

3



Variables, Input, and Output

3.1 Numbers 54

- ◆ Arithmetic Operations ◆ Variables ◆ Augmented Assignments
- ◆ Mathematical Functions ◆ The Integer Data Type
- ◆ Two Other Integer Operators ◆ The Decimal Data Type
- ◆ Multiple Declarations ◆ Parentheses, Order of Precedence
- ◆ Three Kinds of Errors ◆ The Error List Window

3.2 Strings 72

- ◆ Variables and Strings ◆ Using Text Boxes for Input and Output
- ◆ Option Explicit and Option Strict ◆ Concatenation
- ◆ String Properties and Methods ◆ Indices and Substrings ◆ The Empty String
- ◆ Initial Value of a String Variable ◆ Widening and Narrowing
- ◆ Data Types of Literals and Expressions ◆ Internal Documentation
- ◆ Line Continuation
- ◆ Scope of a Variable ◆ Auto Correction

3.3 Input and Output 93

- ◆ Formatting Numeric Output ◆ Dates as Input and Output
- ◆ Using a Masked Text Box for Input ◆ Getting Input from an Input Dialog Box
- ◆ Using a Message Dialog Box for Output ◆ Named Constants
- ◆ Formatting Output with Zones

Summary 109

Programming Projects 111

3.1 Numbers

Much of the data processed by computers consist of numbers. In programming terms, numbers are called **numeric literals**. This section discusses the operations that are performed with numbers and the ways numbers are displayed.

Arithmetic Operations

The five standard arithmetic operations in Visual Basic are addition, subtraction, multiplication, division, and exponentiation. Addition, subtraction, and division are denoted in Visual Basic by the standard symbols $+$, $-$, and $/$, respectively. However, the notations for multiplication and exponentiation differ from the customary mathematical notations.

Mathematical Notation	Meaning	Visual Basic Notation
$a \cdot b$ or $a \times b$	a times b	$a * b$
a^r	a to the r^{th} power	$a ^ r$

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

```
lstBox.Items.Add(n)
```

displays the number n as the last item in the list box. *Add* is called a **method**. (Generally, a method is a process that performs a task for a particular object.) If the parentheses contain a combination of numbers and arithmetic operations, the operations are carried out and then the *Add* method displays the result. Another important method is *Clear*. The statement

```
lstBox.Items.Clear()
```

removes all the items displayed in the list box.

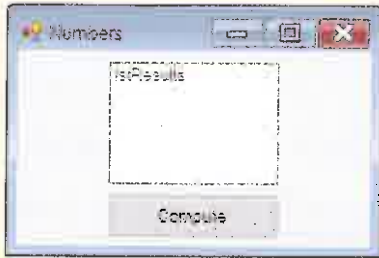


Example 1

Arithmetic Operations The following program applies each of the five arithmetic operations. Preceding the program are the form design and a table showing the names of the objects on the form and the altered settings, if any, for properties of these objects. This form design is also used in the discussion and examples in the remainder of this section.

The word “Run” in the phrasing [Run . . .] indicates that the *Start* button or F5 should be pressed to execute the program. Notice that in the output $3 / 2$ is displayed in decimal form. Visual Basic never displays numbers as fractions. In the evaluation of $2 * (3 + 4)$, the operation inside the parentheses is calculated first.

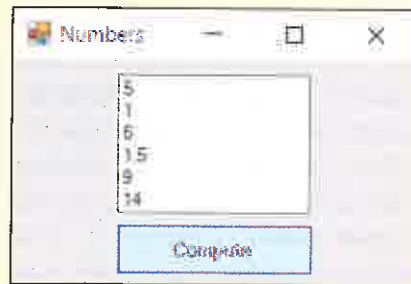
Note: All programs appearing in examples and case studies are provided on the companion website for this book. See the discussion in the Preface for details.



OBJECT	PROPERTY	SETTING
frmArithmetic	Text	Numbers
lstResults		
btnCompute	Text	Compute

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    lstResults.Items.Clear()
    lstResults.Items.Add(3 + 2)
    lstResults.Items.Add(3 - 2)
    lstResults.Items.Add(3 * 2)
    lstResults.Items.Add(3 / 2)
    lstResults.Items.Add(3 ^ 2)
    lstResults.Items.Add(2 * (3 + 4))
End Sub
```

[Run, and then click on the button.]



Variables

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

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

Here’s how this problem would be solved with a computer program:

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim speed As Double
    Dim timeElapsed As Double
    Dim distance As Double
    lstResults.Items.Clear()
    speed = 50
    timeElapsed = 14
```

```

distance = speed * timeElapsed
lstResults.Items.Add(distance)
distance = 410
timeElapsed = distance / speed
lstResults.Items.Add(timeElapsed)

```

End Sub

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

700

8.2

Skip the three lines beginning with Dim for now. We will return to them soon. The sixth line sets the speed to 50, and the seventh line sets the time elapsed to 14. The eighth line multiplies the value for the speed by the value for the time elapsed and sets the distance to that product. The next line displays the answer to the distance-traveled question. The three lines before the End Sub statement answer the time-required question in a similar manner.

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

A variable is an object used to store a value. The value assigned to the variable may change during the execution of the program. In Visual Basic, variable names must begin with a letter or an underscore, and can consist only of letters, digits, and underscores. (The shortest variable names consist of a single letter.) Visual Basic does not distinguish between uppercase and lowercase letters used in variable names. Some examples of variable names are *total*, *numberOfCars*, *taxRate_2016*, and *n*. As a convention, we write variable names in lowercase letters except for the first letters of each additional word (as in *gradeOnFirstExam*). This convention is called **camel casing**. Descriptive variable names such as *distance* and *timeElapsed* help others (and you at a later time) easily recall what the variable represents.

If *var* is a variable and *n* is a numeric literal, then the statement

```
var = n
```

assigns the number *n* to the variable *var*. Such a statement is another example of an assignment statement.

A variable is declared to be of a certain type depending on the sort of data that can be assigned to it. The most versatile type for holding numbers is called **Double**. A variable of type Double can hold whole, fractional, or mixed numbers between about $-1.8 \cdot 10^{308}$ and $1.8 \cdot 10^{308}$. Dim statements (also called **declaration statements**) declare the names and types of the variables to be used in the program. The second, third, and fourth lines of the preceding event procedure declare three variables of type Double and give them the names *speed*, *timeElapsed*, and *distance*. Variables must be declared before values can be assigned to them.

In general, a statement of the form

```
Dim varName As Double
```

declares a variable named *varName* to be of type Double. Actually, the Dim statement causes the computer to set aside a location in memory referenced by *varName*. You might think of the word Dim as meaning “make some space in memory for.” The data type specifies the kind of value that will be placed into the space.

Since *varName* is a numeric variable, the Dim statement initially places the number zero in that memory location. (We say that zero is the **initial value** or **default value** of the variable.) Each subsequent assignment statement having *varName* to the left of the equal sign will change the value of the number.



VideoNote
Numbers &
Strings

The initial value can be set to a value other than zero. To specify a nonzero initial value, follow the declaration statement with an equal sign followed by the initial value. The statement

```
Dim varName As Double = 50
```

declares the specified variable as a variable of type Double and gives it the initial value 50. The statement

```
lstBox.Items.Add(varName)
```

looks into this memory location for the current value of the variable and adds that value to the list box.

IntelliSense provides assistance with both declaration and assignment statements. Consider the pair of statements

```
Dim interestRate As Double
interestRate = 0.05
```

In the first statement, IntelliSense will suggest the word “As” after you type “Dim interestRate”, and will suggest the word “Double” after you type “Dou”. In the second statement, IntelliSense will suggest the word “interestRate” after you type “inte”. Thus, IntelliSense both speeds up the writing of programs and helps protect against spelling errors.

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



Example 2

Evaluate Expressions

The following program displays the default value of a variable and the value of an expression:

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim a As Double
    Dim b As Double = 3
    lstResults.Items.Clear()
    lstResults.Items.Add(a)
    lstResults.Items.Add(b)
    a = 5
    lstResults.Items.Add(a * (2 + b))
End Sub
```

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

```
0
3
25
```

If *var* is a variable, then the assignment statement

```
var = expression
```

first evaluates the expression on the right and then assigns its value to the variable on the left. For instance, the event procedure in Example 2 can be written as

```

Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim a As Double
    Dim b As Double = 3
    Dim c As Double
    lstResults.Items.Clear()
    lstResults.Items.Add(a)
    lstResults.Items.Add(b)
    a = 5
    c = a * (2 + b)
    lstResults.Items.Add(c)
End Sub

```

The expression $a * (2 + b)$ is evaluated to 25, and then this value is assigned to the variable *c*.

Augmented Assignments

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

```
var = var + 1
```

is meaningful. It first evaluates the expression on the right (that is, it adds 1 to the value of the variable *var*) and then assigns this sum to the variable *var*. The effect is to increase the value of the variable *var* by 1. In terms of memory locations, the statement retrieves the value of *var* from *var*'s memory location, uses it to compute $var + 1$, and then places the sum back into *var*'s memory location. This type of calculation is so common that Visual Basic provides a special operator to carry it out. The statement `var = var + 1` can be replaced with the statement

```
var += 1
```

In general, if *n* has a numeric value, then the statement

```
var += n
```

adds the value of *n* to the value of *var*. The operator `+=` is said to perform an **augmented assignment**. Some other augmented assignment operators are `-=`, `*=`, `/=`, and `^=`.

Mathematical Functions

There are several common operations that we often perform on numbers other than the standard arithmetic operations. For instance, we may take the square root of a number or round a number. These operations are performed by built-in functions. Functions associate with one or more values, called the *input*, a single value called the *output*. The function is said to **return** the output value. The three functions considered here have numeric input and output.

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

EXPRESSION	VALUE	EXPRESSION	VALUE	EXPRESSION	VALUE
<code>Math.Sqrt(9)</code>	3	<code>Int(2.7)</code>	2	<code>Math.Round(2.7)</code>	3
<code>Math.Sqrt(0)</code>	0	<code>Int(3)</code>	3	<code>Math.Round(2.317, 2)</code>	2.32
<code>Math.Sqrt(6.25)</code>	2.5	<code>Int(-2.7)</code>	-3	<code>Math.Round(2.317, 1)</code>	2.3

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



Example 3 Evaluate Mathematics Functions The following program evaluates each of the functions for a specific input given by the value of the variable *n*:

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim n As Double
    Dim root As Double
    n = 6.76
    root = Math.Sqrt(n)
    lstResults.Items.Clear()
    lstResults.Items.Add(root)
    lstResults.Items.Add(Int(n))
    lstResults.Items.Add(Math.Round(n, 1))
End Sub
```

[Run, and then click on the *Compute* button. The following is displayed in the list box.]

2.6
6
6.8



Example 4 Evaluate Mathematics Functions The following program evaluates each of the preceding functions at an expression:

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim a As Double
    Dim b As Double
    a = 2
    b = 3
    lstResults.Items.Clear()
    lstResults.Items.Add(Math.Sqrt((5 * b) + 1))
    lstResults.Items.Add(Int((a ^ b) + 0.8))
    lstResults.Items.Add(Math.Round(a / b, 3))
End Sub
```

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

4
8
0.667

The Integer Data Type

In this text, we sometimes need to use variables of type **Integer**. An integer variable is declared with a statement of the form

```
Dim varName As Integer
```

and can be assigned only whole numbers from about -2 billion to 2 billion. Integer variables are commonly used for counting.

Two Other Integer Operators

In addition to the five arithmetic operators discussed at the beginning of this section, the *Mod* operator and the *integer division* operator (`\`) are also available in Visual Basic. Let m and n be positive whole numbers. When you use long division to divide m by n , you obtain an integer quotient and an integer remainder. In Visual Basic, the integer quotient is denoted $m \backslash n$ and the integer remainder is denoted $m \text{ Mod } n$. For instance,

$$\begin{array}{r} 4 \leftarrow 14 \backslash 3 \\ 3 \overline{)14} \\ \underline{12} \\ 2 \leftarrow 14 \text{ Mod } 3 \end{array}$$

Essentially, $m \backslash n$ divides two numbers and chops off the fraction part, and $m \text{ Mod } n$ is the remainder when m is divided by n . Some examples are as follows.

EXPRESSION	VALUE	EXPRESSION	VALUE
$19 \backslash 5$	3	$19 \text{ Mod } 5$	4
$10 \backslash 2$	5	$10 \text{ Mod } 2$	0
$5 \backslash 7$	0	$5 \text{ Mod } 7$	5



Example 5

Convert Lengths The following program converts 41 inches to 3 feet and 5 inches:

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim totalInches As Integer
    Dim feet As Integer
    Dim inches As Integer
    totalInches = 41
    feet = totalInches \ 12
    inches = totalInches Mod 12
    lstResults.Items.Add(feet)
    lstResults.Items.Add(inches)
End Sub
```

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

3
5

Note: You can think of integer division as throwing away the fractional part of an ordinary division.

The Decimal Data Type

The **Double** data type provides the largest and smallest possible magnitudes for a number. However, the **Double** data type is subject to rounding errors. The rounding errors are slight and inconsequential except in applications that manipulate dollar amounts.

The **Decimal** data type uses a different format for storing numbers than the **Double** data type. The **Decimal** data type requires more memory space than the **Double** data type and limits the magnitude of numbers that can be handled. However, the **Decimal** data type is not subject to rounding errors. In this textbook, we use the **Decimal** data type for all financial calculations.

A Decimal variable is declared with a statement of the form

```
Dim varName As Decimal
```

and can be assigned numbers with up to 29 significant digits. For reasons that will be explained in the next section, when a literal number containing a decimal point is assigned to a variable of type Decimal, the letter D must be appended to the end of the literal number.



Example 6

Calculate a New Balance

The following program calculates the balance in a savings account after one year when \$1,025.45 is deposited at 4% interest compounded annually:

```
Private Sub btnCalculate_Click(...) Handles btnCalculate.Click
    Dim balance As Decimal = 1025.45D
    Dim interestRate As Decimal = 0.04D
    balance = balance + (interestRate * balance)
    lstResult.Items.Add(Math.Round(balance, 2))
End Sub
```

[Run, and then click on the button. The following is displayed in the list box.]
1066.47

Multiple Declarations

Several variables of the same type can be declared with a single Dim statement. For instance, the two Dim statements in Example 4 can be replaced by the single statement

```
Dim a, b As Double
```

Two other types of multiple-declaration statement are

```
Dim a As Double, b As Integer, c As Decimal
Dim a As Double = 2, b As Integer = 5, c As Decimal = 1.5D
```

Parentheses, Order of Precedence

Parentheses should be used to clarify the meaning of a numeric expression. When there are insufficient parentheses, the arithmetic operations are performed in the following order of precedence:

1. terms inside parentheses (inner to outer)
2. exponentiation
3. multiplication, division (ordinary and integer), and modulus
4. addition and subtraction.

In the event of a tie, the leftmost operation is performed first. For instance, $8 / 2 * 3$ is evaluated as $(8 / 2) * 3$.

A good programming practice is to use parentheses liberally so that you *never* have to remember the order of precedence. For instance, write $(2 * 3) + 4$ instead of $2 * 3 + 4$ and write $(2 ^ 3) + 4$ instead of $2 ^ 3 + 4$.

Parentheses cannot be used to indicate multiplication, as is commonly done in algebra. For instance, the expression $x(y + z)$ is not valid. It must be written as $x*(y + z)$.

Three Kinds of Errors

Grammatical and punctuation errors are called **syntax errors**. Most syntax errors are spotted by the Code Editor when they are entered. The editor underlines the syntax error with a wavy blue line and displays a description of the error when the mouse cursor is hovered over the wavy line. Some incorrect statements and their errors are shown in Table 3.1.

TABLE 3.1 Three syntax errors.

Statement	Reason for Error
<code>lstBox.Itms.Add(3)</code>	The word "Items" is misspelled.
<code>lstBox.Items.Add(2 +)</code>	The number following the plus sign is missing.
<code>Dim m; n As Integer</code>	The semicolon should be a comma.

Errors that occur while a program is running are called **runtime errors** or **exceptions**. (Exceptions are said to be **thrown** by Visual Basic.) They usually occur because something outside the program does not behave as expected. For instance, if the file `Data.txt` is not in the root folder of the C drive, then a statement that refers to the file by the filespec `"C:\Data.txt"` will cause the program to stop executing and produce a message box with the title

FileNotFoundException was unhandled.

Also, a yellow arrow will appear at the left side of the line of code that caused the error. At that point, you should end the program.

A third type of error is called a **logic error**. Such an error occurs when a program does not perform the way it was intended. For instance, the statement

```
average = firstNum + secondNum / 2
```

is syntactically correct. However, an incorrect value will be generated, since the correct way to calculate the average is

```
average = (firstNum + secondNum) / 2
```

Logic errors are the most difficult type to find. Appendix D discusses debugging tools that can be used to detect and correct logic errors.

The Error List Window

Syntax errors are not only indicated in the Code Editor, but also are listed in the Error List window. **Note:** If the window is not visible, click on *Error List* in the View menu.

