. Negation is applied next. If an expression contains several negation operations, operators are applied from left to right.
4. Multiplication and floating-point division operations are applied next. If an expression contains several multiplication and floating-point division operations, operators are applied from left to right. Multiplication and floating-point division are said to be on the same level of precedence.
5. **`Integer`** division is applied next. If an expression contains several **`Integer`** division operations, operators are applied from left to right.
6. Modulus operators are applied next. If an expression contains several modulus arithmetic operations, operators are applied from left to right.
7. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from left to right. Addition and subtraction also have the same level of precedence.

The rules of operator precedence enable Visual Basic to apply operators in the correct order. Figure 3.15 summarizes these rules of operator precedence. This table will be expanded as we introduce additional Visual Basic operators. A complete precedence chart is included in the Appendices.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`^`	Exponentiation	Evaluated second. If there are several, they are evaluated left to right.
`-`	Negation	Evaluated third. If there are several, they are evaluated left to right.
`*` or `/`	Multiplication and floating-point division	Evaluated fourth. If there are several, they are evaluated left to right.

Fig. 3.15 Precedence of arithmetic operators.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`\`	Division (`Integer`)	Evaluated fifth. If there are several, they are evaluated left to right.
`Mod`	Modulus	Evaluated sixth. If there are several, they are evaluated left to right.
`+` or `-`	Addition and subtraction	Evaluated last. If there are several, they are evaluated left to right.

Fig. 3.15 Precedence of arithmetic operators.

Now let us consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its Visual Basic equivalent.

The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \dfrac{a + b + c + d + e}{5}$

Visual Basic: `m = (a + b + c + d + e) / 5`

The parentheses are required because floating-point division has higher precedence than addition. The entire quantity **(a + b + c + d + e)** is to be divided by 5. If the parentheses are erroneously omitted, we obtain **a + b + c + d + e / 5**, which evaluates as

$$a + b + c + d + \frac{e}{5}$$

The following is the equation of a straight line:

Algebra: $y = mx + b$

Visual Basic: `y = m * x + b`

No parentheses are required. Multiplication has a higher precedence than addition and is applied first.

The following example contains exponentiation, multiplication, floating-point division, addition and subtraction operations:

Algebra: $z = pr^q + w/x - y$

Visual Basic: **Z = p * r ^ q + w / x – y** (order of operators: = 6, * 2, ^ 1, + 4, / 3, – 5)

The circled numbers under the statement indicate the order in which the operators are applied. The exponentiation operator is evaluated first. The multiplication and floating-point division operators are evaluated next in left-to-right order since they have higher precedence than assignment, addition and subtraction. Addition and subtraction operators are evaluated next in left-to-right order (addition followed by subtraction). The assignment operator is evaluated last.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the expression

```
a * (b + c) + c * (d + e)
```

does not contain nested parentheses. Rather, the parentheses are said to be on the same level of precedence.

To develop a better understanding of the rules of operator precedence, consider how a second-degree polynomial is evaluated.

y	=	a	*	x	^	2	+	b	*	x	+	c
	6		2		1		4		3		5	

The circled numbers under the statement indicate the order in which Visual Basic applies the operators.

Suppose that variables **a**, **b**, **c** and **x** are initialized as follows: **a = 2**, **b = 3**, **c = 7** and **x = 5**. Figure 3.16 illustrates the order in which the operators are applied in the preceding second-degree polynomial.

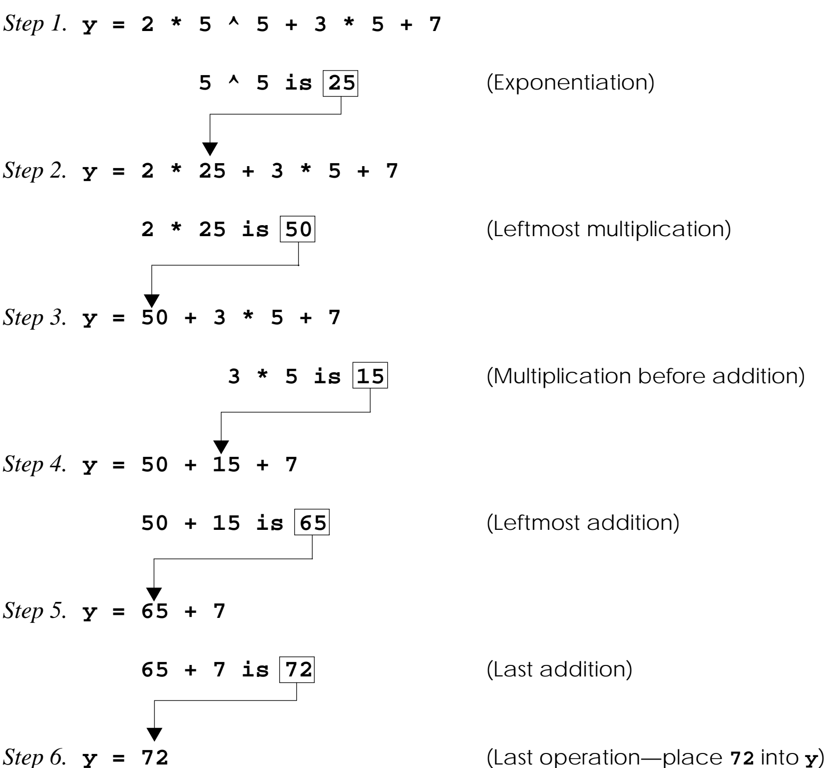


Fig. 3.16 Order in which operators in a second-degree polynomial are evaluated.

As in algebra, it is acceptable to place extra parentheses in an expression to make the expression clearer. Unnecessary parentheses are also called *redundant parentheses*. For example, the preceding assignment statement could be parenthesized as follows without changing its meaning:

```
y = (a * x ^ 2) + (b * x) + c
```

Good Programming Practice 3.10

Placing extra parentheses in an expression can make that expression clearer.

3.8 Decision Making: Comparison Operators

This section introduces a simple version of Visual Basic's ***If/Then*** *structure* that allows a program to make a decision based on the truth or falsity of some *condition*. If the condition is met (i.e., the condition is ***True***), the statement in the body of the **If/Then** structure is executed. If the condition is not met (i.e., the condition is ***False***), the body statement is not executed.

Conditions in **If/Then** structures can be formed by using the *comparison operators* summarized in Fig. 3.17. The comparison operators all have the same level of precedence.

Common Programming Error 3.7

Reversing the order of the symbols in the operators **<>**, **>=** *and* **<=** *as in* **><**, **=>** *and* **=<**, *respectively, are each syntax errors.*

Common Programming Error 3.8

Writing a statement such as **x = y = 0** *and assuming that the variables* **x** *and* **y** *are both assigned zero, when in fact comparisons are taking place, can lead to subtle logic errors.*

Good Programming Practice 3.11

Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, use parentheses to force the order, exactly as you would do in algebraic expressions.

Standard algebraic equality operator or relational operator	Visual Basic comparison operator	Example of Visual Basic condition	Meaning of Visual Basic condition
=	`=`	`d = g`	**d** is equal to **g**
≠	`<>`	`s <> r`	**s** is not equal to **r**
>	`>`	`y > i`	**y** is greater than **i**
<	`<`	`p < m`	**p** is less than **m**
≥	`>=`	`c >= e`	**c** is greater than or equal to **e**
≤	`<=`	`m <= s`	**m** is less than or equal to **s**

Fig. 3.17 Comparison operators.

The next example uses six **`If`**/**`Then`** statements to compare two numbers input by the user. The GUI is shown in Fig. 3.18, the properties in Fig. 3.19 and the code in Fig. 3.20.

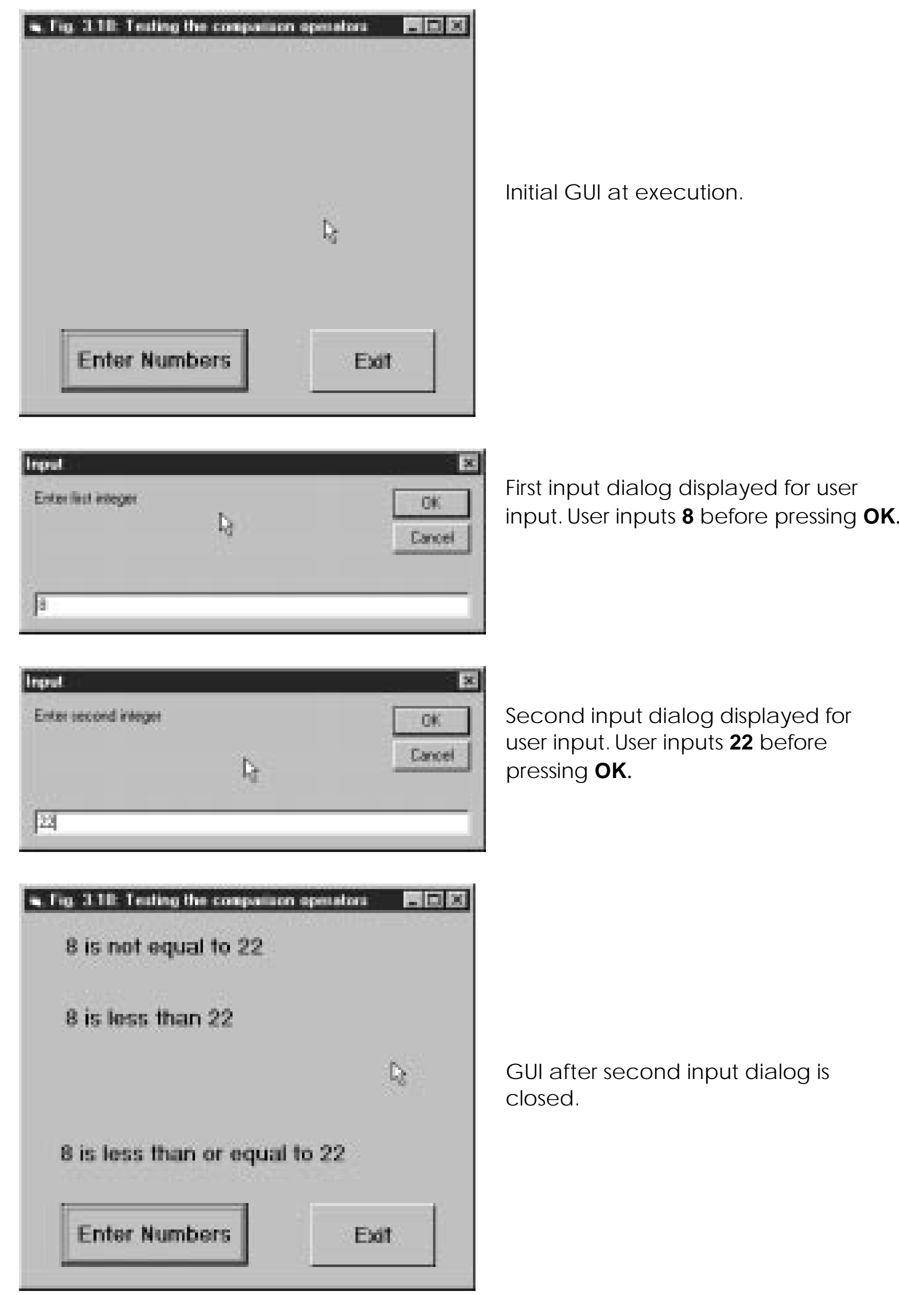


Fig. 3.18 GUI for program that compares two **`Integer`**s.

Object	Icon	Property	Property setting	Property description
form		**Name**	**frmIfThen**	Identifies the form.
		Caption	**Fig. 3.18: Testing the comparison operators**	Form title bar display.
Enter Numbers button		**Name**	**cmdEnterNumbers**	Identifies **Enter Numbers** button.
		Caption	**Enter Numbers**	Text that appears on button.
		Font	**MS Sans Serif bold 12** pt	Font for text on button's face.
Exit button		**Name**	**cmdExit**	Identifies **Exit** button.
		Caption	**Exit**	Text that appears on button.
		Font	**MS Sans Serif bold 12** pt	Font for text on button's face.
Label	A	**Name**	**lblDisplay1**	Identifies the **Label**.
		Caption	(empty)	Text **Label** displays.
		Font	**MS Sans Serif bold 12** pt	Font **Label** for **Label** display.
Label	A	**Name**	**lblDisplay2**	Identifies the **Label**.
		Caption	(empty)	Text **Label** displays.
		Font	**MS Sans Serif bold 12** pt	Font **Label** for **Label** display.
Label	A	**Name**	**lblDisplay3**	Identifies the **Label**.
		Caption	(empty)	Text **Label** displays.
		Font	**MS Sans Serif bold 12** pt	Font **Label** for **Label** display.
Label	A	**Name**	**lblDisplay4**	Identifies the **Label**.
		Caption	(empty)	Text **Label** displays.
		Font	**MS Sans Serif bold 12** pt	Font **Label** for **Label** display.