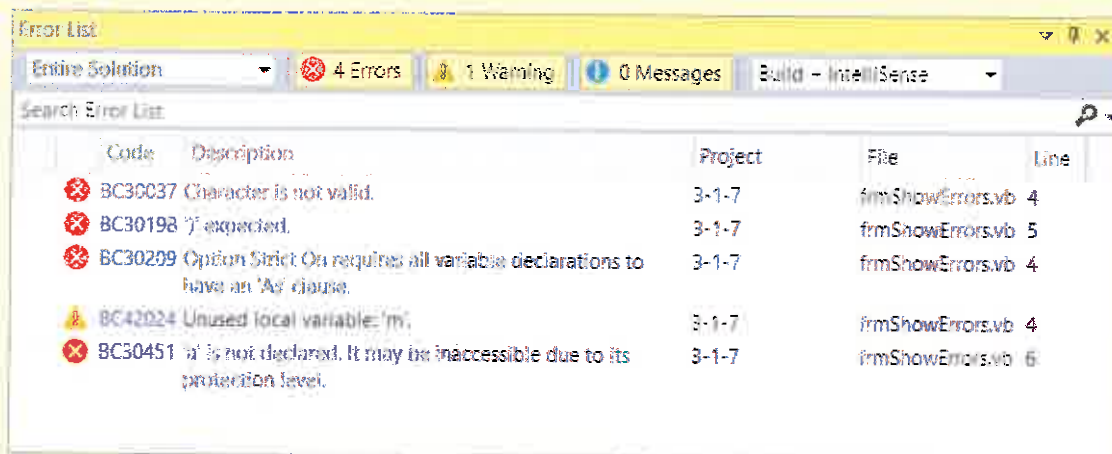


Example 7

Identify Errors The following program contains errors. **Note:** Line 1 of the Code Editor contains the Public Class statement and line 2 is a blank line. Therefore, the Private Sub statement is in line 3 and the Dim statement is in line 4.

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim m; n As Double
    lstResults.Items.Add(5
    lstResults.Items.Add(a)
End Sub
```

[Run, click on the button, and click on the No button in the error dialog box that appears.]



Note: If you double-click on one of the error lines in the Error List window, the Code Editor will appear with the cursor at the location of the error.

■ Comments

1. Declaring all variables at the beginning of each event procedure is considered good programming practice because it makes programs easier to read and helps prevent certain types of errors.
2. Keywords (reserved words) cannot be used as names of variables. For instance, the statements `Dim private As Double` and `Dim sub As Double` are not valid.
3. Names given to variables are sometimes referred to as *identifiers*.
4. In math courses, *literals* are referred to as *constants*. However, the word “constant” has a special meaning in programming languages.
5. Since a numeric expression is any combination of literals, variables, functions, and operators that can be evaluated to produce a number, a single literal or variable is a special case of an expression.
6. Numeric literals used in expressions or assigned to variables must not contain commas, dollar signs, or percent signs. Also, mixed numbers, such as $8 \frac{1}{2}$, are not allowed.
7. Although requesting the square root of a negative number does not terminate the execution of the program, it can produce unexpected results. For instance, the statement

```
lstBox.Items.Add(Math.Sqrt(-1))
```

displays **NaN**. **Note:** NaN is an abbreviation for “Not a Number.”

8. If the value of `numVar` is 0 and `numVar` has type `Double`, then the statements

```
numVarInv = 1 / numVar
lstBox.Items.Add(numVarInv)
lstBox.Items.Add(1 / numVarInv)
```

cause the following items to be displayed in the list box:

```
Infinity
0
```

9. When a number is halfway between two successive whole numbers (such as 1.5, 2.5, 3.5, and 4.5), the `Round` function rounds to the nearest even number. For instance, `Math.Round(2.5)` is 2 and `Math.Round(3.5)` is 4.

10. In scientific notation, numbers are written in the form $b \cdot 10^r$, where b is a number of magnitude from 1 up to (but not including) 10, and r is an integer. Visual Basic displays very large and very small numbers in **scientific notation**, where $b \cdot 10^r$ is written as bE^r . (The letter E is an abbreviation for *exponent*.) For instance, when the statement `lstBox.Items.Add(123 * (10 ^ 15))` is executed, $1.23E + 17$ is displayed in the list box.
11. If the total number of items added to a list box exceeds the number of items that can be displayed, a vertical scroll bar is automatically added to the list box.
12. When you first enter a statement such as `Dim n As Double`, a wavy green line will appear under the variable name and the Error List window will record a warning. The wavy line merely indicates that the variable has not yet been used. Warnings, unlike syntax errors, do not prevent the program from running. If the wavy green line is still present after the entire event procedure has been entered, this will tell you that the variable was never used and that the declaration statement can be removed.
13. Syntax errors prevent programs from starting; runtime and logic errors do not. A runtime error causes a program to stop executing and displays an error message. A logic error is a mistake that might cause incorrect output.
14. The rounding function can take the form `Math.Round(n, r)` or `Math.Round(n)`. That is, the argument r is optional. Microsoft documentation denotes the optionality of the r argument by referring to the function as `Math.Round(n[, r])`.
15. The functions discussed in this section are referred to as **built-in functions** since they are part of the Visual Basic language. Chapter 5 shows how we can create our own functions. Such functions are commonly referred to as *user-defined functions*. The term *user-defined* is a bit of a misnomer; such functions should really be called *programmer-defined functions*.

Practice Problems 3.1

1. Evaluate $2 + 3 * 4$.
2. Explain the difference between the assignment statement

```
var1 = var2
```

and the assignment statement

```
var2 = var1
```
3. Complete the table by filling in the value of each variable after each line is executed.

	a	b	c
Private Sub btnEvaluate_Click(...) Handles btnEvaluate.Click			
Dim a, b, c As Double	0	0	0
a = 3	3	0	0
b = 4	3	4	0
c = a + b			
a = c * a			
lstResults.Items.Add(a - b)			
b = b * b			
End Sub			

4. Write a statement that increases the value of the Double variable *var* by 5%.

EXERCISES 3.1

In Exercises 1 through 6, evaluate the numeric expression without a computer, and then use Visual Basic to check your answer.

- | | |
|------------------|---------------------|
| 1. $3 * 4$ | 2. $7 ^ 2$ |
| 3. $1 / (2 ^ 3)$ | 4. $3 + (4 * 5)$ |
| 5. $(5 - 3) * 4$ | 6. $3 * ((-2) ^ 5)$ |

In Exercises 7 through 12, evaluate the expression.

- | | |
|----------------------|------------------------|
| 7. $7 \setminus 3$ | 8. $14 \text{ Mod } 4$ |
| 9. $5 \setminus 5$ | 10. $7 \text{ Mod } 3$ |
| 11. $14 \setminus 4$ | 12. $5 \text{ Mod } 5$ |

In Exercises 13 through 18, determine whether the name is a valid variable name.

- | | |
|----------------|-----------------|
| 13. sales.2015 | 14. room&Board |
| 15. fOrM_1040 | 16. 1040B |
| 17. expenses? | 18. INCOME 2015 |

In Exercises 19 through 24, evaluate the numeric expression where $a = 2$, $b = 3$ and $c = 4$.

- | | |
|-------------------|-------------------|
| 19. $(a * b) + c$ | 20. $a * (b + c)$ |
| 21. $(1 + b) * c$ | 22. $a ^ c$ |
| 23. $b ^ (c - a)$ | 24. $(c - a) ^ b$ |

In Exercises 25 through 30, write lines of code to calculate and display the value of the expression.

- | | |
|-------------------|-------------------------------------|
| 25. $7 * 8 + 5$ | 26. $(1 + 2 * 9)^3$ |
| 27. 5.5% of 20 | 28. $15 - 3(2 + 3^4)$ |
| 29. $17(3 + 162)$ | 30. $4 \frac{1}{2} - 3 \frac{5}{8}$ |

In Exercises 31 and 32, complete the table by filling in the value of each variable after each line is executed.

31.

	x	y
Private Sub btnEvaluate Click(...) Handles btnEvaluate.Click		
Dim x, y As Double		
x = 2		
y = 3 * x		
x = y + 5		
lstResults.Items.Clear()		
lstResults.Items.Add(x + 4)		
y = y + 1		
End Sub		

32.

	bal	inter	withDr
Private Sub btnEvaluate_Click(...) Handles btnEvaluate.Click			
Dim bal, inter, withDr As Decimal			
bal = 100			
inter = 0.05D			
withDr = 25			
bal += inter * bal			
bal = bal - withDr			
End Sub			

In Exercises 33 through 40, determine the output displayed in the list box by the lines of code.

33. Dim amount As Double

amount = 10

lstOutput.Items.Add(amount - 4)

34. Dim a, b As Integer

a = 4

b = 5 * a

lstOutput.Items.Add(a + b)

35. Dim n As Integer = 7

n += 1

lstOutput.Items.Add(1)

lstOutput.Items.Add(n)

lstOutput.Items.Add(n + 1)

36. Dim num As Integer = 5

num = 2 * num

lstOutput.Items.Add(num)

37. Dim a, b As Integer

lstOutput.Items.Add(a + 1)

a = 4

b = a * a

lstOutput.Items.Add(a * b)

38. Dim tax As Double

tax = 200

tax = 25 + tax

lstOutput.Items.Add(tax)

39. Dim totalMinutes, hours, minutes As Integer

totalMinutes = 135

hours = totalMinutes \ 60

minutes = totalMinutes Mod 60

lstResults.Items.Add(hours)

lstResults.Items.Add(minutes)


```
40. Dim totalOunces, pounds, ounces As Integer
    totalOunces = 90
    pounds = totalOunces \ 16
    ounces = totalOunces Mod 16
    lstResults.Items.Add(pounds)
    lstResults.Items.Add(ounces)
```

In Exercises 41 through 46, identify the errors.

```
41. Dim a, b, c As Double
    a = 2
    b = 3
    a + b = c
    lstOutput.Items.Add(c)
```

```
42. Dim a, b, c, d As Double
    a = 2
    b = 3
    c = d = 4
    lstOutput.Items.Add(5((a + b) / (c + d)))
```

```
43. Dim balance, deposit As Decimal
    balance = 1,234D
    deposit = $100
    lstOutput.Items.Add(balance + deposit)
```

```
44. Dim interest, balance As Decimal
    0.05D = interest
    balance = 800
    lstOutput.Items.Add(interest * balance)
```

```
45. Dim 9W As Double
    9W = 2 * 9W
    lstOutput.Items.Add(9W)
```

```
46. Dim n As Double = 1.2345
    lstOutput.Items.Add(Round(n, 2))
```

In Exercises 47 and 48, rewrite the code using one line.

```
47. Dim quantity As Integer
    quantity = 12
```

```
48. Dim m As Integer
    Dim n As Double
    m = 2
    n = 3
```

In Exercises 49 through 54, find the value of the given function.

49. $\text{Int}(10.75)$

50. $\text{Int}(9 - 2)$

51. $\text{Math.Sqrt}(3 * 12)$

52. $\text{Math.Sqrt}(64)$

53. $\text{Math.Round}(3.1279, 3)$

54. $\text{Math.Round}(-2.6)$

In Exercises 55 through 60, find the value of the given function where a and b are numeric variables of type Double, $a = 5$ and $b = 3$.

55. $\text{Int}(-a/2)$

56. $\text{Math.Round}(a / b)$

57. $\text{Math.Sqrt}(a - 5)$

58. $\text{Math.Sqrt}(4 + a)$

59. $\text{Math.Round}(a + .5)$

60. $\text{Int}(b * 0.5)$

In Exercises 61 through 66, rewrite the statements using augmented assignment operators. Assume that each variable is of type Double.

61. $\text{cost} = \text{cost} + 5$

62. $\text{sum} = \text{sum} * 2$

63. $\text{cost} = \text{cost} / 6$

64. $\text{sum} = \text{sum} - 7$

65. $\text{sum} = \text{sum} ^ 2$

66. $\text{sum} = \text{sum} + 3$

In Exercises 67 through 74, write an event procedure with the header `Private Sub btnCompute_Click(...) Handles btnCompute.Click`, and that has one line of code for each step. Lines that display data should use the given variable names.

67. Calculate Profit The following steps calculate a company's profit:

- Declare the variables *revenue*, *costs*, and *profit* as type Decimal.
- Assign the value 98456 to the variable *revenue*.
- Assign the value 45000 to the variable *costs*.
- Assign the difference between the values of the variables *revenue* and *costs* to the variable *profit*.
- Display the value of the variable *profit* in a list box.

68. Stock Purchase The following steps calculate the amount of a stock purchase:

- Declare the variables *costPerShare*, *numberOfShares*, and *amount* as type Decimal.
- Assign the value 25.625D to the variable *costPerShare*.
- Assign the value 400 to the variable *numberOfShares*.
- Assign the product of the values of *costPerShare* and *numberOfShares* to the variable *amount*.
- Display the value of the variable *amount* in a list box.

69. Discounted Price The following steps calculate the price of an item after a 30% reduction:

- Declare the variables *price*, *discountPercent*, and *markdown* as type Decimal.
- Assign the value 19.95D to the variable *price*.
- Assign the value 30 to the variable *discountPercent*.
- Assign the value of (*discountPercent* divided by 100) times *price* to the variable *markdown*.
- Decrease the value of *price* by the value of *markdown*.
- Display the value of *price* (rounded to two decimal places) in a list box.

70. Break-Even Point The following steps calculate a company's break-even point, the number of units of goods the company must manufacture and sell in order to break even:

- Declare the variables *fixedCosts*, *pricePerUnit*, and *costPerUnit* as type Decimal.
- Assign the value 5000 to the variable *fixedCosts*.
- Assign the value 8 to the variable *pricePerUnit*.
- Assign the value 6 to the variable *costPerUnit*.

- (e) Assign the value of *fixedCosts* divided by (the difference of *pricePerUnit* and *costPerUnit*) to the variable *breakEvenPoint*.
- (f) Display the value of the variable *breakEvenPoint* in a list box.

71. Savings Account The following steps calculate the balance after three years when \$100 is deposited in a savings account at 5% interest compounded annually:

- (a) Declare the variable *balance* as type Decimal.
- (b) Assign the value 100 to the variable *balance*.
- (c) Increase the variable *balance* by 5% of its value. (Write 5% as 0.05D.)
- (d) Increase the variable *balance* by 5% of its value.
- (e) Increase the variable *balance* by 5% of its value.
- (f) Display the value of *balance* (rounded to two decimal places) in a list box.

72. Savings Account The following steps calculate the balance at the end of three years when \$100 is deposited at the beginning of each year in a savings account at 5% interest compounded annually:

- (a) Declare the variable *balance* as type Decimal.
- (b) Assign the value 100 to the variable *balance*.
- (c) Increase the value of variable *balance* by 5% of its value, and add 100. (5% = 0.05D)
- (d) Increase the value of variable *balance* by 5% of its value, and add 100.
- (e) Increase the value of variable *balance* by 5% of its value.
- (f) Display the value of *balance* (rounded to two decimal places) in a list box.

73. Percentage Markup The *markup* of an item is the difference between its *selling price* and its *purchase price*. The *percentage markup* is the quotient *markup/purchase price* expressed as a percentage. The following steps calculate the percentage markup of an item.

- (a) Declare the variable *purchasePrice* as type Decimal and assign it the value 215.50.
- (b) Declare the variable *sellingPrice* as type Decimal and assign it the value 644.99.
- (c) Declare the variable *markup* as type Decimal and assign it the difference of the selling price and the purchase price.
- (d) Declare the variable *percentageMarkup* as type Decimal and assign it 100 times the quotient of the markup and the purchase price.
- (e) Display the percentage markup in a list box.

74. Profit Margin The *markup* of an item is the difference between its *selling price* and its *purchase price*. The *profit margin* is the quotient *markup/selling price* expressed as a percentage. The following steps calculate the profit margin of an item.

- (a) Declare the variable *purchasePrice* as type Decimal and assign it the value 215.50.
- (b) Declare the variable *sellingPrice* as type Decimal and assign it the value 29.99.
- (c) Declare the variable *markup* as type Decimal and assign it the difference of the selling price and the purchase price.
- (d) Declare the variable *profitMargin* as type Decimal and assign it 100 times the quotient of the markup and the selling price.
- (e) Display the profit margin in a list box.

In Exercises 75 through 86, write a program to solve the problem and display the answer in a list box. The program should use variables for each of the quantities.

- 75. Corn Production** Suppose each acre of farmland produces 18 tons of corn. How many tons of corn can be grown on a 30-acre farm? See Fig. 3.1.



FIGURE 3.1 Outcome of Exercise 75.

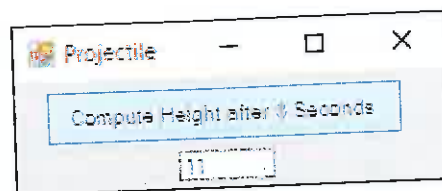


FIGURE 3.2 Outcome of Exercise 76.

- 76. Projectile Motion** Suppose a ball is thrown straight up in the air with an initial velocity of 50 feet per second and an initial height of 5 feet. How high will the ball be after 3 seconds? See Fig. 3.2.

Note: The height after t seconds is given by the expression $-16t^2 + v_0t + h_0$, where v_0 is the initial velocity and h_0 is the initial height.

- 77. Average Speed** If a car left Washington, D.C., at 2 o'clock and arrived in New York at 7 o'clock, what was its average speed? **Note:** New York is 233 miles from Washington. See Fig. 3.3.

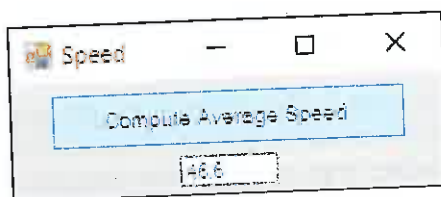


FIGURE 3.3 Outcome of Exercise 77.

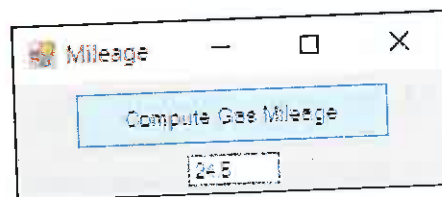


FIGURE 3.4 Outcome of Exercise 78.

- 78. Gas Mileage** A motorist wants to determine her gas mileage. At 23,352 miles (on the odometer) the tank is filled. At 23,695 miles the tank is filled again with 14 gallons. How many miles per gallon did the car average between the two fillings? See Fig. 3.4.

- 79. Water Usage** A survey showed that Americans use an average of 1600 gallons of water per person per day, including industrial use. How many gallons of water are used each year in the United States? **Note:** The current population of the United States is about 315 million people. See Fig. 3.5.

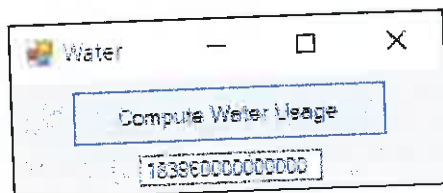


FIGURE 3.5 Outcome of Exercise 79.

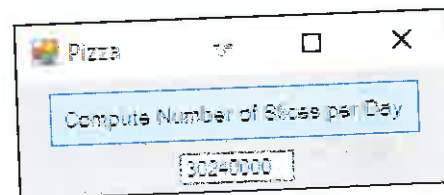


FIGURE 3.6 Outcome of Exercise 80.

- 80. Pizza** Americans eat an average of 350 slices of pizza per second. How many slices of pizza do they eat per day? See Fig. 3.6.
- 81. Restaurants** About 12% of the restaurants in the United States are pizzerias, and there are about 70,000 pizzerias in the United States. Estimate the total number of restaurants in the United States. See Fig. 3.7.

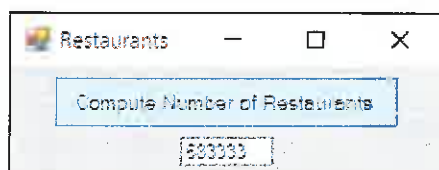


FIGURE 3.7 Outcome of Exercise 81.

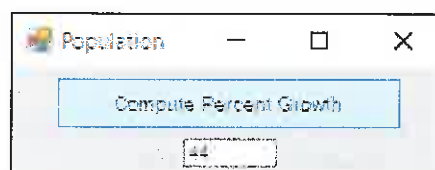


FIGURE 3.8 Outcome of Exercise 82.

- 82. Population Growth** The population of the United States was about 281 million in 2000 and is predicted to be about 404 million in 2050. Approximate the percentage population growth in the United States during the first half of the 21st century. Round the percentage to the nearest whole number. See Fig. 3.8.
- 83. Convert Speeds** On May 6, 1954, British runner Sir Roger Bannister became the first person to run the mile in less than 4 minutes. His average speed was 24.20 kilometers per hour. Write a program that converts kilometers per hour to miles per hour. See Fig. 3.9. **Note:** One kilometer is .6214 of a mile.

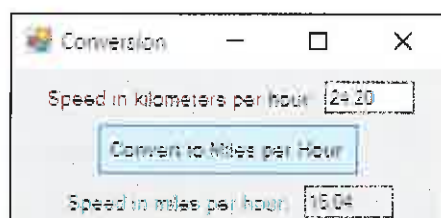


FIGURE 3.9 Outcome of Exercise 83.

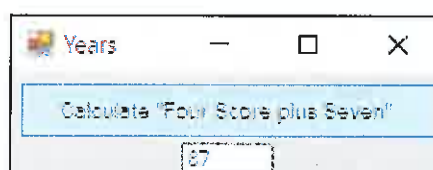


FIGURE 3.10 Outcome of Exercise 84.

- 84. Gettysburg Address** The number 20 is called a *score*. Write a program to calculate the value of “four score plus seven.” See Fig. 3.10.
- 85. Calories** Estimate the number of calories in one cubic mile of chocolate ice cream. See Fig. 3.11. **Note:** There are 5280 feet in a mile and one cubic foot of chocolate ice cream contains about 48,600 calories.

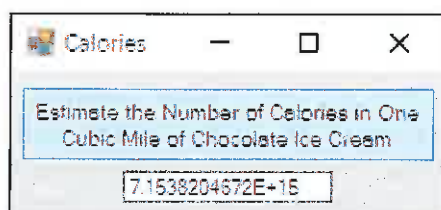


FIGURE 3.11 Outcome of Exercise 85.

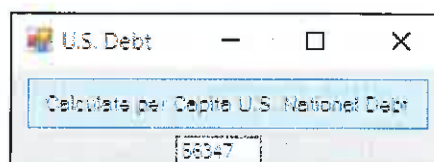


FIGURE 3.12 Outcome of Exercise 86.

- 86. U.S. National Debt** Suppose the U.S. national debt is $1.82 \cdot 10^{13}$ dollars and the U.S. population is $3.23 \cdot 10^8$. Calculate the per capita U.S. national debt. Display the answer rounded to the nearest whole number. See Fig. 3.12.

Solutions to Practice Problems 3.1

1. 14. Multiplications are performed before additions. If the intent is for the addition to be performed first, the expression should be written $(2 + 3) * 4$.
2. The first assignment statement assigns the value of the variable *var2* to the variable *var1*, whereas the second assignment statement assigns *var1*'s value to *var2*.

3.

	a	b	c
Private Sub btnEvaluate_Click(...) Handles btnEvaluate.Click			
Dim a, b, c As Double	0	0	0
a = 3	3	0	0
b = 4	3	4	0
c = a + b	3	4	7
a = c * a	21	4	7
lstResults.Items.Add(a - b)	21	4	7
b = b * b	21	16	7
End Sub			

Each time an assignment statement is executed, only one variable (the variable to the left of the equal sign) has its value changed.

4. Each of the three following statements increases the value of *var* by 5%.