Fig. 3.19 Object properties for program that compares two **Integer**s.

```
1  ' Code listing for Fig. 3.18
2  ' Program compares two numbers
3  Option Explicit    ' Force explicit declarations
4
5  Private Sub cmdEnterNumbers_Click()
6     Dim num1 As Integer, num2 As Integer
7
8     ' Clear Labels
9     lblDisplay1.Caption = ""
10    lblDisplay2.Caption = ""
11    lblDisplay3.Caption = ""
12    lblDisplay4.Caption = ""
13
14    ' Get values from user
15    num1 = InputBox("Enter first integer", "Input")
16    num2 = InputBox("Enter second integer", "Input")
17
18    ' Test the relationships between the numbers
19    If num1 = num2 Then
20       lblDisplay1.Caption = num1 & " is equal to " & num2
21    End If
22
23    If num1 <> num2 Then
24       lblDisplay1.Caption = num1 & " is not equal to " & num2
25    End If
26
27    If num1 > num2 Then
28       lblDisplay2.Caption = num1 & " is greater than " & num2
29    End If
30
31    If num1 < num2 Then
32       lblDisplay2.Caption = num1 & " is less than " & num2
33    End If
34
35    If num1 >= num2 Then
36       lblDisplay3.Caption = num1 & _
37                             " is greater than or equal to " _
38                             & num2
39    End If
40
41    If num1 <= num2 Then
42       lblDisplay4.Caption = num1 & _
43                            " is less than or equal to " & num2
44    End If
45
46 End Sub
47
48 Private Sub cmdExit_Click()
49    End
50 End Sub
```

Fig. 3.20 Program that compares two **Integer**s.

The statement

```
Option Explicit
```

forces variables to be explicitly declared. The `Option Explicit` statement is always placed in the general declaration. `Option Explicit` can either be typed directly into the general declaration or placed there by Visual Basic when the ***Require Variable Declaration*** *checkbox* is checked. The ***Require Variable Declaration*** checkbox is on the **Options** dialog ***Editor*** *tab,* as shown in Fig. 3.21. The **Options** dialog is displayed when the **Tool** menu's **Options** menu item is selected. ***Require Variable Declaration*** is unchecked by default. Once checked, each new form associated with a project includes `Option Explicit` in the general declaration. Note: If ***Require Variable Declaration*** is unchecked and the form already exists, `Option Explicit` will not be added to the general declaration. The programmer must type it in the general declaration. However, each time a new form is created, `Option Explicit` is added by Visual Basic.

Testing and Debugging Tip 3.3

Force variable declarations by using `Option Explicit`.

Common Programming Error 3.9

If variable names are misspelled when not using `Option Explicit`, a misspelled variable name will be declared and initialized to zero, usually resulting in a run-time logic error.

Note that Fig. 3.21 labels a few **Editor** tab features relevant to our earlier discussion of **Full Module View** (Fig. 3.6). The user can also set the number of spaces that corresponds to a tab in the ***Tab Width*** `TextBox`.

Forces explicit declarations by adding the line `Option Explicit` to the general declaration (when checked)

Full Module View is the default when checked

When checked, displays **Procedure Separator** lines in **Full Module View**

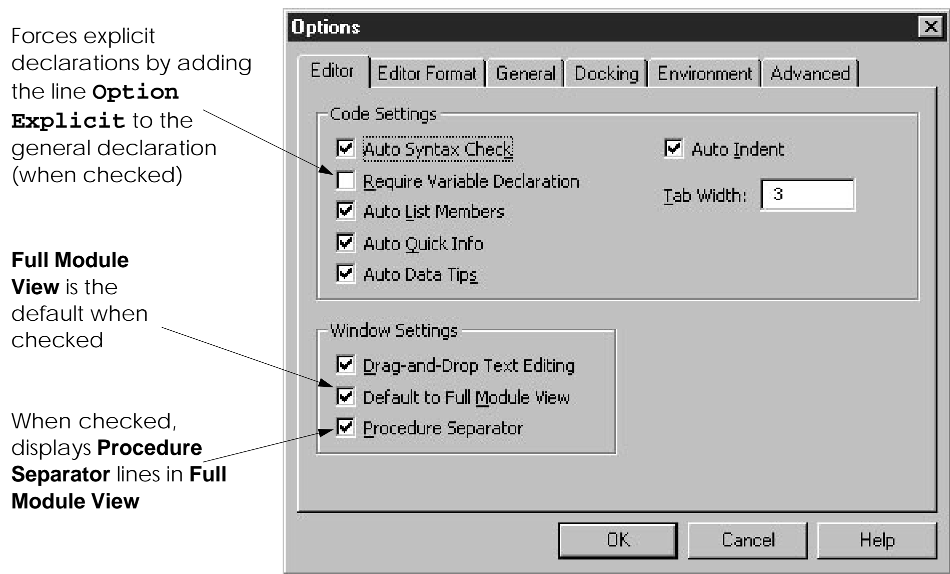


Fig. 3.21 **Options** window displaying **Editor** tab.

In procedure **cmdEnterNumbers_Click**, variables **num1** and **num2** are declared as **Integer**s. Variables can be declared just about anywhere in a procedure.Variables may be declared on separate lines or on a single line.

Good Programming Practice 3.12

If you prefer to place declarations at the beginning of a procedure, separate those declarations from executable statements in that procedure with one blank line to highlight where the declarations end and the executable statements begin.

Good Programming Practice 3.13

Always place a blank line before and after a group of declarations that appears between executable statements in the body of a procedure. This makes the declarations stand out in the program and contributes to program readability.

*Function **InputBox*** is used to get the values for **num1** and **num2** with the lines

```
num1 = InputBox("Enter first integer", "Input")
num2 = InputBox("Enter second integer", "Input")
```

Function **InputBox** displays an *input dialog,* which is shown in Fig. 3.22. The first argument (i.e., **"Enter first integer"**) is the prompt and the second argument (i.e., **"Input"**) determines what is displayed in the input dialog's title bar. When displayed, the dialog is *modal*—the user cannot interact with the form until the dialog is closed.