```
var = var + (0.05 * var)
var = 1.05 * var
var += 0.05 * var
```

3.2 Strings

The most common types of data processed by Visual Basic are numbers and strings. Sentences, phrases, words, letters of the alphabet, names, telephone numbers, addresses, and social security numbers are all examples of strings. Formally, a **string literal** is a sequence of characters that is treated as a single item. (The characters in our strings will be letters, digits, punctuation, and special symbols such as \$, #, and %.) Double quotes are used to mark its beginning and end. String literals can be assigned to variables, displayed in text boxes and list boxes, and combined by an operation called concatenation (denoted by &).

Variables and Strings

A **string variable** is a name used to refer to a string. The allowable names of string variables are the same as those of numeric variables. The value of a string variable is assigned or altered with assignment statements and can be displayed in a list box like the value of a numeric variable. String variables are declared with statements of the form

```
Dim varName As String
```



Example 1

Display Output The following program shows how assignment statements and the Add method are used with strings. The string variable *president* is assigned a value by the third line, and this value is displayed by the sixth line. The quotation marks surrounding each string literal mark the beginning and end of the string. They are not part of the literal and are not displayed by the Add method. (The form for this example contains a button and a list box.) **Note:** The Code Editor colors string literals red.


```

Private Sub btnDisplay_Click(. . .) Handles btnDisplay.Click
    Dim president As String
    president = "George Washington"
    lstOutput.Items.Clear()
    lstOutput.Items.Add("president")
    lstOutput.Items.Add(president)
End Sub

```

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

```

president
George Washington

```

If x, y, \dots, z are characters and *strVar* is a string variable, then the statement

```
strVar = "xy...z"
```

assigns the string literal $xy \dots z$ to the variable and the statement

```
lstBox.Items.Add("xy...z")
```

or

```
lstBox.Items.Add(strVar)
```

displays the string $xy \dots z$ in a list box. If *strVar2* is another string variable, then the statement

```
strVar2 = strVar
```

assigns the value of the variable *strVar* to the variable *strVar2*. (The value of *strVar* will remain the same.) String literals used in assignment or `lstBox.Items.Add` statements must be surrounded by quotation marks, but string variables are never surrounded by quotation marks.

Using Text Boxes for Input and Output

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

```
strVar = txtBox.Text
```

and

```
txtBox.Text = strVar
```

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

Numbers typed into text boxes are stored as strings. Such strings must be converted to numeric values before they can be assigned to numeric variables or used in numeric expressions. The functions `Cdbl`, `Cdec`, and `Cint` convert strings representing numbers (such as "20") into numbers of type `Double`, `Decimal`, and `Integer`, respectively. Going in the other direction, the function `Cstr` converts a number into a string representation of the number. Therefore, statements such as

```
dblVar = Cdbl(txtBox.Text)
```

and

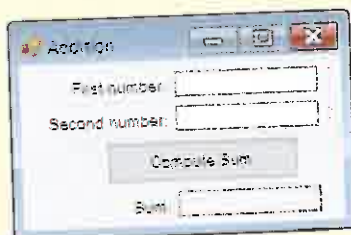
```
txtBox.Text = Cstr(dblVar)
```

can be used to assign the contents of a text box to the double variable *dblVar* and vice versa. *CDbl*, *CDec*, *CInt*, and *CStr*, which stand for “convert to Double”, “convert to Decimal”, “convert to Integer”, and “convert to String”, are referred to as **data conversion** or **type-casting functions**.



Example 2

Addition The following program adds two numbers supplied by the user:

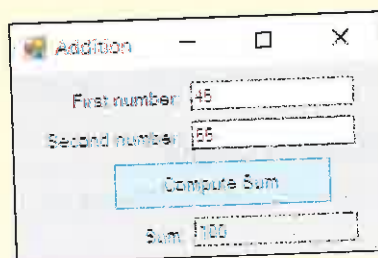


OBJECT	PROPERTY	SETTING
frmAdd	Text	Addition
lblFirstNum	Text	First number:
txtFirstNum		
lblSecondNum	Text	Second number:
txtSecondNum		
btnCompute	Text	Compute Sum
lblSum	Text	Sum:
txtSum	ReadOnly	True

```
Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim num1, num2, sum As Double
    num1 = CDbl(txtFirstNum.Text)
    num2 = CDbl(txtSecondNum.Text)
    sum = num1 + num2
    txtSum.Text = CStr(sum)
End Sub
```

End Sub

[Run, type 45 into the first text box, type 55 into the second text box, and click on the button.]



Option Explicit and Option Strict

Option Explicit and **Option Strict** affect how programs are written. **Option Explicit** requires that all variables be declared with **Dim** statements. The disabling of this option can lead to errors resulting from the misspelling of names of variables. **Option Strict** requires explicit conversions with typecasting functions in most cases where a value or variable of one type is assigned to a variable of another type. The absence of this option can lead to data loss. Having both **Option Explicit** and **Option Strict** enabled is considered good programming practice. In this book, we assume that both options are in effect.

Visual Basic provides a way to enforce **Option Explicit** and **Option Strict** for all programs you create. Click on **Options** in the Menu bar's Tools menu to open the Options dialog box. In the left pane, expand the Projects and Solutions entry. Then click on the subentry VB Defaults. Four default project settings will appear on the right. (See Fig. 3.13 on the next page.) If the settings for **Option Explicit** and **Option Strict** are not already set to On, change them to On. **Note:** **Option Infer** is discussed in Chapter 6.

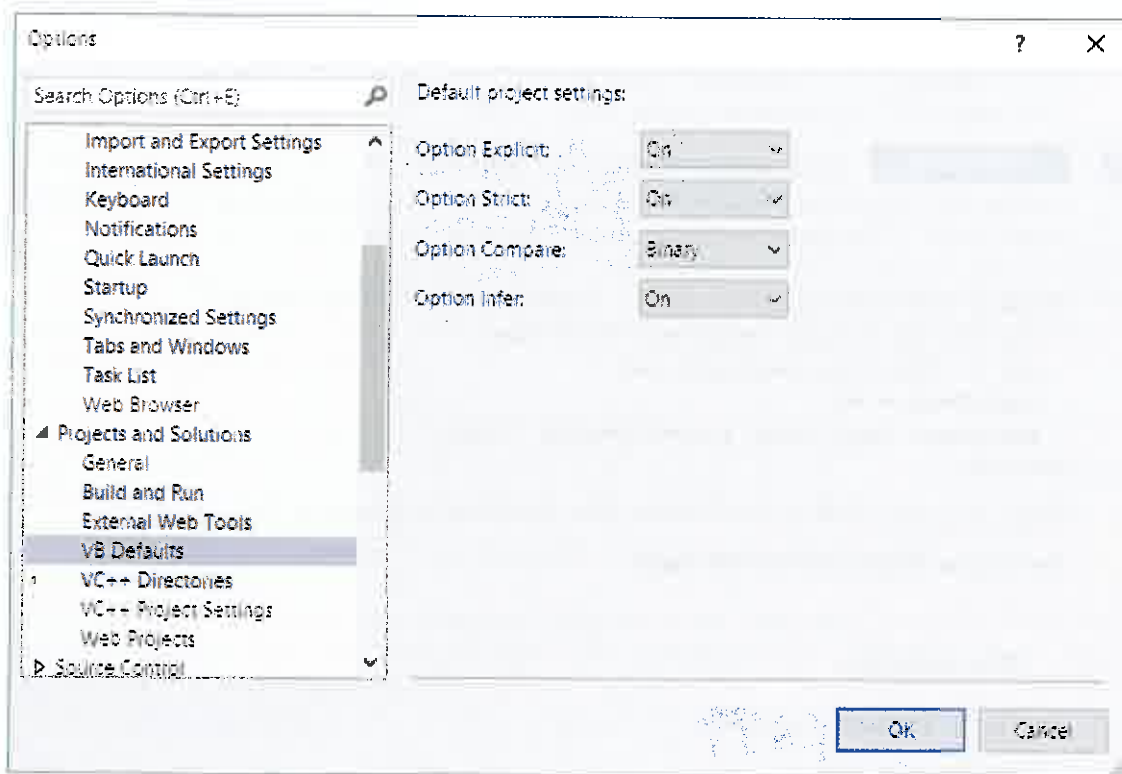


FIGURE 3.13 Option default project settings.

Concatenation

Two strings can be combined to form a new string consisting of the strings joined together. This operation is called **concatenation** and is represented by an ampersand (&). For instance, "good" & "bye" is "goodbye". (Concatenation can be thought of as *addition for strings*.) A combination of strings and ampersands that can be evaluated to form a string is called a **string expression**. When a string expression appears in an assignment statement or an Add method, the string expression is evaluated before being assigned or displayed.

Example 3 Concatenate Strings The following program illustrates concatenation. (The form for this example contains a button and a text box.) Notice the space at the end of the string assigned to *quote1*. If that space was not present, then the statement that assigns a value to *quote* would have to be `quote = quote1 & " " & quote2` in order to achieve the same output.

```
Private Sub btnDisplay_Click(...) Handles btnDisplay.Click
    Dim quote1, quote2, quote As String
    quote1 = "The ballgame isn't over, "
    quote2 = "until it's over."
    quote = quote1 & quote2
    txtOutput.Text = quote & " Yogi Berra"
End Sub
```

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

The ball game isn't over, until it's over. Yogi Berra

Visual Basic also allows strings to be concatenated with numbers and allows numbers to be concatenated with numbers. In each case, the result is a string.



Example 4

Concatenate Strings and Numbers The following program concatenates a string with a number. Notice that a space was inserted after the word "has" and before the word "keys." (The form for this example contains a button and a text box.)

```
Private Sub btnDisplay_Click(...) Handles btnDisplay.Click
    Dim str As String, numOfKeys As Integer
    str = "The piano keyboard has "
    numOfKeys = 88
    txtOutput.Text = str & numOfKeys & " keys."
End Sub
```

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

The piano keyboard has 88 keys.

The statement

```
strVar = strVar & strVar2
```

will assign to *strVar* a new string that is the concatenation of *strVar* and *strVar2*. The same result can be accomplished with the statement

```
strVar &= strVar2
```

String Properties and Methods

We have seen that controls, such as text and list boxes, have properties and methods. A control placed on a form is an example of an object. A string is also an object, and, like a control, has both properties and methods that are specified by following the string with a period and the name of the property or method. The **Length** property gives the number of characters in a string. The **ToUpper** and **ToLower** methods produce a copy of a string in uppercase and lowercase characters. The **Trim** method deletes all leading and trailing spaces from a string. The **Substring** method extracts a sequence of consecutive characters from a string. The **IndexOf** method searches for the first occurrence of one string in another and gives the position at which the first occurrence is found.

If *str* is a string, then

str.Length

is the number of characters (including spaces and punctuation marks) in the string,

str.ToUpper

is the string with all its letters capitalized,

str.ToLower

is the string with all its letters in lowercase, and

str.Trim

is the string with all spaces removed from the front and back of the string. For instance,

EXPRESSION	VALUE	EXPRESSION	VALUE
"Visual".Length	6	"Visual".ToUpper	VISUAL
"123 Hike".Length	8	"123 Hike".ToLower	123 hike
"a" & " b ".Trim & "c"	abc		

Indices and Substrings

The **position** or **index** of a character in a string is identified with one of the numbers 0, 1, 2, 3, For instance, the first character of a string is said to have index 0, the second character is said to have index 1, and so on. Figure 3.14 shows the indices of the characters of the string "spam & eggs".

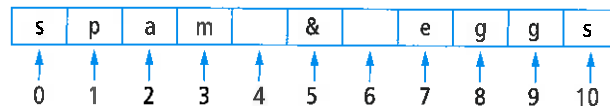


FIGURE 3.14 Indices of the characters of the string "spam & eggs".

A **substring** of a string is a sequence of consecutive characters from the string. For instance, consider the string "spam & eggs". The substrings "spa", "am", and "ggs" begin at positions 0, 2, and 8, respectively.

If *str* is a string, then

str.Substring(*m*, *n*)

is the substring of *str* consisting of *n* characters beginning with the character having index *m* of *str*. If the comma and the number *n* are omitted, then the substring starts at position *m* and continues until the end of *str*. The value of

str.IndexOf(*str2*)

is -1 if *str2* is not a substring of *str*; otherwise it is the beginning index of the first occurrence of *str2* in *str*. Some examples using these two methods are as follows:

EXPRESSION	VALUE	EXPRESSION	VALUE
"fantastic".Substring(0, 3)	"fan"	"fantastic".IndexOf("tas")	3
"fantastic".Substring(6, 2)	"ti"	"fantastic".IndexOf("a")	1
"fantastic".Substring(6)	"tic"	"fantastic".IndexOf("tn")	-1

The IndexOf method has a useful extension. The value of **str.IndexOf(*str2*, *n*)**, where *n* is an integer, is the index of the first occurrence of *str2* in *str* having index *n* or greater. For instance, the value of **"Mississippi".IndexOf("ss", 3)** is 5.

Like the numeric functions discussed before, string properties and methods also can be applied to variables and expressions.



Example 5 String Operations The following program uses variables and expressions with the property and methods just discussed:

```
Private Sub btnEvaluate_Click(...) Handles btnEvaluate.Click
    Dim str1, str2, str3 As String
    str1 = "Quick as "
    str2 = "a wink"
    lstResults.Items.Clear()
```

```

lstResults.Items.Add(str1.Substring(0, 7))
lstResults.Items.Add(str1.IndexOf("c"))
lstResults.Items.Add(str1.Substring(0, 3))
lstResults.Items.Add((str1 & str2).Substring(6, 6))
lstResults.Items.Add((str1 & str2).ToUpper)
lstResults.Items.Add(str1.Trim & str2)
str3 = str2.Substring(str2.Length - 4)
lstResults.Items.Add("The average " & str3 & " lasts .1 second.")
End Sub

```

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

```

Quick a
3
Qui
as a w
QUICK AS A WINK
Quick asa wink
The average wink lasts .1 second.

```

Note: In Example 5, *c* is in the third position of *str1*, and there are three characters of *str1* to the left of *c*. In general, there are *n* characters to the left of the character in position *n*. This fact is used in Example 6.



Example 6

Parse a Name The following program parses a name. The fifth line locates the position, call it *n*, of the space separating the two names. The first name will contain *n* characters, and the last name will consist of all characters to the right of the *n*th character.

```

Private Sub btnAnalyze_Click(...) Handles btnAnalyze.Click
    Dim fullName, firstName, lastName As String
    Dim n As Integer
    fullName = txtName.Text
    n = fullName.IndexOf(" ")
    firstName = fullName.Substring(0, n)
    lastName = fullName.Substring(n + 1)
    lstResults.Items.Clear()
    lstResults.Items.Add("First name: " & firstName)
    lstResults.Items.Add("Your last name has " & lastName.Length & " letters.")
End Sub

```

[Run, type "Charles Babbage" into the text box, and then click on the button.]

The screenshot shows a window titled "Parse a Name". Inside, there is a text box labeled "Name (first and last only):" containing the text "Charles Babbage". Below the text box is a button labeled "Analyze Name". To the right of the button is a list box containing two items: "First name: Charles" and "Your last name has 7 letters."

■ The Empty String

The string "", which contains no characters, is called the **empty string** or the **zero-length string**. It is different from " ", the string consisting of a single space.

The statement `lstBox.Items.Add("")` inserts a blank line into the list box. The contents of a text box can be cleared with either the statement

```
txtBox.Clear()
```

or the statement

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

■ Initial Value of a String Variable

When a string variable is declared with a `Dim` statement, it has the keyword `Nothing` as its default value. To specify a different initial value, follow the declaration statement with an equal sign followed by the initial value. For instance, the statement

```
Dim pres As String = "Adams"
```

declares the variable `pres` to be of type `String` and assigns it the initial value "Adams".

An error occurs whenever an attempt is made to access a property or method for a string variable having the value `Nothing` or to display it in a list box. Therefore, unless a string variable is guaranteed to be assigned a value before being used, you should initialize it—even if you just assign the empty string to it.

■ Widening and Narrowing

The assignment of a value or variable of type `Double` to a variable of type `Integer` or `Decimal` is called **narrowing** because the possible values of an `Integer` and `Decimal` variable are a subset of the possible values of a `Double` variable. The assignment of a value or variable of type `Decimal` to a variable of type `Integer` is another example of narrowing. As there are more possible values of a variable of type `Double` than of type `Integer` or `Decimal`, assigning in the reverse direction is called **widening**. Option Strict requires the use of a typecasting function when narrowing, but allows widening without a typecasting function. Specifically, widening statements of the form

```
dblVar = intVar, decVar = intVar, and dblVar = decVar
```

are valid. However, narrowing statements of the form

```
intVar = dblVar, intVar = decVar, and decVar = dblVar
```

are not valid. They must be replaced with

```
intVar = CInt(dblVar), intVar = CInt(decVar), and decVar = CDec(dblVar)
```

Great care must be taken when computing with `Integer` variables. For instance, the value of an expression involving division or exponentiation has type `Double` and therefore cannot be assigned to an `Integer` variable without explicit conversion even if the value is a whole number. For instance, Option Strict makes each of the following two assignment statements invalid.

```
Dim m As Integer
m = 2 ^ 3
m = 6 / 2
```

In order to avoid such errors, we primarily use variables of type `Integer` for counting or identifying positions.

■ Data Types of Literals and Expressions

Visual Basic considers literal numbers written without a decimal point to have type Integer, and literal numbers written with a decimal point to have type Double. Only literal numbers with the suffix D are considered to have type Decimal.

The value of an expression involving exponentiation has type Double and therefore must be converted before it can be assigned to an Integer or Decimal variable. For instance, Option Strict makes the following statements invalid:

```
Dim m As Integer = 2 ^ 3
Dim n As Decimal = 2D ^ 3
```

They must be changed to the following statements:

```
Dim m As Integer = CInt(2 ^ 3)
Dim n As Decimal = CDec(2D ^ 3)
```



A quotient of two Decimal values or a quotient of a Decimal value and an Integer value has type Decimal. However, the quotient of two Integer values has type Double.

If one of the terms in a numeric expression has data type Double, then the value of the expression will have type Double. If one of the terms in a numeric expression has data type Decimal and there are no terms of type Double, then the value of the expression will have type Decimal. Table 3.2 gives some examples of the data types of expressions.

TABLE 3.2 Data types of expressions.

Expression	Data Type	Expression	Data Type
3D + 4.6 + 2.2D	Double	3 + 4D	Decimal
(6/2) + 5	Double	6D / 3D	Decimal
3.0 + 4D	Double	6D / 3	Decimal

■ Internal Documentation

Program documentation is the inclusion of comments that are meant to be read only by the programmer and the person maintaining the code. They specify the intent of the program, the purpose of the variables, and the tasks performed by individual portions of the program. To create a comment statement, begin the line with an apostrophe. Such a statement appears green on the screen and is completely ignored by Visual Basic when the program is executed. Comments are sometimes called *remarks*. A line of code can be documented by adding an apostrophe, followed by the desired information, after the end of the line. The **Comment Out** button () and the **Uncomment** button () on the Toolbar can be used to comment and uncomment selected blocks of code.



Example 7

Parse a Name The following rewrite of Example 6 uses internal documentation. The first comment describes the entire program, the comment in line 5 gives the meaning of the variable, and the final comment describes the purpose of the three lines that follow it.

```
Private Sub btnAnalyze_Click(...) Handles btnAnalyze.Click
    'Determine a person's first name and the length of the second name
    Dim fullName, firstName, lastName As String
```

```

Dim m As Integer
Dim n As Integer 'location of the space separating the two names
fullName = txtName.Text
n = fullName.IndexOf(" ")
firstName = fullName.Substring(0, n)
lastName = fullName.Substring(n + 1)
m = lastName.Length
'Display the desired information in a list box
lstResults.Items.Clear()
lstResults.Items.Add("First name: " & firstName)
lstResults.Items.Add("Your last name has " & m & " letters.")
End Sub

```

Some of the benefits of documentation are as follows:

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

Good programming practice requires that programmers document their code while they are writing it. In fact, many software companies require a certain level of documentation before they release software and some judge a programmer's performance on how well their code is documented.

■ Line Continuation

Thousands of characters can be typed in a line of code. If you use a statement with more characters than can fit in the window, Visual Basic scrolls the Code Editor toward the right as needed. However, most programmers prefer having lines that are no longer than the width of the Code Editor. A long statement can be split across two or more lines by ending each line (except the last) with an underscore character (`_`) preceded by a space. For instance, the line

```
Dim quotation As String = "Good code is its own best documentation."
```

can be written as

```
Dim quotation As String = "Good code is its own " & _
    "best documentation."
```

A feature called **implicit line continuation** allows underscore characters to be omitted from the end of a line that obviously has a continuation—for instance, a line that ends with an ampersand, an arithmetic operator, or a comma. We use this feature throughout this textbook. For example, the line above will be written

```
Dim quotation As String = "Good code is its own " &
    "best documentation."
```

Line continuation, with or without an underscore character, cannot be used inside a pair of quotation marks. Whenever you want to display a literal string on two lines of the screen, you must first break it into two shorter strings joined with an ampersand. IntelliSense is reliable in letting you know if you have broken a line improperly.

Line continuation, with or without an underscore character, does not work with comment statements. Therefore, each line of a comment statement must begin with its own apostrophe.



VideoNote
Variable
Scope

Scope of a Variable

When a variable is declared in an event procedure with a `Dim` statement, a portion of memory is set aside to hold the value of the variable. As soon as the `End Sub` statement for the procedure is reached, the memory location is freed up; that is, the variable is discarded. The variable is said to be **local** to the procedure and to have **procedure scope**. In general, the **scope** of a variable is the portion of the program that can refer to it.

Visual Basic provides a way to make a variable recognized by every procedure in a form's code. Such a variable is called a **class-level variable** and is said to have **class scope**. The `Dim` statement for a class-level variable can be placed anywhere between the statements `Public Class formName` and `End Class`, provided that the `Dim` statement is not inside an event procedure. Normally, we place the `Dim` statement just after the `Public Class formName` statement. (We refer to this region as the **Declarations section** of the class.) When a class-level variable has its value changed by a procedure, the value persists even after the procedure has finished executing.

If an event procedure declares a local variable with the same name as a class-level variable, then the name refers to the local variable for code inside the procedure. To refer to the class-level variable inside the procedure, prefix the name with `Me` followed by a period.



Example 8 Enumeration The following program uses a variable having class scope to keep track of the number of times a button has been clicked:

```
Public Class frmCount
```

```
    Dim numTimes As Integer = 0    'class-level variable
```

```
    Private Sub btnPushMe_Click(...) Handles btnPushMe.Click
```

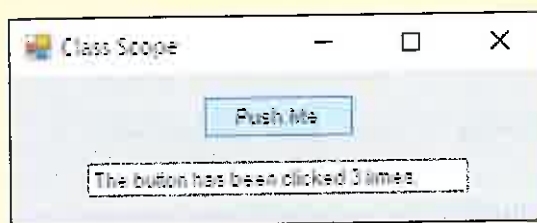
```
        numTimes += 1
```

```
        txtOutput.Text = "The button has been clicked " &  
                        numTimes & " times."
```

```
    End Sub
```

```
End Class
```

[Run, and click on the button three times.]



Auto Correction

The **Auto Correction** feature of IntelliSense suggests corrections when errors occur and allows you to select a correction to be applied to the code. When an invalid statement is entered, a wavy red error line appears under the incorrect part of the statement, and the Auto Correction feature is available for the error. When you hover the cursor over the wavy line, a small Error Correction icon (🔧) appears along with a tinted rectangular box describing the error. Figures 3.15, 3.16, and 3.17 show how Auto Correction points out a type-conversion error and assists with the correction.

When you click on the down-arrow of the Error Correction icon (or the last line of the tinted box) in Fig. 3.15, a potential fix for the problem is displayed. See Fig. 3.16.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Han
    Dim n As Integer
```



FIGURE 3.15 Description-of-Error box.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim n As Integer
```



FIGURE 3.16 Suggested-Change box.

When you click on *Preview changes*, a Preview Changes window (see Fig. 3.17) shows a corrected portion of the program. After you click on the *Apply* button, the recommended correction to the program will be carried out.

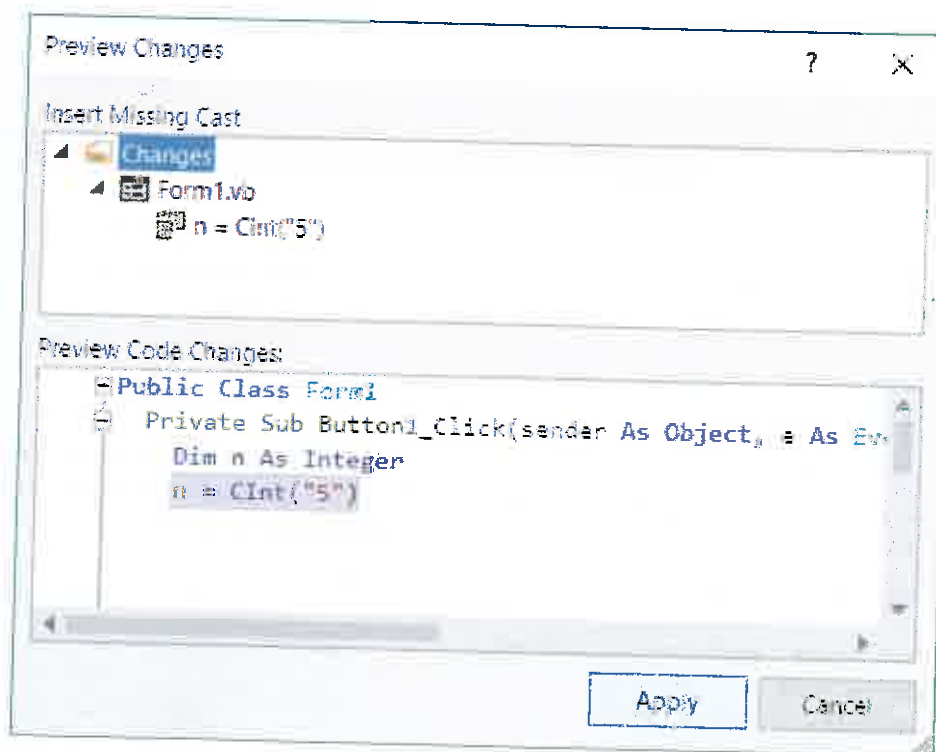


FIGURE 3.17 Description-of-Error box.

Comments

1. From the Code Editor, you can determine the type of a variable by letting the mouse pointer hover over the variable name until a tooltip giving the type appears. This feature of IntelliSense is called **Quick Info**.
2. Variable names should describe the role of the variable. Also, some programmers use a prefix, such as *dbl* or *str*, to identify the type of a variable. For example, they would use names like *dblInterestRate* and *strFirstName*. This naming convention is not needed in Visual Basic for the reason mentioned in Comment 1, and is no longer recommended by Microsoft.
3. Since a string expression is any combination of literals, variables, functions, methods, and operators that can be evaluated to produce a string, a single string literal or variable is a special case of an expression.
4. The functions *CInt*, *CDec*, and *CDBl* are user friendly. If the user types a number into a text box and precedes it with a dollar sign or inserts commas as separators, the values of *CInt* (*txtBox.Text*), *CDec* (*txtBox.Text*), and *CDBl* (*txtBox.Text*) will be the number with the dollar sign and/or commas removed.
5. The *Trim* method is useful when reading data from a text box. Sometimes users type spaces at the end of the input. Unless the spaces are removed, they can cause havoc elsewhere in the program.
6. When an incorrect or missing conversion to a number or date is detected by the Code Editor, Auto Correction recommends and implements the use of the *CInt*, *CDec*, *CDBl*, or *CDate* functions.

When an incorrect or missing conversion to a string is detected by the Code Editor, Auto Correction recommends the function *CType*. For instance, the recommended correction for `txtBox.Text = 5` is `txtBox.Text = CType(5, String)`. However, we have decided to favor `txtBox.Text = CStr(5)` since the use of the *CStr* function is consistent with the other conversion function recommendations. Also, the use of *CStr* makes code less cluttered and easier to read.

Practice Problems 3.2

1. What is the value of `"Computer".IndexOf("E")`?
2. What is the difference in the output produced by the following two statements? Why is *CStr* used in the first statement, but not in the second?