The input dialog contains a **Label**, two buttons and a **TextBox**. The **Label** displays the first argument passed to **InputBox**. The user clicks the **OK** button after entering a value in the **TextBox**. The **Cancel** button is pressed to cancel input. For this example, the values returned by successive calls to **InputBox** are assigned to **Integer**s **num1** and **num2**. The text representation of a number is implicitly converted (i.e., **"78"** is converted to **78**). If a value entered cannot be properly converted, a run-time error occurs. Pressing **Cancel** also creates a run-time error, because the empty string cannot be converted to an **Integer**. We discuss handling run-time errors in Chapter 13.

The line

```
If num1 = num2 Then
```

compares the contents of **num1** to the contents of **num2** for equality. If **num1** is equivalent to **num2**, the statement

Fig. 3.22 Dialog displayed by function **InputBox**.

```
lblDisplay1.Caption = num1 & " is equal to " & num2
```

is executed. The *string concatenation operator*, **&**, concatenates the implicitly converted values of **num1** and **num2** to strings. Keywords ***End If*** mark the end of the **If**/**Then** block. Since there is one statement in the body of the **If**/**Then**, the statement could be rewritten on a single line as

```
If num1 = num2 Then lblDisplay1 = num1 & " is equal to " & num2
```

End If is not required to terminate a single-line **If**/**Then**. We will use the **End If** convention throughout this book. If the condition is **False**, the next **If**/**Then** is tested. Note that in the above statement, we mentioned **lblDisplay1**, not **lblDisplay1.Caption**. Each control has a *default property* (a property that is used when only the control's **Name** is used). A **Label**'s default property is **Caption**.

Good Programming Practice 3.14

*Write each **If**/**Then** structure on multiple lines using the **End If** to terminate the condition. Indent the statement in the body of the **If**/**Then** structure to highlight the body of the structure and to enhance program readability.*

Good Programming Practice 3.15

Explicitly writing the default property improves program readability. Since default properties are different for most controls, omitting the property name can make the code more difficult to read.

Notice the use of spacing in Fig. 3.20. *White-space characters* such as tabs and spaces are normally ignored by the compiler (except when placed inside a set of double quotes). Statements may be split over several lines if the *line-continuation character,* _, is used (e.g., lines 36-38). A minimum of one white-space character must precede the line-continuation character.

Common Programming Error 3.10

Splitting a statement over several lines without the line-continuation character is a syntax error.

Common Programming Error 3.11

Not preceding the line-continuation character with at least one white-space character is a syntax error.

Common Programming Error 3.12

Placing anything, including comments, after a line-continuation character is a syntax error.

Several statements may be combined onto a single line by using a colon, **:**, between the statements. For example, the two statements

```
square = number ^ 2
cube = number ^ 3
```

could be combined on the single line

```
square = number ^ 2 : cube = number ^ 3
```

Statements can be spaced according to the programmer's preferences.

Common Programming Error 3.13

Splitting an identifier or a keyword is a syntax error.

Good Programming Practice 3.16

Even though Visual Basic provides the colon to combine multiple statements on a single line, writing only one statement per line improves program readability.

Summary

- With visual programming, the programmer has the ability to create graphical user interfaces (GUIs) by pointing and clicking with the mouse.
- Visual programming eliminates the need for the programmer to write code that generates the form, code for all the form's properties, code for form placement on the screen, code to create and place a **`Label`** on the form, code to change foreground and background colors, etc.
- The programmer creates the GUI and writes code to describe what happens when the user interacts (clicks, presses a key, double-clicks, etc.) with the GUI. These interactions, called events, are passed into the program by the Windows operating system.
- With event-driven programs, the user dictates the order of program execution.
- Event procedures are bodies of code that respond to events and are automatically generated by the IDE. All the programmer need do is locate them and add code to respond to the events. Only events relevant to a particular program need be coded.
- The **Properties** window contains the **Object box** that determines which object's properties are displayed. The **Object box** lists the form and all objects on the form. An object's properties are displayed in the **Properties** window when an object is clicked.
- Property **`TabIndex`** determines which control gets the focus (i.e., becomes the active control) when the Tab key is pressed at runtime. The control with a **`TabIndex`** value of **`0`** gets the initial focus. Pressing the Tab key at runtime transfers the focus to the control with a **`TabIndex`** of **`1`**.
- Pressing the **End** button terminates the program.
- Code is written in the **Code** window. The **Code** window is displayed by clicking the **Properties** window's **View Code** button.
- Visual Basic creates the event procedure name by appending the event type (**`Click`**) to the property **`Name`** (with an underscore _ added). **`Private Sub`** marks the beginning of the procedure. The **`End Sub`** statement marks the end of the procedure. Code the programmer wants executed is placed between the procedure definition header and the end of the procedure (i.e., **`End Sub`**).
- The **Object box** lists the form and all objects associated with the form. The **Procedure box** lists the procedures associated with the object displayed in the **Object box**.
- Programmers insert comments to document programs and improve program readability. Comments also help other people read and understand the program code. Comments do not cause the computer to perform any action when a program is run. A comment can begin with either **`'`** or **`Rem`** (for "remark") and is a single-line comment that terminates at the end of the current line.
- A program can print on the form using the **`Print`** method. Drawing directly on the form using **`Print`** is not the best way of displaying information, especially if the form contains controls because a control can hide text that is displayed with **`Print`**. This problem is solved by displaying the text in a control.
- The **`End`** statement terminates program execution (i.e., places the IDE in design mode).
- When a line of code is typed and Enter pressed, Visual Basic responds either by generating a syntax error (also called a compile error) or by changing the colors on the line.

- A syntax error is a violation of the language syntax (i.e., a statement is not written correctly). As a general rule, syntax errors tend to occur when statements are missing information, statements have extra information, names are misspelled, etc.
- If a statement does not generate syntax errors when the Enter key is pressed, a coloring scheme (called syntax-color highlighting) is imposed on the line of code. Comments are changed to green. The event procedure names remain black. Words recognized by Visual Basic are called keywords (also called reserved words) and appear blue.
- Keywords (i.e., **`Private`**, **`Sub`**, **`End`**, **`Print`**, etc.) cannot be used for anything other than for the feature they represent. Any improper use results in a syntax error. In addition to syntax color highlighting, Visual Basic may convert some lowercase letters to uppercase, and vice versa. The colors used for comments, keywords, etc. can be set using the **Editor Format** tab in the **Options** dialog (from the **Tools** menu).
- The **`TextBox`** control is the primary control for obtaining user input. **`TextBox`**es can also be used to display text.
- **`Text`** is the most commonly used **`TextBox`** property. The **`Text`** property stores the text for the **`TextBox`**. **`TextBox`**es have their **`Enabled`** property set to **`True`** by default. If the **`Enabled`** property is **`False`**, the user cannot interact with the **`TextBox`**.
- The **`MaxLength`** property value limits how many characters can be entered in a **`TextBox`**. The default value is **`0`**, which means that any number of characters can be input.
- Code that resides in the general declaration is available to every event procedure. The general declaration can be accessed with the **Code** window's **Object box**.
- A variable is a location in the computer's memory where a value can be stored for use by a program. A variable name is any valid identifier. Variable names cannot be keywords and must begin with a letter. The maximum length of a variable name is 255 characters containing only letters, numbers and underscores.
- Visual Basic is not case-sensitive—uppercase and lowercase letters are treated the same.
- Keyword **`Dim`** explicitly declares variables. Keyword **`As`** describes the variable's type (i.e., what type of information can be stored). **`Integer`** means that the variable holds **`Integer`** values (i.e., whole numbers such as 8, –22, 0, 31298). **`Integer`**s have a range of +32768 to –32767. **`Integer`** variables are initialized to **`0`** by default.
- Variables can also be declared special symbols called type-declaration characters such as the percent sign, **`%`**, for **`Integer`**. Not all types have type declaration characters.
- If a variable is not given a type when its declared, its type defaults to **`Variant`**. The **`Variant`** data type can hold any type of value (i.e., **`Integer`**s, **`Single`**s, etc.).
- When writing an assignment statement, the keyword **`Let`** is optional.
- The pair of double quotes, **`""`**, is called an empty string. Assigning an empty string to a **`TextBox`**'s **`Text`** property "clears" the **`TextBox`**.
- Variable names correspond to locations in the computer's memory. Every variable has a name, a type, a size and a value.
- Whenever a value is placed in memory, the value replaces the previous value in that location. Storing a value in a memory location is known as destructive read-in. When a value is read out of a memory location, the process is nondestructive.
- Caret (**`^`**) indicates exponentiation and asterisk (**`*`**) indicates multiplication.
- Most of the arithmetic operators are binary operators because they each operate on two operands.
- Visual Basic has separate operators for **`Integer`** and floating-point division. **`Integer`** division yields an **`Integer`** result. Fractional parts in **`Integer`** division are rounded before the division.

- Floating-point division yields a floating-point result (with a decimal point).
- The modulus operator, **Mod**, yields the **Integer** remainder after **Integer** division. Like the **Integer** division operator, the modulus operator rounds any fractional part before performing the operation. The expression **x Mod y** yields the remainder after **x** is divided by **y**. A remainder of 0 indicates that **y** divides evenly into **x**.
- The negation operator, **-**, changes the sign of a number from positive to negative (or a vice versa). The negation operator is a unary operator; it operates on one operand.
- Arithmetic expressions must be written in straight-line form.
- Parentheses are used in expressions much as in algebraic expressions.
- Parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer. Parentheses are said to be at the "highest level of precedence." Operators in the innermost pair of parentheses are applied first.
- As in algebra, it is acceptable to place extra parentheses in an expression to make the expression clearer. Unnecessary parentheses are also called redundant parentheses.
- The **If**/**Then** structure makes a decision based on the truth or falsity of some condition. If the condition is **True**, the statement in the body of the **If**/**Then** structure is executed. If the condition is **False**, the body statement is not executed.
- Conditions in **If**/**Then** structures can be formed by using the comparison operators.
- The **Option Explicit** statement forces variables to be explicitly declared. The **Option Explicit** statement is placed in the general declaration. **Option Explicit** can either be typed directly into the general declaration or placed there by Visual Basic when the **Require Variable Declaration** checkbox is checked.
- You can set the number of spaces that correspond to a tab in the **Tab Width** **TextBox**.
- Variables can be declared almost anywhere in a procedure.Variables may be declared on separate lines or on a single line.
- Function **InputBox** displays an input dialog. The first argument is the prompt and the second determines what is displayed in the input dialog's title bar. When displayed, the dialog is modal—the user cannot interact with the form until the dialog is closed.
- The ampersand operator, **&**, concatenates strings.
- Keywords **End If** mark the end of the **If**/**Then** block. **End If** is not required to terminate a single-line **If**/**Then**.
- Each control has a default property (a property that is used when only the control's **Name** is used). A **Label**'s default property is **Caption**.
- White-space characters such as tabs and spaces are normally ignored by the compiler.
- Statements may be split over several lines if the line-continuation character, _, is used. A minimum of one white-space character must precede the line-continuation character.
- Statements may be combined onto a line by using a colon, **:**, between the statements.
- It is incorrect to split identifiers and keywords.

Terminology

addition operator, **+**
arithmetic operators
As keyword
assignment operator, **=**
asterisk, *****
binary operator
button
Cancel button
caret, **^**
Code window

colon, **:**
comments
comparison operators
compile error
condition
default property
destructive read-in
Editor tab
Editor Format tab
embedded parentheses
empty string
Enabled property
End If
End keyword
End Sub
event
event-driven programming
event monitoring
event procedure
event type
explicit declaration
False keyword
floating-point number
focus
Full Module View
general declaration
identifier
If/**Then** structure
implicit declaration
InputBox function
Integer division operator, ****
Integer keyword
keyword
Let keyword
line-continuation character, _
Margin Indicator bar
MaxLength property
modal
modulus operator, **Mod**
negation operator, **-**
nested parentheses
nondestructive read-in
object
Object box
OK button
operand
operator
operator precedence
Option Explicit
Options dialog
percent sign, **%**
Print method
Procedure box
procedure definition header
Procedure Separator
Procedure View
Require Variable Declaration checkbox
Rem
reserved word
single-line comment
statement
string
string concatenation operator, **&**
Sub keyword
syntax color highlighting
syntax error
TabIndex property
Tab key
Tab Width TextBox
text
TextBox control
Text property
True keyword
type
type declaration character
unary operator
variable
Variant

Common Programming Errors

3.1 Using a keyword as a variable name is a syntax error.

3.2 Attempting to declare a variable name that does not begin with a letter is a syntax error.

3.3 Exceeding an **Integer**'s range is a run-time error.

3.4 Attempting to use a type declaration character and keyword **As** together is a syntax error.

3.5 It is an error to assume that the **As** clause in a declaration distributes to other variables on the same line. For example, writing the declaration **Dim x As Integer, y** and assuming that both **x** and **y** would be declared as **Integer**s would be incorrect, when in fact the declaration would declare **x** to be an **Integer** and **y** (by default) to be a **Variant**.