```
txtBox.Text = CStr(8 + 8)
txtBox.Text = 8 & 8
```

3. Give an example of a prohibited statement that invokes an Auto Correction helper box with the heading "Option Strict On disallows implicit conversion from 'String' to 'Double'." Also, give the suggestion for fixing the error.
4. Identify the error in the following lines of code:

```
Dim price, newPrice As Decimal
price = 15
newPrice = 15 + 1.25
```


EXERCISES 3.2

In Exercises 1 through 28, determine the output displayed in the text box or list box by the lines of code.

1. `txtBox.Text = "Visual Basic"`
2. `lstBox.Items.Add("Hello")`
3. `Dim var As String`
`var = "Ernie"`
`lstBox.Items.Add(var)`
4. `Dim var As String`
`var = "Bert"`
`txtBox.Text = var`
5. `txtBox.Text = "f" & "lute"`
6. `lstBox.Items.Add("a" & "cute")`
7. `Dim var As Double`
`var = 123`
`txtBox.Text = CStr(var)`
8. `Dim var As Double`
`var = 3`
`txtBox.Text = CStr(var + 5)`
9. `txtBox.Text = "Your age is " & 21 & ", "`
10. `txtBox.Text = "Fred has " & 2 & " children."`
11. `Dim r, b As String`
`r = "A ROSE"`
`b = " IS "`
`txtBox.Text = r & b & r & b & r`
12. `Dim s As String, n As Integer`
`s = "trombones"`
`n = 76`
`txtBox.Text = n & " " & s`
13. `Dim num As Double`
`txtBox.Text = "5"`
`num = 0.5 + Cdbl(txtBox.Text)`
`txtBox.Text = CStr(num)`
14. `Dim num As Integer = 2`
`txtBox.Text = CStr(num)`
`txtBox.Text = CStr(1 + CInt(txtBox.Text))`
15. `txtBox.Text = "good"`
`txtBox.Text &= "bye"`
16. `Dim var As String = "eight"`
`var &= "h"`
`txtBox.Text = var`
17. `Dim var As String = "WALLA"`
`var &= var`
`txtBox.Text = var`
18. `txtBox.Text = "mur"`
`txtBox.Text &= txtBox.Text`

19. `lstBox.Items.Add("aBc".ToUpper)`
`lstBox.Items.Add("Wallless".IndexOf("lll"))`
`lstBox.Items.Add("five".Length)`
`lstBox.Items.Add(" 55 ".Trim & " mph")`
`lstBox.Items.Add("UNDERSTUDY".Substring(5, 3))`
20. `lstBox.Items.Add("8 Ball".ToLower)`
`lstBox.Items.Add("colonel".IndexOf("k"))`
`lstBox.Items.Add("23.45".Length)`
`lstBox.Items.Add("revolutionary".Substring(1))`
`lstBox.Items.Add("whippersnapper".IndexOf("pp", 5))`
21. `Dim a As Integer = 4`
`Dim b As Integer = 2`
`Dim c As String = "Municipality"`
`Dim d As String = "pal"`
`lstBox.Items.Add(c.Length)`
`lstBox.Items.Add(c.ToUpper)`
`lstBox.Items.Add(c.Substring(a, b) & c.Substring(5 * b))`
`lstBox.Items.Add(c.IndexOf(d))`
22. `Dim m As Integer = 4`
`Dim n As Integer = 3`
`Dim s As String = "Microsoft"`
`Dim t As String = "soft"`
`lstOutput.Items.Add(s.Length)`
`lstOutput.Items.Add(s.ToLower)`
`lstOutput.Items.Add(s.Substring(m, n - 1))`
`lstOutput.Items.Add(s.IndexOf(t))`
23. How many positions does a string of eight characters have?
24. What is the highest numbered position for a string of eight characters?
25. (True or False) If *n* is the length of *str*, then `str.Substring(n - 1)` is the string consisting of the last character of *str*.
26. (True or False) If *n* is the length of *str*, then `str.Substring(n - 2)` is the string consisting of the last two characters of *str*.

In Exercises 27 through 34, identify any errors.

27. `Dim phoneNumber As Double`
`phoneNumber = "234-5678"`
`txtBox.Text = "My phone number is " & phoneNumber`
28. `Dim quote As String`
`quote = I came to Casablanca for the waters.`
`txtBox.Text = quote & ": " & "Bogart"`
29. `Dim end As String`
`end = "happily ever after."`
`txtBox.Text = "They lived " & end`
30. `Dim hiyo As String`
`hiyo = "Silver"`
`txtBox = "Hi-Yo " & hiYo`
31. `Dim num As Double = 1234`
`txtBox.Text = CStr(num.IndexOf("2"))`
32. `Dim num As Integer = 45`
`txtBox.Text = CStr(num.Length)`

33. `Dim m As Decimal, n As Integer`
`m = 2 ^ 3`
`n = 4 / 2`

34. `Dim m As Decimal, n As Integer`
`m = 3.45`
`n = 2.0 + 3.0`

In Exercises 35 and 36, write an event procedure with the header `Private Sub btnDisplay_Click(...) Handles btnDisplay.Click`, and having one line for each step. Display each result by assigning it to the `txtOutput.Text` property. Lines that display data should use the given variable names.

35. **Inventor** The following steps give the name and birth year of a famous inventor:
- Declare the variables *firstName*, *middleName*, and *lastName* as String. Declare the variable *yearOfBirth* as type Integer.
 - Assign "Thomas" to the variable *firstName*.
 - Assign "Alva" to the variable *middleName*.
 - Assign "Edison" to the variable *lastName*.
 - Assign 1847 to the variable *yearOfBirth*.
 - Display the inventor's full name followed by a comma and his year of birth.

36. **Price of Ketchup** The following steps compute the price of ketchup:
- Declare the variable *item* as type String. Declare the variables *regularPrice* and *discount* as type Decimal.
 - Assign "ketchup" to the variable *item*.
 - Assign 1.80 to the variable *regularPrice*.
 - Assign .27 to the variable *discount*.
 - Assign to the variable *discountPrice* the expression $1.80 * (1 - .27)$. (Note: The product is 1.53.)
 - Display the sentence "1.53 is the sale price of ketchup."

In Exercises 37 and 38, write a line of code to carry out the task. Specify where in the program the line of code should be placed.

37. Declare the variable *str* as a string variable visible to all parts of the program.
38. Declare the variable *str* as a string variable visible only to the `btnTest_Click` event procedure.

In Exercises 39 through 44, write a program to carry out the stated task.

39. **Distance from a Storm** If *n* is the number of seconds between lightning and thunder, the storm is $n/5$ miles away. Write a program that reads the number of seconds between lightning and thunder and reports the distance of the storm rounded to two decimal places. A sample run is shown in Fig. 3.18.

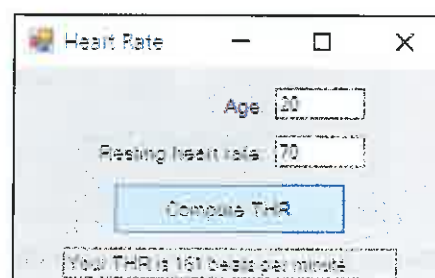
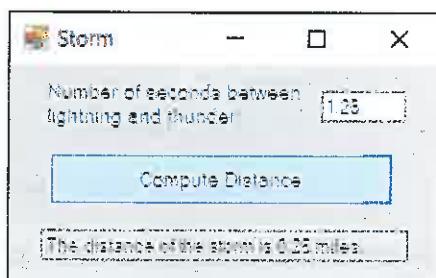


FIGURE 3.18 Possible outcome of Exercise 39. FIGURE 3.19 Possible outcome of Exercise 40.

- 40. Training Heart Rate** The American College of Sports Medicine recommends that you maintain your *training heart rate* during an aerobic workout. Your training heart rate is computed as $.7 * (220 - a) + .3 * r$, where a is your age and r is your resting heart rate (your pulse when you first awaken). Write a program to read a person's age and resting heart rate and display the training heart rate. (Determine *your* training heart rate.) A sample run is shown in Fig. 3.19 on the previous page.
- 41. Triathlon** The number of calories burned per hour by cycling, running, and swimming are 200, 475, and 275, respectively. A person loses 1 pound of weight for each 3500 calories burned. Write code to read the number of hours spent at each activity and then display the number of pounds worked off. A sample run is shown in Fig. 3.20.

Triathlon

Number of hours cycling: 2

Number of hours running: 3

Number of hours swimming: 1

Compute Weight Loss

0.6 pounds were lost

FIGURE 3.20 Possible outcome of Exercise 41.

Electricity

Device: lightbulb

Wattage: 100

Hours used: 720

Price per kWh in cents: 12.65

Calculate Cost

Cost (in dollars): 9.11

FIGURE 3.21 Possible outcome of Exercise 42.

- 42. Cost of Electricity** The cost of the electricity used by a device is given by the formula

$$\text{cost of electricity (in dollars)} = \frac{\text{wattage of device} \cdot \text{hours used} \cdot \text{cost per kWh in cents}}{100,000}$$

where kWh is an abbreviation for “kilowatt hour.” The cost per kWh of electricity varies with locality. On April 1, 2015, the average cost of electricity for a residential customer in the United States was 12.65¢ per kWh. Write a program that allows the user to calculate the cost of operating an electrical device. Figure 3.21 calculates the cost of keeping a light bulb turned on for an entire month.

- 43. Add Times** Write a program to add two times, where the times are given in hours and minutes. See Fig. 3.22. The program should use both integer division and the Mod operator.

Add Times

First time: 3 hours 30 minutes

Second time: 2 hours 20 minutes

Add the Two Times

5 hours and 10 minutes

FIGURE 3.22 Possible outcome of Exercise 43.

Baseball

Team: Yankees

Games won: 84 Games lost: 79

Compute Percentage

The Yankees won 81.9 percent of their games

FIGURE 3.23 Possible outcome of Exercise 44.

- 44. Baseball** Write code to read the name of a baseball team, the number of games won, and the number of games lost, and display the name of the team and the percentage of games won. A sample run is shown in Fig. 3.23.

In the following exercises, write a program to carry out the task. The program should use variables for each of the quantities.

- 45. Income** Request a company's annual revenue and expenses as input, and display the company's net income (revenue minus expenses). See Fig. 3.24.

FIGURE 3.24 Possible outcome of Exercise 45. FIGURE 3.25 Possible outcome of Exercise 46.

- 46. Price-to-Earnings Ratio** Request a company's earnings-per-share for the year and the price of one share of stock as input, and then display the company's price-to-earnings ratio (that is, price/earnings). See Fig. 3.25.

- 47. Car Speed** The formula $s = \sqrt{24d}$ gives an estimate of the speed in miles per hour of a car that skidded d feet on dry concrete when the brakes were applied. Write a program that requests the distance skidded and then displays the estimated speed of the car. See Fig. 3.26.

FIGURE 3.26 Possible outcome of Exercise 47. FIGURE 3.27 Possible outcome of Exercise 48.

- 48. Percentages** Convert a percentage to a decimal. See Fig. 3.27.

- 49. Enumeration** Write a program that contains a button and a read-only text box on the form, with the text box initially containing 100. Each time the button is clicked on, the number in the text box should decrease by 1. See Fig. 3.28.

FIGURE 3.28 Possible outcome of Exercise 49. FIGURE 3.29 Possible outcome of Exercise 50.

- 50. Area Code** Write a program that requests a (complete) phone number in a text box and then displays the area code in another text box when a button is clicked on. See Fig. 3.29.
- 51. Average** Write a program that allows scores to be input one at a time, and then displays the average of the scores upon request. (See Fig. 3.30.) The user should type a score into the top text box and then click on the *Record* button. This process can be repeated as many times as desired. At any time the user should be able to click on the *Calculate* button to display the average of all the scores that were entered so far. **Note:** This program requires two class-level variables.

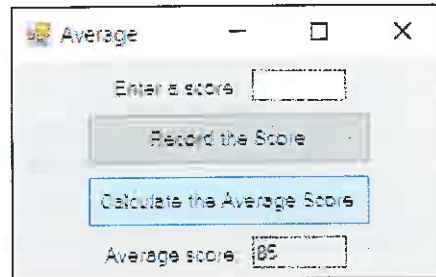


FIGURE 3.30 Possible outcome of Exercise 51.

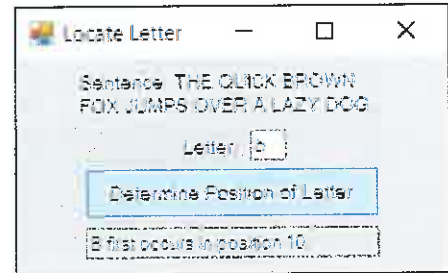


FIGURE 3.31 Possible outcome of Exercise 52.

- 52. Locate a Letter** Write a program that requests a letter, converts it to uppercase, and gives its first position in the sentence "THE QUICK BROWN FOX JUMPS OVER A LAZY DOG." See Fig. 3.31.
- 53. Server's Tip** Calculate the amount of a server's tip, given the amount of the bill and the percentage tip as input. See Fig. 3.32.

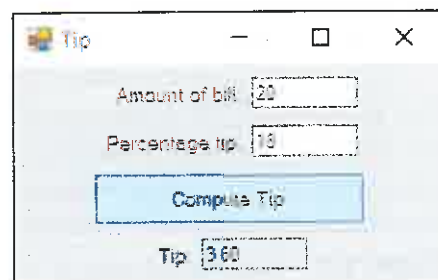


FIGURE 3.32 Possible outcome of Exercise 53.

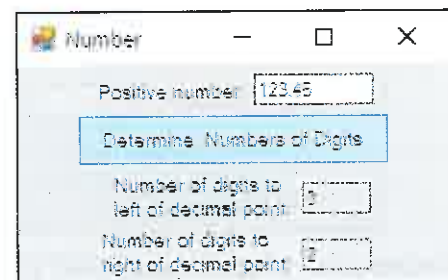


FIGURE 3.33 Possible outcome of Exercise 54.

- 54. Analyze a Number** Write a program that requests a positive number containing a decimal point as input and then displays the number of digits to the left of the decimal point and the number of digits to the right of the decimal point. See Fig. 3.33.
- 55. Word Replacement** Write a program that requests a sentence, a word in the sentence, and another word and then displays the sentence with the first word replaced by the second. In Fig. 3.34 the user responds by typing "What you don't know won't hurt you." into the first text box and "know" and "owe" into the second and third text boxes, and the message "What you don't owe won't hurt you." is displayed.

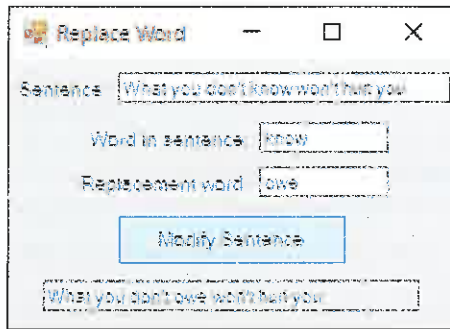


FIGURE 3.34 Possible outcome of Exercise 55.



FIGURE 3.35 Possible outcome of Exercise 56.

- 56. Sales** Write a program that allows furniture sales (item and price) to be displayed in a list box one at a time, and then shows the total commission of the sales (6%) upon request. (See Fig. 3.35.) The user should type each item and price into the text boxes and then click the *Display* button. This process can be repeated as many times as desired. At any time the user should be able to click the *Show* button to display the total commission of all the sales that were entered. **Note:** This program requires a class-level variable.
- 57. Addition** Add an event procedure to Example 2 so that txtSum will be cleared whenever either the first number or the second number is changed.
- 58. Future Value** If P dollars (called the principal) is invested at $r\%$ interest compounded annually, then the future value of the investment after n years is given by the formula

$$\text{future value} = P \left(1 + \frac{r}{100} \right)^n$$

Write a program that calculates the balance of the investment after the user gives the principal, interest rate, and number of years. Figure 3.36 shows that \$1000 invested at 5% interest will grow to \$1,157.63 in 3 years.

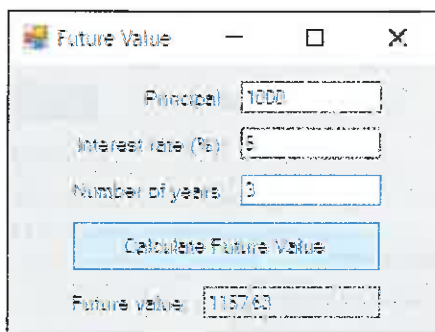


FIGURE 3.36 Possible outcome of Exercise 58.

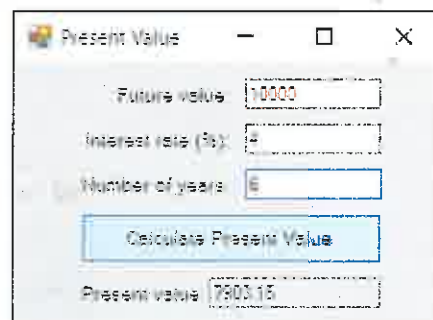


FIGURE 3.37 Possible outcome of Exercise 59.

- 59. Present Value** The present value of F dollars at interest rate $r\%$ compounded annually for n years is the amount of money that must be invested now in order to have F dollars (called the future value) in n years. The formula for present value is

$$\text{present value} = \frac{F}{\left(1 + \frac{r}{100}\right)^n}$$

Write a program that calculates the present value of an investment after the user gives the future value, interest rate, and number of years. Figure 3.37 shows that at 4% interest, \$7,903.15 must be invested now in order to have \$10,000 after 6 years.

- 60. Convert Months** Write a program that allows the user to enter a whole number of months and then converts that amount of time to years and months. See Fig. 3.38. The program should use both integer division and the Mod operator.

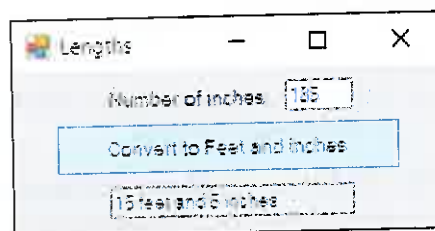
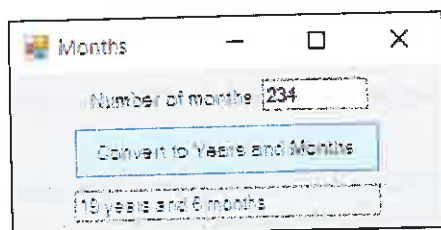


FIGURE 3.38 Possible outcome of Exercise 60. **FIGURE 3.39** Possible outcome of Exercise 61.

- 61. Convert Lengths** Write a program that allows the user to enter a whole number of inches and then converts that length to feet and inches. See Fig. 3.39. The program should use both integer division and the Mod operator.

- 62. Equivalent Interest Rates** Interest earned on municipal bonds from an investor's home state is not taxed, whereas interest earned on CDs is taxed. Therefore, in order for a CD to earn as much as a municipal bond, the CD must pay a higher interest rate. How much higher the interest rate must be depends on the investor's tax bracket. Write a program that allows the user to enter a tax bracket and a municipal bond interest rate and then finds the CD interest rate with the same yield. See Fig. 3.40. **Note:** If the tax bracket is expressed as a decimal, then

$$\text{CD interest rate} = \frac{\text{municipal bond interest rate}}{(1 - \text{tax bracket})}$$

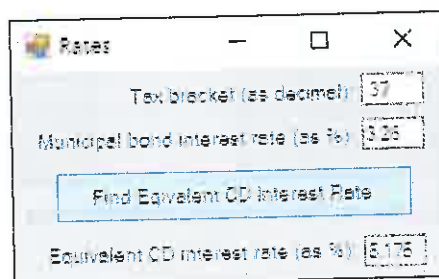


FIGURE 3.40 Possible outcome of Exercise 62.

Solutions to Practice Problems 3.2

1. -1. There is no uppercase letter *E* in the string "Computer". `IndexOf` distinguishes between uppercase and lowercase.
2. The first statement displays 16 in the text box, whereas the second statement displays 88. With `Option Strict` in effect, the first statement would not be valid if `CStr` were missing, since `8 + 8` is a number and `txtBox.Text` is a string. Visual Basic treats the second statement as if it were