3.6 Expressions or values that cannot be implicitly converted result in run-time errors.

3.7 Reversing the order of the symbols in the operators `<>`, `>=` and `<=` as in `><`, `=>` and `=<`, respectively, are syntax errors.

3.8 Writing a statement such as `x = y = 0` and assuming that the variables `x` and `y` are both assigned zero, when in fact comparisons are taking place. This can lead to subtle logic errors.

3.9 If variable names are misspelled when not using `Option Explicit`, a misspelled variable name will be declared and initialized to zero, usually resulting in a run-time logic error.

3.10 Splitting a statement over several lines without the line-continuation character is a syntax error.

3.11 Not preceding the line-continuation character with at least one white-space character is a syntax error.

3.12 Placing anything, including comments, after a line-continuation character is a syntax error.

3.13 Splitting an identifier or a keyword is a syntax error.

Good Programming Practices

3.1 Prefix the name of `CommandButton`s with `cmd`. This allows easy identification of `CommandButton`s.

3.2 Comments written to the right of a statement should be preceded by several spaces to enhance program readability.

3.3 Visual Basic statements can be long. You might prefer to write comments above the line(s) of code you are documenting.

3.4 Precede comments that occupy a single line with a blank line. The blank line makes the comment stand out and improves program readability.

3.5 Indent statements inside the bodies of event procedures. We recommend three spaces of indentation. Indenting statements increases program readability.

3.6 Prefix the name of `TextBox`es with `txt` to allow easy identification of `TextBox`es.

3.7 Begin each identifier with a lowercase letter. This will allow you to distinguish between a valid identifier and a keyword.

3.8 Choosing meaningful variable names helps a program to be "self-documenting." A program becomes easier to understand simply by reading the code rather than having to read manuals or having to use excessive comments.

3.9 Explicitly declaring variables makes programs clearer.

3.10 Placing extra parentheses in an expression can make that expression clearer.

3.11 Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, use parentheses to force the order, exactly as you would do in algebraic expressions.

3.12 If you prefer to place declarations at the beginning of a procedure, separate those declarations from executable statements in that procedure with one blank line to highlight where the declarations end and the executable statements begin.

3.13 Always place a blank line before and after a group of declarations that appears between executable statements in the body of a procedure. This makes the declarations stand out in the program and contributes to program readability.

3.14 Write each `If`/`Then` structure on multiple lines using the `End If` to terminate the condition. Indent the statement in the body of the `If`/`Then` structure to highlight the body of the structure and to enhance program readability.

3.15 Explicitly writing the default property improves program readability. Since default properties are different for most controls, omitting the property name can make the code more difficult to read.

3.16 Even though Visual Basic provides the colon to combine multiple statements on a single line, writing only one statement per line improves program readability.

Testing and Debugging Tips

3.1 As Visual Basic processes the line you typed, it may find one or more syntax errors. Visual Basic will display an error message indicating what the problem is and where on the line the problem is occurring.

3.2 Syntax color highlighting helps the programmer avoid using keywords accidentally.

3.3 Force variable declarations by using **`Option Explicit`**.

Software Engineering Observation

3.1 Even though multiple **`End`** statements are permitted, use only one. Normal program termination should occur in only one place.

Self-Review Exercises

3.1 Fill in the blanks in each of the following:

a) Keywords ________ begin the body of an event procedure and keywords ________ end the body of an event procedure.
b) When a value is placed into a memory location, it is known as __________ read-in.
c) What arithmetic operation(s) is/are on the same level of precedence as multiplication? _____
d) When parentheses are nested in an arithmetic expression, which set of parentheses is evaluated first? __________
e) A location in a computer's memory that may contain different values at various times throughout program execution is called a __________.
f) By default, **`Integer`** variables are initialized to the value __________.

3.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) A comment's text is printed on the form as the comment is executed.
b) The **`Rem`** statement stores a string in the Visual Basic variable **`Remark`**.
c) **`Option Explicit`** forces explicit variable declaration.
d) All variables, when declared explicitly, must be given a data type either by using the **`As`** keyword or by using a type-declaration character (if the data type has one).
e) The variables **`number`** and **`NuMbEr`** are identical.
f) Declarations can appear almost anywhere in the body of an event procedure.
g) The modulus operator, **`Mod`**, can be used only with **`Integer`** operands. Attempts to use floating-point numbers (e.g., 19.88, 801.93, 3.14159, etc.) are syntax errors.
h) The arithmetic operators **`*`**, **`/`** and **`\`** all have the same level of precedence.
i) Visual Basic syntax always requires arithmetic expressions to be enclosed in parentheses—otherwise, syntax errors occur.

3.3 Fill in the blanks in each of the following:

a) The __________ property limits the number characters input in a **`TextBox`**.
b) The default data type is __________.
c) The __________ character is the symbol for the string concatenation operator.
d) When a value is read out of a memory location, it is known as __________ readout.

3.4 Write a single statement to accomplish each of the following:

a) Explicitly declare the variables **`cj`**, **`ventor`** and **`num`** to be of type **`Integer`**.
b) Assign "**`Hello!`**" to the **`Label lblGreeting`**.
c) Combine the following three lines into a single line:

```
' Initialization
total% = 0
counter% = 1
```

d) Assign the sum of **x**, **y** and **z** to the variable **sum**. Assume that each variable is of type **Integer**.
e) Decrement the variable **count** by 1, then subtract it from the variable **total**, and assign the result to the variable **u**. Assume all variables to be of type **Integer**.
f) Assign the product of the **Integer** variables **r**, **i**, **m**, **e** and **s** to the variable **g**.
g) Calculate the remainder after **total** is divided by **counter** and assign the result to **remainder**. Assume the variables to be of type **Integer**.
h) Assign the value returned from function **InputBox** to the variable **userInput**. The function **InputBox** should display the message "**Enter your data**." The **InputBox**'s title bar should display "**Data Input**." Assume the variable **userInput** to be of type **Integer**.