```
txtBox.Text = CStr(8) & CStr(8)
```

3. Some possibilities are

PROHIBITED STATEMENT	SUGGESTION FOR FIXING
<code>Dim x As Double = "23"</code>	Replace "23" with <code>Cdbl("23")</code> .
<code>dblVar = txtBox.Text</code>	Replace <code>txtBox.Text</code> with <code>Cdbl(txtBox.Text)</code> .
<code>dblVar = 2 & 3</code>	Replace <code>2 & 3</code> with <code>Cdbl(2 & 3)</code> .

4. In the third line, `15 + 1.25` has type `Double` and therefore cannot be assigned to a variable of type `Decimal`. The line should be replaced with either `newPrice = 15 + 1.25D` or `newPrice = CDec(15 + 1.25)`. *Note:* The second line is fine since `15` is of type `Integer` and an `Integer` value *can* be assigned to a `Decimal` variable.

3.3 Input and Output

■ Formatting Numeric Output

So far, we have relied on the `CStr` function to convert numbers into strings. The `ToString` method does the same job, and has the added feature that it allows numbers to be displayed in familiar formats. For instance, it allows a number to be displayed with thousands separators and a specified number of decimal places, allows an amount of money to be displayed with a dollar sign, and allows a percentage to be displayed with a percent symbol.

The value of

```
variableName.ToString(formatString)
```

where *formatString* might be "N" for a number, "C" for an amount of money (Currency), and "P" for a percent is a suitably formatted string representation of the value of the variable. The value is given rounded to two decimal places. The number of decimal places can be changed by following the letter with the number.

The variable in the aforementioned `ToString` expression can be replaced by a literal number or an expression involving literals. Expressions involving literals must be surrounded by parentheses. For clarity, we will surround all literals and expressions with parentheses. Table 3.3 gives some examples of the use of the `ToString` method.

TABLE 3.3 Some formatted values.

Expression	Value
<code>(12345.676).ToString("N")</code>	12,345.68
<code>(12345.676D).ToString("C")</code>	\$12,345.68
<code>(0.185).ToString("P")</code>	18.50 %
<code>(1 + 0.5^3).ToString("N3")</code>	1.125
<code>(-1000D).ToString("C0")</code>	(\$1,000)
<code>(50/75).ToString("P4")</code>	66.6667 %

Notice that the currency format string "C" uses the accountant's convention of denoting negative amounts by surrounding them with parentheses.

Dates as Input and Output

So far, all input and output has been either numbers or strings. However, applications sometimes require dates as input and output. Visual Basic has a **Date** data type and a **Date** literal.

A variable of type **Date** is declared with a statement of the form

```
Dim varName As Date
```

Just as string literals are written surrounded by quotation marks, date literals are written surrounded by number signs. For instance, the statement

```
Dim dayOfIndependence As Date = #7/4/1776#
```

declares a date variable and assigns a value to it.

The function **CDate** converts a string to a date. For instance, the statement

```
Dim dt As Date = CDate(txtBox.Text)
```

assigns the contents of a text box to a variable of type **Date**.

Dates can be formatted with the **ToString** method. If *dateVar* is a variable of type **Date**, then the value of

```
dateVar.ToString("D")
```

is a string consisting of the date specified by *dateVar* with the day of the week and the month spelled out. For instance, the two lines of code

```
Dim dayOfIndependence As Date = #7/4/1776#
txtBox.Text = dayOfIndependence.ToString("D")
```

display Thursday, July 04, 1776 in the text box. **Note:** If "D" is replaced with "d", 7/4/1776 will be displayed in the text box.

There are many functions involving dates. Several of them are listed in Table 3.4.

TABLE 3.4 Some date functions.

Function	Output
Today	current date (such as 7/14/2016)
dt.Day	1, 2, . . . , 31
dt.Month	1, 2, . . . , 12
dt.Year	such as 2016, 2017, etc.
dt.ToString("dddd")	Sunday, Monday, . . . , Saturday
dt.ToString("MMMM")	January, February, . . . , December
DateDiff(DateInterval.Day, dt1, dt2)	number of days between the two dates

Note: The way certain dates are formatted depends on the *Home location* setting in Windows. The examples in this book use the United States location setting and therefore the value of *Today* is displayed in the form mm/dd/yyyy. With the Italian location setting, for example, the value of *Today* is displayed in the form dd/mm/yyyy and days and months have names such as Martedì and Aprile.

AddYears is a useful method for working with dates. If *dt* is a variable of type **Date**, and *n* is an integer, then the value of **dt.AddYears(n)** is the value of *dt* advanced by *n* years. When *n* is negative, the value of *dt* is decreased. Two similar methods are **AddDays** and **AddMonths**.

`DateDiff(DateInterval.Year, d1, d2)` gives the number of years (sort of) between the two dates. It must be employed with care since it only uses the year parts of the two dates.

■ Using a Masked Text Box for Input

Problems can arise when the wrong type of data are entered as input into a text box. For instance, if the user replies to the request for an age by entering “twenty-one” into a text box, the program can easily crash. Sometimes this type of predicament can be avoided by using a masked text box for input. (In later chapters, we will consider other ways of ensuring the integrity of input.)

In the Toolbox, the icon for the MaskedTextBox control [MaskedTextBox] consists of a pair of parentheses containing a period and followed by a hyphen. The most important property of a masked text box is the `Mask` property that is used to restrict the characters entered into the box. Also, the `Mask` property is used to show certain characters in the control—to give users a visual cue that they should be entering a phone number or a social security number, for example. Some possible settings for the `Mask` property are shown in Table 3.5. The first four settings can be selected from a list of specified options. The last three settings generalize to any number of digits, letters, or ampersands. If the `Mask` property is left blank, then the MaskedTextBox control is nearly identical to the TextBox control.

TABLE 3.5 Some settings for the `Mask` property.

Setting	Effect
000-00-0000	The user can enter a social security number.
000-0000	The user can enter a phone number (without an area code).
(000)000-0000	The user can enter a phone number (with an area code).
00/00/0000	The user can enter a date.
0000000	The user can enter a positive integer consisting of up to 7 digits.
LLLLL	The user can enter a string consisting of up to 5 letters.
&&&&&&&&	The user can enter a string consisting of up to 8 characters.

Suppose a form contains a masked text box whose `Mask` property has the setting 000-00-0000. When the program is run, the string “__-__-__” will appear in the masked text box. The user will be allowed to type a digit in place of each of the nine underscore characters. The hyphens cannot be altered, and no characters can be typed anywhere else in the masked text box.

At run time, the characters 0, L, and & in the setting for a `Mask` property are replaced by underscore characters that are place holders for digits, letters, and characters, respectively. (Spaces are also allowed. However, trailing spaces are dropped.) When the characters “-”, “(”, “)”, or “/” appear in a setting for a `Mask` property, they appear as themselves in the masked text box and cannot be altered. There are some other mask settings, but these seven will suffice for our purposes.

Figure 3.41(a) shows a masked text box during design time. It looks like an ordinary text box. However, the *Tasks* button for the masked text box is used to set the `Mask` property rather than the `Multiline` property. Figure 3.41(b) on the next page shows the result of clicking on the *Tasks* button. Then, clicking on “Set Mask. . .” brings up the Input Mask dialog box shown in Figure 3.42. (This input dialog box is the same input dialog box that is invoked when you click on the ellipsis in the `Mask` property’s Settings box.) You can use this input dialog box to select a commonly used value for the `Mask` property, or create your own customized mask in the Mask text box. To produce the settings 00/00/0000 and 000-00-0000, click on “Short date” and “Social security number”, respectively. We use the prefix *mtb* for the names of masked text boxes.



FIGURE 3.41 The Masked TextBox control.

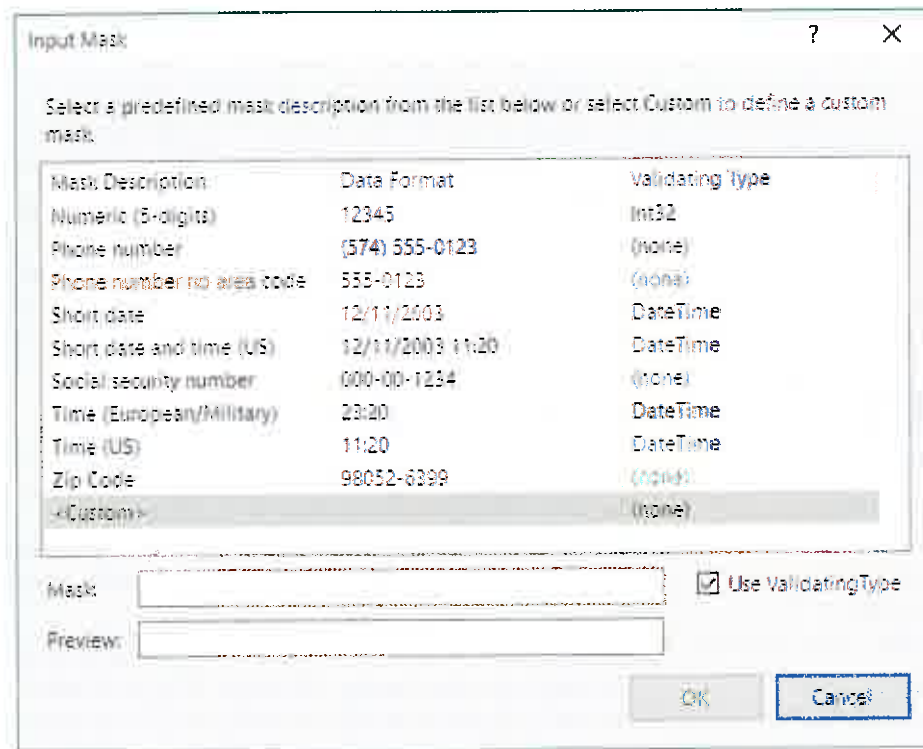
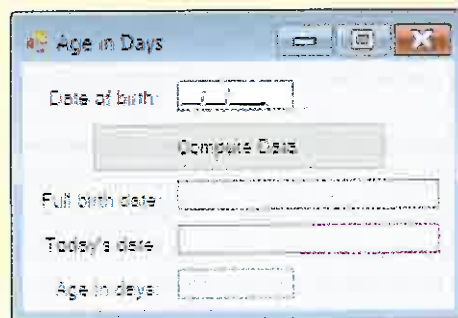


FIGURE 3.42 Input dialog box used to set the Mask property of a masked text box.

**Example 1****Age in Days**

The following program gives information pertaining to a date input by the user. The mask for the masked text box can be set by clicking on *Short date* in the Input Mask dialog box.



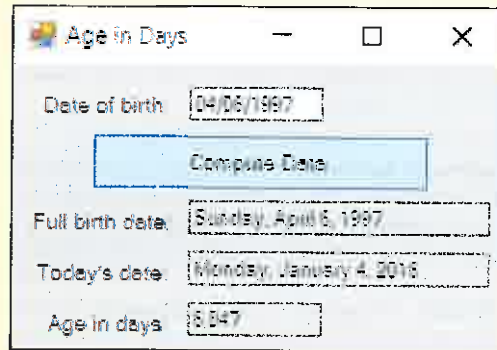
OBJECT	PROPERTY	SETTING
frmAge	Text	Age in Days
lblDayOfBirth	Text	Date of birth:
mtbDayOfBirth	Mask	00/00/0000
btnComputeOfBirth	Text	Compute Data
lblFullDateOfBirth	Text	Full birth date:
txtFullDateOfBirth	ReadOnly	True
lblToday	Text	Today's date:
txtToday	ReadOnly	True
lblAgeInDays	Text	Age in days:
txtAgeInDays	ReadOnly	True


```

Private Sub btnCompute_Click(...) Handles btnCompute.Click
    Dim dob As Date = CDate(mtbDayOfBirth.Text)
    txtFullDateOfBirth.Text = dob.ToString("D")
    txtToday.Text = Today.ToString("D")
    txtAgeInDays.Text = DateDiff(DateInterval.Day, dob, Today).ToString("N0")
End Sub

```

[Run, enter your birthday, and click on the button. One possible outcome is the following.]



Getting Input from an Input Dialog Box

Normally, a text box is used to obtain input, where the type of information requested is specified in a label adjacent to the text box. Sometimes, we want just one piece of input and would rather not have a text box and label stay on the form permanently. The problem can be solved with an **input dialog box**. When a statement of the form

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

is executed, an input dialog box similar to the one shown in Fig. 3.43 pops up on the screen. After the user types a response into the text box at the bottom of the dialog box and presses the Enter key (or clicks on the OK button), the response is assigned to the string variable. The *title* argument is optional and provides the text that appears in the title bar. The *prompt* argument is a string that tells the user what information to type into the text box.

When you type the opening parenthesis following the word `InputBox`, the Code Editor displays a line containing the general form of the `InputBox` statement. See Fig. 3.44. This feature of IntelliSense is called **Parameter Info**. Optional parameters are surrounded by square brackets. All the parameters in the general form of the `InputBox` statement are optional except for *prompt*.

The response typed into an input dialog box is treated as a single string value, no matter what is typed. (Quotation marks are not needed and, if included, are considered as part of the string.) Numeric data typed into an input dialog box should be converted to a number with `CDbl` or `CInt` before being assigned to a numeric variable or used in a calculation. Just as with a text box, the typed data must be literals. They cannot be variables or expressions. For instance, *num*, $1/2$, and $2 + 3$ are not acceptable.



VideoNote
Input Boxes
and Message
Boxes

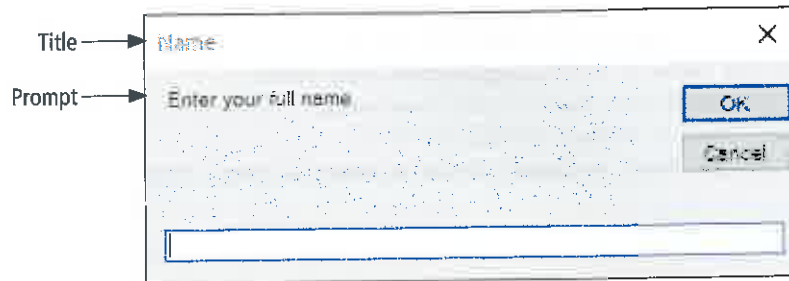


FIGURE 3.43 Sample input dialog box.

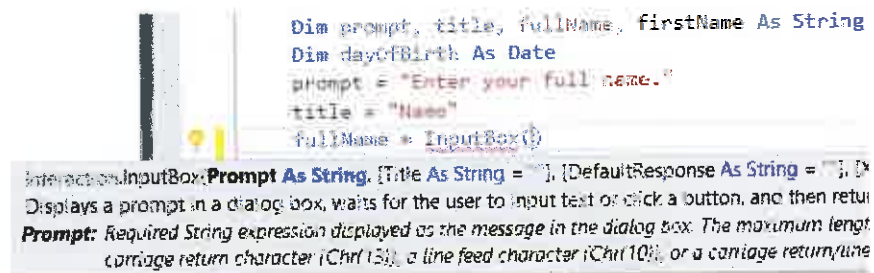


FIGURE 3.44 Parameter Info feature of IntelliSense.

**Example 2****Age in Days** The following program uses two InputBox functions.

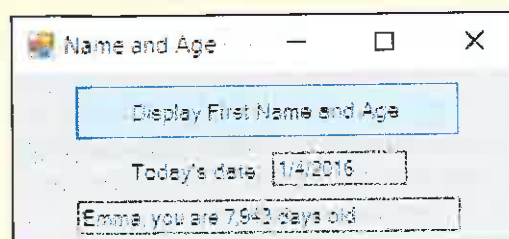
Whenever an InputBox function is encountered in a program, an input dialog box appears, and execution stops until the user responds to the request. The function returns the value entered into the input dialog box.