3.5 Write a statement or comment to accomplish each of the following:
a) State that a program will calculate the product of three **Integer**s.
b) Print the message "**printing to the form**" on the form using the **Print** method.
c) Force variable declarations.
d) Compute the **Integer** average of the three **Integer**s contained in variables **x**, **y** and **z**, and assign the result to the **Integer** variable **result**.
e) Print on the form "**The product is**" followed by the value of the **Integer** variable **result**.
f) Compare the **Integer** variables **sum1** and **sum2** for equality. If the result is true, set the **Integer** variable **flag** to 76.

3.6 Identify and correct the error(s) in each of the following statements:
a) `Dim False As Integer`
b) `Dim variable, inputValue As Integers`
c) `Integer oscii    Rem  declare variable`
d) `a + b = c     ' add a, b and assign result to c`
e) `d = t Modulus r + 50`
f) `variable = -65800     ' variable is of type Integer`
g) `" Change BackColor property's value`
h)
```
If (x > y)
   frmMyForm.Print x
```
i) `Dim triplett As Integer, picks As Integer, End As Integer`
j) `triplett = picks = 10    ' Initialize both variables to 10`
k) `x : y = oldValue    Rem assign oldValue to both x and y`

3.7 Given the equation $b = 8e^5 - n$, which of the following, if any, are correct statements for this equation?
a) `b = 8 * e ^ 5 - n`
b) `b = ( 8 * e ) ^ 5 - n`
c) `b = 8 * ( e ^ 5 ) - n`
d) `b = 8 * e ^ ( 5 - n )`
e) `b = ( 8 * e ) ^ ( ( 5 ) - n )`
f) `b = 8 * e * e ^ 4 - n`

3.8 State the order of evaluation of the operators in each of the following statements, and show the value of **m** after each statement is performed. Assume **m** to be an **Integer** variable.
a) `m = 7 + 3 * 6 \ 2 - 1`
b) `m = 2 Mod 2 + 2 * 2 - 2 / 2`
c) `m = 8 + 10 \ 2 * 5 - 16 \ 2`
d) `m = -5 - 8 Mod 4 + 7 * (2 ^ 2 + 2)`
e) `m = 10 Mod 3 ^ 1 ^ 2 - 8`

Answers to Self-Review Exercises

3.1 a) **Sub**, **End Sub**. b) destructive. c) floating-point division (**/**). d) innermost. e) variable. f) zero.

3.2
a) False. Comments are not executable statements; nothing is printed.
b) False. **Rem** is simply another way of writing a comment.
c) True.
d) False. If a variable is not explicitly given a type, then it is given the default data type of **Variant**.
e) True. Visual Basic is not case-sensitive.
f) True.
g) False. Floating-point numbers are rounded to **Integer**s before **Mod** is performed.
h) False. Multiplication (*****) and floating-point division (**/**) have the same precedence. **Integer** division (****) has a lower precedence.
i) False. Visual Basic does not require all expressions to use parentheses.

3.3 a) **MaxLength**. b) **Variant**. c) ampersand, **&**. d) nondestructive.

3.4
a) `Dim cj As Integer, ventor As Integer, num As Integer`
b) `lblGreeting.Caption = "Hello!"`
c) `total% = 0 : counter% = 1     ' Initialization`
d) `sum = x + y + z`
e) `u = total - (count - 1)`
f) `g = r * i * m * e * s`
g) `remainder = total Mod counter`
h) `userInput = InputBox("Enter your data", "Data Input")`

3.5
a) `' This program will calculate the product of three integers`
b) `Print "printing to the form"`
c) `Option Explicit   ' In general declaration`
d) `result = (x + y + z) / 3`
e) `Print "The product is " & result`
f)
```
If sum1 = sum2 Then
   flag = 76
End If
```

3.6 a) **False** is a keyword and may not be used as an identifier. Use a non-keyword as the variable name.
b) **Integers** should be **Integer**.
c) A variable cannot be declared this way. Correction: **Dim oscii As Integer**.
d) The variable storing the result of the assignment (**c**) must be the left operand of the assignment operator. The statement should be rewritten as **c = a + b**.
e) **Modulus** should be **Mod**.
f) The number –65800 is out of range for an **Integer**. The value being assigned should be in the range –32,768 to 32,767.
g) The double quotes should be single quotes or **Rem** to form a comment.
h) The **Then** keyword is missing and the statement should either be contained on one line or be terminated by **End If**.
i) **End** is a keyword and may not be used as an identifier.
j) A comparison is being made rather than an assignment. Each assignment should be done separately.
```
   triplett = 10
   picks = 10
```

k) Invalid syntax. Each assignment must be done separately.

```
x = oldValue
y = oldValue
```

3.7 a, c, f.

3.8 a)
```
m = 7 + 3 * 6 \ 2 - 1
m = 7 + 18 \ 2 - 1
m = 7 + 9 - 1
m = 16 - 1
m = 15
```
b)
```
m = 2 Mod 2 + 2 * 2 - 2 / 2
m = 2 Mod 2 + 4 - 2 / 2
m = 2 Mod 2 + 4 - 1
m = 0 + 4 - 1
m = 4 - 1
m = 3
```
c)
```
m = 8 + 10 \ 2 * 5 - 16 \ 2
m = 8 + 10 \ 10 - 16 \ 2
m = 8 + 1 - 16 \ 2
m = 8 + 1 - 8
m = 9 - 8
m = 1
```
d)
```
m = -5 - 8 Mod 4 + 7 * (2 ^ 2 + 2)
m = -5 - 8 Mod 4 + 7 * (4 + 2)
m = -5 - 8 Mod 4 + 7 * 6
m = -5 - 8 Mod 4 + 42
m = -5 - 0 + 42
m = -5 + 42
m = 37
```
e)
```
m = 10 Mod 3 ^ 1 ^ 2 - 8
m = 10 Mod 3 ^ 1 - 8
m = 10 Mod 3 - 8
m = 1 - 8
m = -7
```