```

Private Sub btnDisplay_Click(...) Handles btnDisplay.Click
    Dim prompt, title, fullName, firstName As String
    Dim dateOfBirth As Date
    prompt = "Enter your full name."
    title = "Name"
    fullName = InputBox(prompt, title)
    firstName = fullName.Substring(0, fullName.IndexOf(" "))
    prompt = "Enter your date of birth."
    title = "Birthday"
    dateOfBirth = CDate(InputBox(prompt, title))
    txtToday.Text = CStr(Today)
    txtOutput.Text = firstName & ", you are " &
        DateDiff(DateInterval.Day, dateOfBirth, Today).ToString("N0") & " days old."
End Sub

```

[Run, click on the button, enter Emma Smith into the first input dialog box, and enter 4/6/1994 into the second input dialog box.]



Using a Message Dialog Box for Output

Sometimes you want to grab the user's attention with a brief message such as "Correct" or "Nice try, but no cigar." You want this message to appear on the screen only until the user has read it. This task is easily accomplished with a **message dialog box** such as the one shown in Fig. 3.45.

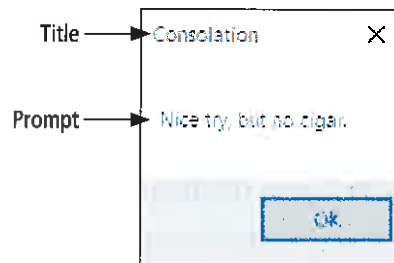


FIGURE 3.45 Sample message dialog box.

When a statement of the form

```
MessageBox.Show(prompt, title)
```

is executed, where *prompt* and *title* are strings, a message dialog box appears with *prompt* displayed and title bar caption *title*, and stays on the screen until the user presses Enter, clicks on the Close button in the upper-right corner, or clicks OK. For instance, the statement

```
MessageBox.Show("Nice try, but no cigar.", "Consolation")
```

produces the output in Fig. 3.45. You can omit the value for the argument *title* and just execute **MessageBox.Show(prompt)**. If you do, the title bar will be blank and the rest of the message dialog box will appear as before.

Named Constants

Often a program uses a special constant whose value does not change during program execution. Some examples might be the minimum wage, the sales tax rate, and the name of a master file. Programs are often made easier to understand and maintain if such a constant is given a name. Visual Basic has an object, called a **named constant**, that serves this purpose. A named constant is declared and used in a manner similar to a variable. The two main differences are that in the declaration of a named constant, **Dim** is replaced with **Const**, and the value of the named constant cannot be changed elsewhere in the program. Named constants can be thought of as read-only variables.

A named constant is declared and assigned a value with a statement of the form

```
Const CONSTANT_NAME As DataType = value
```

The standard convention is that the names be written in uppercase letters with words separated by underscore characters. Like a **Dim** statement, a **Const** statement can be placed in the Declarations section of a class (for class scope) or in a procedure (for procedure scope). Named constant declarations in procedures usually are placed near the beginning of the procedure. Some examples of named constant declarations are

```
Const INTEREST_RATE As Double = 0.04  
Const PI As Double = 3.1416  
Const BOOK_TITLE As String = "Programming with VB2015"
```

Examples of statements using these named constants are

```
interestEarned = INTEREST_RATE * Cdbl(txtAmount.Text)
circumference = 2 * PI * radius
MessageBox.Show(BOOK_TITLE, "Title of Book")
```

Although the value of a named constant such as **INTEREST_RATE** will not change during the execution of a program, the value may need to be changed at a later time. The programmer can adjust to this change by altering just one line of code instead of searching through the entire program for each occurrence of the old interest rate.

■ Formatting Output with Zones (Optional)

Data can be displayed in tabular form in a list box. In order to have the items line up nicely in columns, you must

1. use a fixed-width font such as Courier New so that each character will have the same width.
2. divide the line into zones with a format string.

Figure 3.46 shows a line of a list box divided into zones of widths 15 characters, 10 characters, and 8 characters. The leftmost zone is referred to as zone 0, the next zone is zone 1, and so on. These zones are declared in a string with the statement

```
Dim fmtStr As String = "{0, 15}{1, 10}{2, 8}"
```

Note: The pairs of numbers are surrounded by curly brackets, not parentheses.

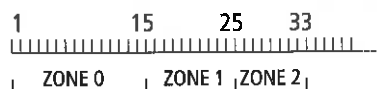


FIGURE 3.46 Zones.

If *data0*, *data1*, and *data2* are strings or numbers, the statement

```
lstOutput.Items.Add(String.Format(fmtStr, data0, data1, data2))
```

displays the pieces of data right justified into the zones. If any of the width numbers (such as 15, 10, or 8) is preceded with a minus sign, the data placed into the corresponding zone will be left justified.



Example 3

The following program displays information about two colleges in the United States. **Note:** The endowments are in billions of dollars. The final column tells what fraction of the student body graduates.



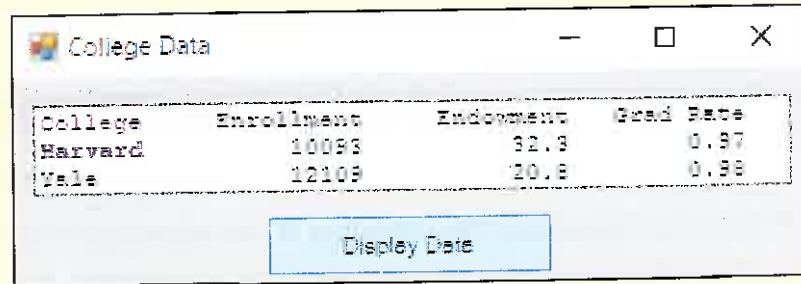
OBJECT	PROPERTY	SETTING
frmColleges	Text	College Data
lstColleges	Font	Courier New
btnDisplay	Text	Display Table

```

Private Sub btnDisplay_Click(...) Handles btnDisplay.Click
    Dim fmtStr As String = "{0,-10}{1,12}{2,14}{3,13}"
    lstColleges.Items.Clear()
    lstColleges.Items.Add(String.Format(fmtStr, "College",
                                         "Enrollment", "Endowment", "Grad Rate"))
    lstColleges.Items.Add(String.Format(fmtStr, "Harvard", 10093, 32.3, 0.97))
    lstColleges.Items.Add(String.Format(fmtStr, "Yale", 12109, 20.8, 0.98))
End Sub

```

[Run, and click on the button.]



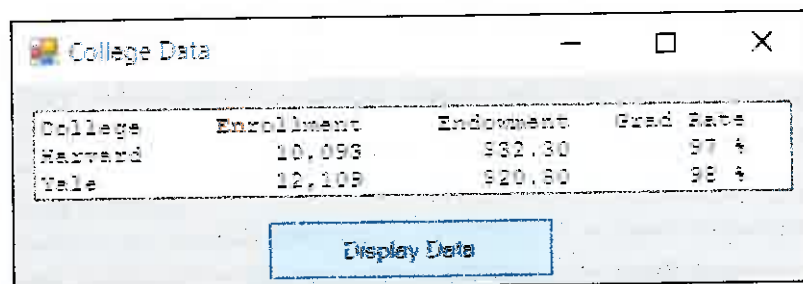
There is no limit to the number of zones, and the zones can be of any widths. In addition, by placing a colon and formatting symbols after the width, you can instruct Visual Basic to specially format numeric data. (String data are not affected.) The most used formatting symbols consist of a letter (N for number, C for currency, or P for percent) followed by a digit specifying the number of decimal places. If there is no digit after the letter, two decimal places are displayed. Here are some examples of such formatting.

ZONE FORMAT TERM	NUMBER TO BE FORMATTED	NUMBER DISPLAYED
{1, 12: N3}	1234.5679	1,234.568
{1, 12: N0}	34.6	35
{1, 12: C1}	1234.567	\$1,234.6
{1, 12: P}	0.569	56.90%

If the second line of the program in Example 1 is replaced with

```
Dim fmtStr As String = "{0,-10}{1,12:N0}{2,14:C}{3,12:PC}"
```

the output will be as follows.



The format strings considered so far consist of a sequence of pairs of curly brackets. We also can insert spaces between successive pairs of brackets. If so, the corresponding zones in the output will be separated by those spaces. The lines of code

```
Dim fmtStr As String = "{0, 5} {1, -5}" 'Two spaces after the
                                         first right curly bracket
lstOutput.Items.Add("12345678901234567890")
lstOutput.Items.Add(String.Format(fmtStr, 1, 2))
```

produce the output

```
12345678901234567890
  1  2
```

Practice Problems 3.3

1. What is the difference between `(12.345).ToString` and `CStr(12.345)`?
2. What is the difference in the outcomes of the following two sets of code?

```
strVar = InputBox("How old are you?", "Age")
numVar = CDb1(strVar)
txtOutput.Text = numVar
```

```
numVar = CDb1(InputBox("How old are you?", "Age"))
txtOutput.Text = numVar
```

EXERCISES 3.3

In Exercises 1 through 56, determine the output produced by the lines of code.

1. `txtOutput.Text = (1234.56).ToString("N0")`
2. `txtOutput.Text = (-12.3456).ToString("N3")`
3. `txtOutput.Text = (1234).ToString("N1")`
4. `txtOutput.Text = (12345).ToString("N")`
5. `txtOutput.Text = (0.012).ToString("N1")`
6. `txtOutput.Text = (5 * (10 ^ -2)).ToString("N1")`
7. `txtOutput.Text = (-2 / 3).ToString("N")`
8. `Dim numVar As Double = Math.Round(1.2345, 1)`
`txtOutput.Text = numVar.ToString("N")`
9. `Dim numVar As Double = Math.Round(12345.9)`
`txtOutput.Text = numVar.ToString("N3")`
10. `Dim numVar As Double = Math.Round(12.5)`
`txtOutput.Text = numVar.ToString("N0")`
11. `Dim numVar As Double = Math.Round(11.5)`
`txtOutput.Text = numVar.ToString("N0")`
12. `txtOutput.Text = (1234.5D).ToString("C")`
13. `txtOutput.Text = (12345.67D).ToString("C0")`
14. `txtOutput.Text = (-1234567D).ToString("C")`
15. `txtOutput.Text = (-0.225D).ToString("C")`


```

16. txtOutput.Text = (32 * (10 ^ -2)).ToString("C")
17. txtOutput.Text = (4D / 5D).ToString("C")
18. txtOutput.Text = (0.04D).ToString("C0")
19. txtOutput.Text = (0.075D).ToString("C")
20. txtOutput.Text = (-0.05D).ToString("C3")
21. txtOutput.Text = (1).ToString("P")
22. txtOutput.Text = (0.01).ToString("P")
23. txtOutput.Text = (2 / 3).ToString("P")
24. txtOutput.Text = (3 / 4).ToString("P1")
25. txtOutput.Text = "Pay to France " & (27267622D).ToString("C")
26. txtOutput.Text = "Manhattan was purchased for " & (24D).ToString("C")
27. Dim popUSover24 As Double = 177.6      'Million
    Dim collegeGrads As Double = 45.5      'Million
    '45.5/177.6 = 0.2561937
    txtOutput.Text = (collegeGrads / popUSover24).ToString("P1") &
        " of the U.S. population 25+ years old are college graduates."
28. Dim degrees As String = (1711500).ToString("N0")
    txtOutput.Text = degrees & " degrees were conferred."
29. txtOutput.Text = "The likelihood of Heads is " &
    (1 / 2).ToString("P0")
30. txtOutput.Text = "Pi = " & (3.1415926536).ToString("N4")
31. txtOutput.Text = CStr(#10/23/2015#)
32. Dim dt As Date = #6/18/2017# 'Father's Day
    txtOutput.Text = dt.ToString("D")
33. Dim dt As Date = #11/24/2016# 'Thanksgiving Day
    txtOutput.Text = dt.ToString("D")
34. Dim dt As Date = #1/1/2000#
    txtOutput.Text = CStr(dt.AddYears(15))
35. Dim dt As Date = #9/29/2017#
    txtOutput.Text = CStr(dt.AddDays(3))
36. Dim dt As Date = #10/9/2015#
    txtOutput.Text = CStr(dt.AddMonths(4))
37. Dim dt As Date = #4/5/2017#
    txtOutput.Text = CStr(dt.AddYears(2))
38. Dim dt As Date = #10/1/2015#
    txtOutput.Text = CStr(dt.AddDays(32))
39. Dim dt1 As Date = #2/1/2016# '2016 was a leap year
    Dim dt2 As Date = dt1.AddMonths(1)
    txtOutput.Text = CStr(DateDiff(DateInterval.Day, dt1, dt2))
40. Dim dt1 As Date = #1/1/2016# '2016 was a leap year
    Dim dt2 As Date = #1/1/2017#
    txtOutput.Text = CStr(DateDiff(DateInterval.Day, dt1, dt2))
41. Dim dt1 As Date = Today
    Dim dt2 As Date = dt1.AddDays(5)
    txtOutput.Text = CStr(DateDiff(DateInterval.Day, dt1, dt2))

```

42. `Dim dt As Date = #1/2/2016#`
`txtOutput.Text = CStr(dt.Day)`
43. `Dim dt As Date = #1/2/2016#`
`txtOutput.Text = CStr(dt.Month)`
44. `Dim dt As Date = #1/2/2016#`
`txtOutput.Text = CStr(dt.Year)`
45. `Dim dt As Date = #1/2/2016#`
`MessageBox.Show(CStr(dt.Day + dt.Year))`
46. `Dim dt As Date = #1/2/2016#`
`MessageBox.Show("1, 2, " & CStr(dt.Day + dt.Month))`
47. `Dim bet As Decimal 'amount bet at roulette`
`bet = CDec(InputBox("How much do you want to bet?", "Wager"))`
`txtOutput.Text = "You might win " & 36 * bet & " dollars."`
 (Assume that the response is 10.)
48. `Dim word As String`
`word = InputBox("Word to negate:", "Negatives")`
`txtOutput.Text = "un" & word`
 (Assume that the response is *tied*.)
49. `Dim lastName, message, firstName As String`
`lastName = "Jones"`
`message = "What is your first name Mr. " & lastName & "?"`
`firstName = InputBox(message, "Name")`
`txtOutput.Text = "Hello " & firstName & " " & lastName`
 (Assume that the response is *John*.)
50. `Dim intRate, doublingTime As Decimal 'interest rate, time to double`
`intRate = CDec(InputBox("Current interest rate?", "Interest"))`
`doublingTime = 72D / intRate`
`lstOutput.Items.Add("At the current interest rate, money will")`
`lstOutput.Items.Add("double in " & doublingTime & " years.")`
 (Assume that the response is 4.)
51. `Const SALES_TAX_RATE As Decimal = 0.06D`
`Dim price As Decimal = 100`
`Dim cost = (1 + SALES_TAX_RATE) * price`
`txtOutput.Text = cost.ToString("C")`
52. `Const ESTATE_TAX_EXEMPTION As Double = 1000000`
`Const TAX_RATE As Decimal = 0.45D`
`Dim valueOfEstate As Decimal = 3000000`
`Dim tax As Double = TAX_RATE * (valueOfEstate - ESTATE_TAX_EXEMPTION)`
`txtOutput.Text = "You owe " & tax.ToString("C") & " in estate taxes."`

In Exercises 53 through 56, determine the output produced by the lines of code where Courier New is the font setting for the list box.

53. `Dim fmtStr As String = "{0,-13}{1,-10}{2,-7:N0}"`
`lstOutput.Items.Add("123456789012345678901234567890")`
`lstOutput.Items.Add(String.Format(fmtStr, "Mountain", "Place", "Ht (ft)"))`
`lstOutput.Items.Add(String.Format(fmtStr, "K2", "Kashmir", 28250))`

- ```

54. Dim fmtStr As String = "{0,11} {1,-11}" 'Three spaces
 lstOutput.Items.Add("12345678901234567890")
 lstOutput.Items.Add(String.Format(fmtStr, "College", "Mascot"))
 lstOutput.Items.Add(String.Format(fmtStr, "Univ. of MD", "Terrapins"))
 lstOutput.Items.Add(String.Format(fmtStr, "Duke", "Blue Devils"))

55. 'Elements in a 150 Pound Person
 Dim fmtStr As String = "{0,-7} {1,-7:N1} {2,-7:P1}" 'Two spaces
 lstOutput.Items.Clear()
 lstOutput.Items.Add("12345678901234567890")
 lstOutput.Items.Add(String.Format(fmtStr, "Element", "Weight", "Percent"))
 lstOutput.Items.Add(String.Format(fmtStr, "Oxygen", 97.5, 97.5 / 150))
 lstOutput.Items.Add(String.Format(fmtStr, "Carbon", 27, 27 / 150))

56. Dim fmtStr As String = "{0,10} {1,-10:C0}" 'Three spaces
 lstOutput.Items.Clear()
 lstOutput.Items.Add("12345678901234567890")
 lstOutput.Items.Add(String.Format(fmtStr, "", "Tuition"))
 lstOutput.Items.Add(String.Format(fmtStr, "College", "& Fees"))
 lstOutput.Items.Add(String.Format(fmtStr, "Stanford", 42690))
 lstOutput.Items.Add(String.Format(fmtStr, "Harvard", 42292))

```

In Exercises 57 through 64, identify any errors.

- ```

57. Const n As Integer = 5
    n += 1
    txtOutput.Text = CStr(n)

58. Const n As String = "abc"
    n = n.ToUpper
    txtOutput.Text = n

59. Dim num As Double
    num = InputBox("Pick a number from 1 to 10.")
    txtOutput.Text = "Your number is " & num

60. info = InputBox()

61. Dim num As Double = (123456).ToString("N")
    lstOutput.Items.Add(num)

62. txtOutput.Text = ($1234).ToString("C")