Exercises

3.9 Identify and correct the error(s) in each of the following statements:

a) Assume that `Option Explicit` has been set.

```
' Event code for procedure
Private Sub cmdDisplay_Click()
   value1 = 5 : value2 = 10

   If value1 > value2 Then
      Print value1
   End If
End Sub
```

b) Assume that `Option Explicit` has not been set.

```
' Event code for procedure
Private Sub lblGreeting_Click()
   Low1Val = 8

   ' Display the value in lblGreeting's Caption property
   lblGreeting = Low1Val
End Sub
```

c) **animalName = "Giant " Cat "Parrot"   ' Concatenate strings**
d) **thisIsAnIncrediblyLongVariableNameOf45Letters As Integer**
e) Assume that the **Integer** variables **c** and **j** are declared and initialized to **47** and **55**, respectively.

```
Dim x As Integer

If c =< j Then
   x = 79
   frmMyForm.Print x
End If
```

f) Assume that the variables **q**, **pcm** and **qp** are declared as **Integer**s.

```
' Executable statement
q = 76 ; qp =   ' Hard return after =
78 ; pcm = 61
```

3.10 Write a single statement or line that accomplishes each of the following:
a) Print the message **"Visual Basic 6!!!!"** on the form.
b) Assign the product of variables **width22** and **height88** to variable **area51**.
c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).
d) Calculate the area of a circle and assign it to the **Integer** variable **circleArea**. Use the formula $area = (\pi r^2)$, the variable **radius** and the value 3.14159 for π.
e) Concatenate the following two strings using the string concatenation operator and assign the result to **Label lblHoliday**'s **Caption**: **"Merry Christmas"** and **" and a Happy New Year"**.

3.11 Fill in the blanks in each of the following:
a) __________ are used to document a program and improve its readability.
b) A statement that makes a decision is __________.
c) Calculations are normally performed by __________ statements.
d) The __________ statement terminates program execution.
e) The __________ method is used to display information to the form.
f) A __________ is a message to the user indicating that some action is required.

3.12 State which of the following are *true* and which are *false*. If *false*, explain why.
a) **Integer** division has the same precedence as floating-point division.
b) The following are all valid variable names: **_under_bar_**, **m928134**, **majestic12**, **her_sales**, **hisAccountTotal**, **cmdWrite**, **b**, **creditCardBalance1999**, **YEAR_TO_DATE**, **__VoLs__LiSt__**.
c) The statement **squareArea = side ^ 2** is a typical example of an assignment statement.

d) A valid arithmetic expression with no parentheses is evaluated from left to right regardless of the operators used in that expression.
e) The following are all invalid variable names: **2quarts**, **1988**, **&67h2**, **vols88**, ***true_or_FALSE**, **99_DEGREES**, **_this**, **Then**.
f) Visual Basic automatically generates the beginning and end code of event procedures.

3.13 Given the following declarations, list the type for each variable declared.

a) `Dim traveler88 As Integer`
b) `number% = 76`
c) `Dim cars As Integer, trucks`
d) `Dim touchDowns, fieldGoals As Integer`
e) `portNumber = 80    ' Implicit declaration`

3.14 Given the equation $y = ax^3 + 7$, which of the following, if any, are correct statements for this equation?

a) `y = a * ( x ^ 3 + 7 )`
b) `y = ( a * x ) ^ 3 ) + 7`
c) `y = ( a * x * x * x + 7 )`
d) `y = ( a * ( x * ( x * x ) ) + 7 )`
e) `y = ( a * ( x * x ) ^ 2 ) + 7`
f) `y = (a) * (x) * (x) * (x) + (7)`

3.15 State the order of evaluation of the operators in each of the following statements, and show the value of **x** after each statement is performed. Assume **x** to be an **Integer** variable.

a) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) )`
b) `x = 1 + 2 * 3 - 4 / 4 - 12 \ 6 * 6`
c) `x = ( ( 10 - 4 * 2 ) \ 2 + ( 13 - 2 * 5 ) ) ^ 2`
d) `x = 8.2 Mod 3 + 2 / 2 - -3`
e) `x = -2 + 7.4 \ 5 - 6 / 4 Mod 2`

3.16 Which, if any, of the following statements contain variables involved in destructive read-in?

a) `myVariable = txtTextBox.Text`
b) `V = O + L + S + 8 * 8`
c) `Print "Destructive read-in"`
d) `Print "a = 8"`
e) `Print x = 22`
f) `Print userName`

3.17 What, if anything, prints when each of the following statements is performed? If nothing prints, then answer "nothing." Assume that **x = 2** and **y = 3**.

a) `Print x`
b) `Print -y ^ 2`
c) `Print x + x`
d) `Print "x ="`
e) `txtTextBox.Text = "x + y"`
f) `z = x + y`
g) `Print x + y * 4 ^ 2 / 4 & " is the magic number!"`

3.18 Write a program that inputs three different **Integer**s using function **InputBox** and prints the sum, the average, the product, the smallest and the largest of these numbers on the form using **Print**. Use only the single-selection version of the **If**/**Then** statement you learned in this chapter.

Provide an **Exit** button to terminate program execution. (Hint: Each **`Print`** statement is similar to **`Print "Sum is "; sum`**. The semicolon (**`;`**) instructs Visual Basic to print the variable's value immediately after the last character printed.)

3.19 Write a program that reads in the radius of a circle as an **`Integer`** and prints the circle's diameter, circumference and area to the form using the **`Print`** method. Do each of these calculations inside a **`Print`** statement. Use the following formulas (r is the radius): $diameter = 2r$, $circumference = 2\pi r$, $area = \pi r^2$. Use the value 3.14159 for π. (Note: In this chapter, we have discussed only **`Integer`** variables. In Chapter 4 we will discuss floating-point numbers (i.e., values that can have decimal points and data type **`Single`**).

3.20 Enhance Exercise 3.19 by displaying the diameter, circumference and area in **`Label`**s.

3.21 Write a temperature conversion program that converts a Fahrenheit temperature to a Celsius temperature. Provide a **`TextBox`** for user input and a **`Label`** for displaying the converted temperature. Provide a **Input** button to read the value from the **`TextBox`**. Also provide the user with an **Exit** button to end program execution. Use the following formula: *Celsius = 5 / 9 x (Fahrenheit – 32).*

3.22 Enhance Exercise 3.21 to provide a conversion from Fahrenheit to Kelvin. Display the converted Kelvin temperature in a second **`Label`**. Use the formula: *Kelvin = Celsius + 273.*

3.23 Modify Exercise 3.21 to use function **`InputBox`** for input.