63. MessageBox("Olive Kitteridge", "Pulitzer Prize for Fiction")

64. MessageBox.Show(1776, "Year of Independence")

```

In Exercises 65 through 70, give a setting for the Mask property of a masked text box used to input the stated information.

65. A number from 0 to 999.
66. A word of at most ten letters.
67. A Maryland license plate consisting of three letters followed by three digits. (Example: BHC365)
68. A license plate consisting of a digit followed by three letters and then three digits. (Example: 7BHC365)

69. An ISBN number. [Every book is identified by a ten-character International Standard Book Number (ISBN). The first nine characters are digits and the last character is either a digit or the letter X.] (Example: 0-32-108599-X)

70. A two-letter state abbreviation. (Example: CA)

In Exercises 71 and 72, write a statement to carry out the task.

71. Pop up a message dialog box with “Good Advice” in the title bar and the message “First solve the problem. Then write the code.”

72. Pop up a message dialog box with “Taking Risks Proverb” in the title bar and the message “You can’t steal second base and keep one foot on first.”

In Exercises 73 and 74, write an event procedure with the header `Private Sub btnDisplay_Click(...)` Handles `btnDisplay.Click`, and having one, two, or three lines for each step. Lines that display data should use the given variable names.

73. Inflation The following steps calculate the percent increase in the cost of a typical grocery basket of goods:

- Declare the variables *begOfYearCost*, *endOfYearCost*, and *percentIncrease* as type Decimal.
- Assign 200 to the variable *begOfYearCost*.
- Request the cost at the end of the year with an input dialog box, and assign it to the variable *endOfYearCost*.
- Assign $(\text{endOfYearCost} - \text{begOfYearCost}) / \text{begOfYearCost}$ to the variable *percentIncrease*.
- Display a sentence giving the percent increase for the year.

74. Walk-a-Thon The following steps calculate the amount of money earned in a walk-a-thon:

- Declare the variables *pledge* and *miles* as type Decimal.
- Request the amount pledged per mile from an input dialog box, and assign it to the variable *pledge*.
- Request the number of miles walked from an input dialog box, and assign it to the variable *miles*.
- Display a sentence giving the amount to be paid.

75. Reminder Design a form with two text boxes labeled “Name” and “Phone number”. Then write an event procedure that shows a message dialog box stating “Be sure to include the area code!” when the second text box receives the focus. See Fig. 3.47.

 A screenshot of a Windows application window titled "Reminder". The window has a standard Windows title bar with minimize, maximize, and close buttons. Inside the window, there are two text input fields. The first field is labeled "Name" and the second field is labeled "Phone number".

FIGURE 3.47 Form for Exercise 75.

 A screenshot of a Windows application window titled "Count Days". The window has a standard Windows title bar. Inside, there is a button labeled "Calculate Number of Days". Below the button is a text box displaying the numerical value "57.478".

FIGURE 3.48 Possible outcome of Exercise 76.

76. Declaration of Independence Write a program that calculates the number of days since the Declaration of Independence was ratified (7/4/1776). See Fig. 3.48.

77. Length of Year Write a program that requests a year in a masked text box and then displays the number of days in the year. See Fig. 3.49. *Hint:* Use the AddYears method and the DateDiff function.

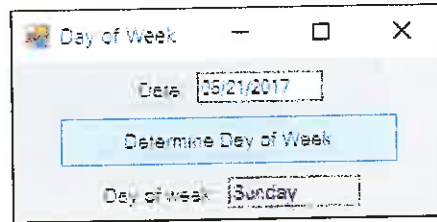
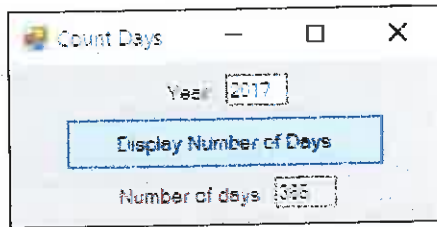


FIGURE 3.49 Possible outcome of Exercise 77. **FIGURE 3.50** Possible outcome of Exercise 78.

78. Day of Week Write a program that requests a date in a masked text box, and then displays the day of the week (such as Sunday, Monday, . . .) for that date. See Fig. 3.50.

79. Day of Week Write a program that requests a date as input and then displays the day of the week (such as Sunday, Monday, . . .) for that date ten years hence. See Fig. 3.51.

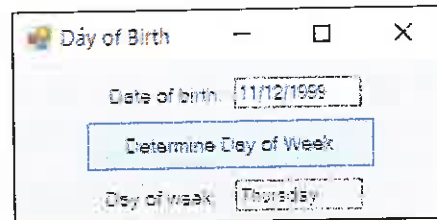


FIGURE 3.51 Possible outcome of Exercise 79. **FIGURE 3.52** Possible outcome of Exercise 80.

80. Day of Birth Write a program that requests the user's date of birth and then displays the day of the week (such as Sunday, Monday) on which the user will have (or had) their 21st birthday. See Fig. 3.52.

81. Convert Date Formats Dates in the U.S. are written in the format month/day/year, whereas dates in Europe are written in the format day/month/year. Write a program to convert a date from U.S. format to European format. See Fig. 3.53. Use the Day and Month methods for variables of type Date.

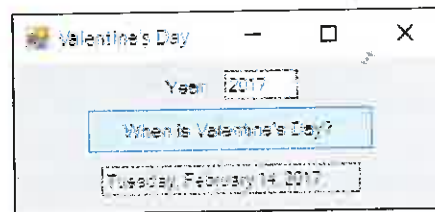
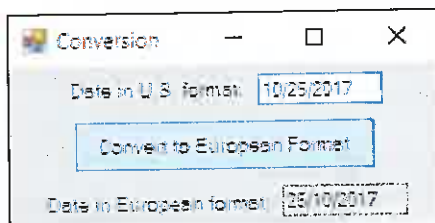


FIGURE 3.53 Possible outcome of Exercise 81. **FIGURE 3.54** Possible outcome of Exercise 82.

82. Valentine's Day Write a program to determine the full date of Valentine's Day for a year input by the user. See Fig. 3.54. *Note:* Valentine's Day is always celebrated on the 14th of February.

- 83. Length of Month** Write a program that requests a month and a year as input and then displays the number of days in that month. **Hint:** Use the AddMonths method. See Fig. 3.55.

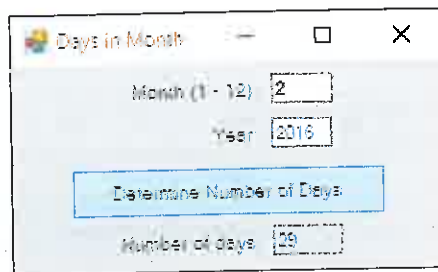


FIGURE 3.55 Possible outcome of Exercise 83.

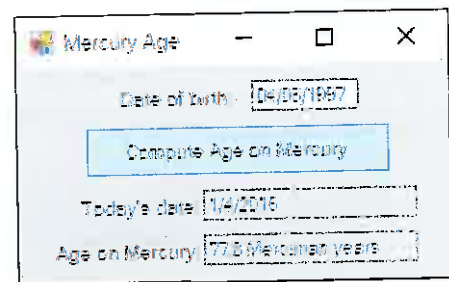


FIGURE 3.56 Possible outcome of Exercise 84.

- 84. How Old Would You Be on Mercury?** The length of a Mercurian year is 88 Earth days. Write a program that requests your date of birth and computes your Mercurian age. See Fig. 3.56.

- 85. Change in Salary** A common misconception is that if you receive a 10% pay raise and later a 10% pay cut, your salary will be unchanged. Write a program that requests a salary as input and then calculates the salary after receiving a 10% pay raise followed by a 10% pay cut. The program also should calculate the percentage change in salary. See Fig. 3.57.

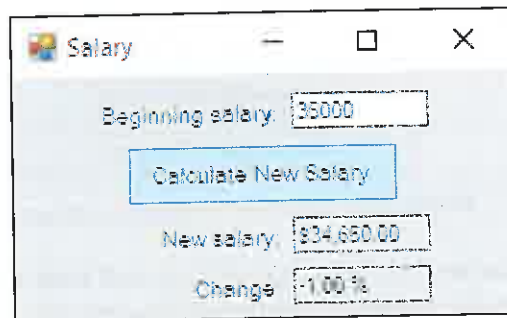


FIGURE 3.57 Possible outcome of Exercise 85.

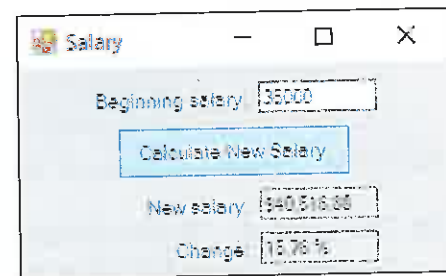


FIGURE 3.58 Possible outcome of Exercise 86.

- 86. Change in Salary** A common misconception is that if you receive three successive 5% pay raises, then your original salary will have increased by 15%. Write a program that requests a salary as input and then calculates the salary after receiving three successive 5% pay raises. The program also should calculate the percentage change in salary. See Fig. 3.58.

- 87. Taxi Fare** If a taxi ride costs \$1 for the first $\frac{1}{4}$ mile and 20¢ for each additional $\frac{1}{4}$ mile, write a program to find the cost of a taxi ride given the number of miles. **Note:** Assume that the meter clicks on the quarter-mile. See Fig. 3.59.

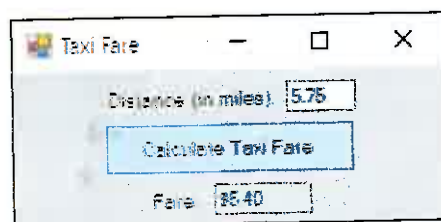


FIGURE 3.59 Possible outcome of Exercise 87.

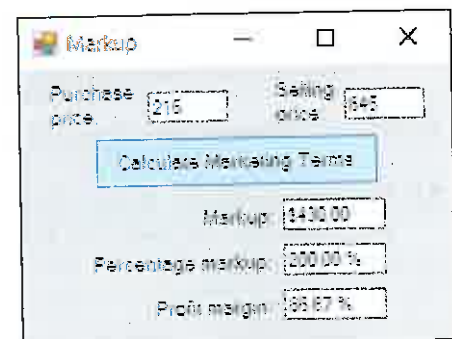


FIGURE 3.60 Possible outcome of Exercise 88.

- 88. Marketing Terms** The *markup* of an item is the difference between its *selling price* and its *purchase price*. Two other marketing terms are

$$\text{percentage markup} = \frac{\text{markup}}{\text{purchase price}} \quad \text{and} \quad \text{profit margin} = \frac{\text{markup}}{\text{selling price}}$$

where the quotients are expressed as percentages. Write a program that computes the markup, percentage markup, and profit margin of an item. See Fig. 3.60. Notice that when the purchase price is tripled, the percentage markup is 200%.

Solutions to Practice Problems 3.3

1. None. **Note:** The Auto Correction feature of IntelliSense seems to prefer the use of CStr. When the smart editor encounters a statement such as `textBox.Text = 12.345`, it signals that a syntax error has occurred and recommends changing 12.345 to CStr (12.345). The convention in this book is to use the ToString method whenever formatting is wanted and to otherwise use CStr.
2. The outcomes are identical. In this text, we primarily use the second style.

CHAPTER 3 SUMMARY

1. Three types of *literals* that can be stored and processed by Visual Basic are numbers, strings, and dates.
2. Many Visual Basic tasks are carried out by methods such as Clear (erases the contents of a text box or list box), Add (places an item into a list box), ToUpper (converts a string to uppercase), ToLower (converts a string to lowercase), Trim (removes leading and trailing spaces from a string), IndexOf (searches for a specified substring in a string and gives its position if found), and Substring (produces a sequence of consecutive characters from a string).
3. The *arithmetic operations* are +, -, *, /, ^, \, and Mod. The only string operation is &, concatenation. An *expression* is a combination of literals, variables, functions, and operations that can be evaluated.
4. A *variable* is a name used to refer to data. Variable names must begin with a letter or an underscore and may contain letters, digits, and underscores. Dim statements declare variables, specify the data types of the variables, and assign initial values to the variables. In this book, most variables have data types Double, Decimal, Integer, String, or Date.
5. Values are assigned to variables by *assignment statements*. The values appearing in assignment statements can be literals, variables, or expressions. String literals used in assignment statements must be surrounded by quotation marks. Date literals used in assignment statements must be surrounded by number signs.
6. *Comment statements* are used to explain formulas, state the purposes of variables, and articulate the purposes of various parts of a program.
7. *Option Explicit* requires that all variables be declared with Dim statements. *Option Strict* requires the use of conversion functions in certain situations.
8. The *Error List window* displays, and helps you find, errors in the code. The *Auto Correction* feature of IntelliSense suggests corrections when errors occur.
9. *Line continuation* is used to extend a Visual Basic statement over two or more lines.
10. The *scope* of a variable is the portion of the program in which the variable is visible and can be used. A variable declared inside an event procedure is said to have *procedure scope* and is visible only inside the procedure. A variable declared in the Declarations section of a class is said to have *class scope* and is visible throughout the entire program.

11. *Masked text boxes* help obtain correct input with a Mask property that limits the kind of data that can be typed into the text box.
12. The *Date* data type facilitates computations involving dates.
13. An *input dialog box* is a window that pops up and displays a message for the user to respond to in a text box. The response is assigned to a variable.
14. A *message dialog box* is a window that pops up to display a message to the user.
15. *Named constants* store values that cannot change during the execution of a program. They are declared with *Const* statements.
16. The following *functions* accept numbers, strings, or dates as input and return numbers or strings as output.

FUNCTION/METHOD	INPUT	OUTPUT
Cdbl	string or number	number
CDec	string or number	number
CInt	string or number	number
CStr	string or number	string
ToString("N")	number	string
ToString("C")	number	string
ToString("P")	number	string
ToString("D")	date	string
ToString("d")	date	string
DateDiff	date, date	number
InputBox	string, string	string
Int	number	number
Math.Round	number, number	number
Math.Sqrt	number	number

CHAPTER 3 PROGRAMMING PROJECTS

1. **Calculator** Write a program that allows the user to specify two numbers and then adds, subtracts, or multiplies them when the user clicks on the appropriate button. The output should give the type of arithmetic performed and the result. See Fig. 3.61. **Note:** When one of the numbers in an input text box is changed, the output text box should be cleared.

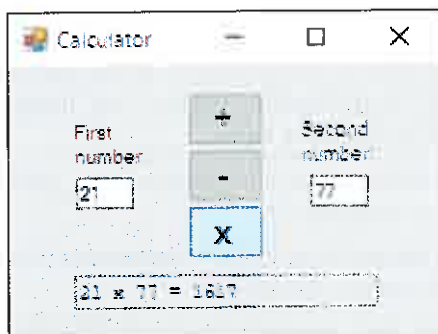


FIGURE 3.61 Possible outcome of Programming Project 1.

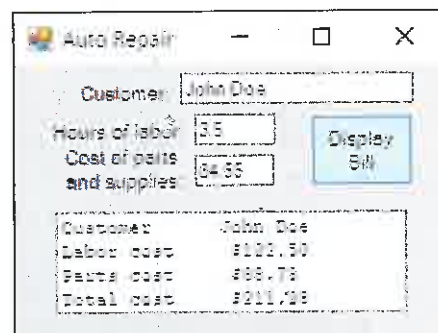


FIGURE 3.62 Possible outcome of Programming Project 2.

2. **Repair Bill** Suppose automobile repair customers are billed at the rate of \$35 per hour for labor. Also, suppose costs for parts and supplies are subject to a 5% sales tax. Write a program to display a simplified bill. The customer's name, the number of hours of labor, and the cost of parts and supplies should be entered into the program via text

boxes. When a button is clicked, the customer's name and the three costs should be displayed in a list box, as shown in Fig. 3.62.

3. **Change** Write a program to make change for an amount of money from 0 through 99 cents input by the user. The output of the program should show the number of coins from each denomination used to make change. See Fig. 3.63.

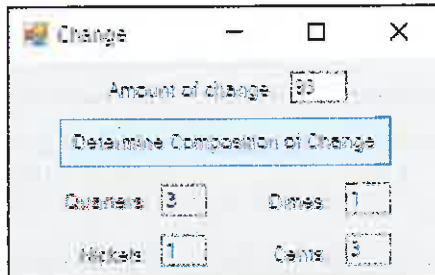


FIGURE 3.63 Possible outcome of Programming Project 3.

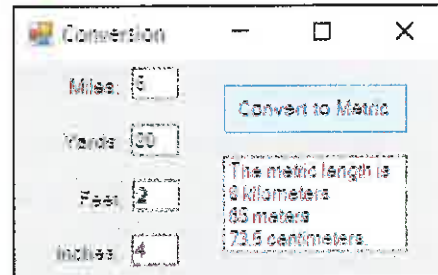


FIGURE 3.64 Possible outcome of Programming Project 4.

4. **Length Conversion** Write a program to convert a U.S. Customary System length in miles, yards, feet, and inches to a Metric System length in kilometers, meters, and centimeters. A sample run is shown in Fig. 3.64. After the numbers of miles, yards, feet, and inches are read from the text boxes, the length should be converted entirely to inches and then divided by 39.37 to obtain the value in meters. The Int function should be used to break the total number of meters into a whole number of kilometers and meters. The number of centimeters should be displayed to one decimal place. The needed formulas are as follows:

$$\text{total inches} = 63360 * \text{miles} + 36 * \text{yards} + 12 * \text{feet} + \text{inches}$$

$$\text{total meters} = \text{total inches} / 39.37$$

$$\text{kilometers} = \text{Int}(\text{meters} / 1000)$$

5. **Car Loan** If A dollars are borrowed at $r\%$ interest compounded monthly to purchase a car with monthly payments for n years, then the monthly payment is given by the formula

$$\text{monthly payment} = \frac{i}{1 - (1 + i)^{-12n}} \cdot A,$$

where $i = \frac{r}{1200}$. Write a program that calculates the monthly payment after the user gives the amount of the loan, interest rate, and number of years. Figure 3.65 shows that monthly payments of \$234.23 are required to pay off a 5-year loan of \$12,000 at 6.4% interest.

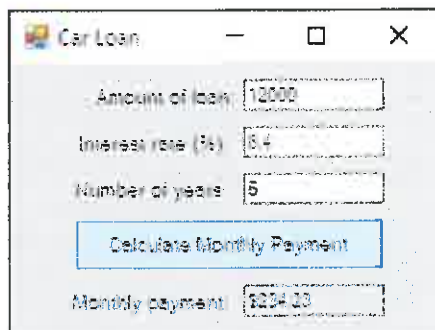


FIGURE 3.65 Possible outcome of Programming Project 5.



FIGURE 3.66 Possible outcome of Programming Project 6.

- 6. Unit Price** Write a program that requests the price and weight of an item in pounds and ounces, and then determines the price per ounce. See Fig. 3.66 on the previous page.
- 7. Bond Yield** One measure of a bond's performance is its *Yield To Maturity* (YTM). YTM values for government bonds are complex to calculate and are published in tables. However, they can be approximated with the simple formula $YTM = \frac{intr + a}{b}$, where $intr$ is the interest earned per year, $a = \frac{\text{face value} - \text{current market price}}{\text{years until maturity}}$, and $b = \frac{\text{face value} + \text{current market price}}{2}$. For instance, suppose a bond has a face value of \$1000, a coupon interest rate of 4%, matures in 15 years, and currently sells for \$1180. Then $intr = .04 \cdot 1000 = 40$, $a = \frac{1000 - 1180}{15} = -12$, $b = \frac{1000 + 1180}{2} = 1090$, and $YTM = \frac{40 - 12}{1090} \approx 2.57\%$. **Note:** The *face value* of the bond is the amount it will be redeemed for when it matures, and the *coupon interest rate* is the interest rate stated on the bond. If a bond is purchased when it is first issued, then the YTM is the same as the coupon interest rate. Write a program that requests the face value, coupon interest rate, current market price, and years until maturity for a bond, and then calculates the bond's YTM. See Fig. 3.67.

FIGURE 3.67 Possible outcome of Programming Project 7.