

3

Variables, Constants, and Calculations

at the completion of this chapter, you will be able to . . .

1. Distinguish between variables, constants, and controls.
2. Differentiate among the various data types.
3. Apply naming conventions incorporating standards and indicating scope and data type.
4. Declare variables using the Dim statement.
5. Select the appropriate scope for a variable.
6. Convert text input to numeric values.
7. Perform calculations using variables and constants.
8. Format values for output using the formatting functions.
9. Use Try/Catch blocks for error handling.
10. Display message boxes with error messages.
11. Accumulate sums and generate counts.

In this chapter you will learn to do calculations in Visual Basic. You will start with text values input by the user, convert them to numeric values, and perform calculations on them. You will also learn to format the results of your calculations and display them for the user.

Although the calculations themselves are quite simple (addition, subtraction, multiplication, and division), there are some important issues to discuss first. You must learn about variables and constants, the various types of data used by Visual Basic, and how and where to declare variables and constants. Variables are declared differently, depending on where you want to use them and how long you need to retain their values.

The code below is a small preview to show the calculation of the product of two text boxes. The first group of statements (the `Dims`) declares the variables and their data types. The second group of statements converts the text box contents to numeric and places the values into the variables. The last line performs the multiplication and places the result into a variable. The following sections of this chapter describe how to set up your code for calculations.

```
'Dimension the variables
Dim intQuantity As Integer
Dim decPrice As Decimal
Dim decExtendedPrice As Decimal

'Convert input text to numeric and assign values to variables
intQuantity = CInt(txtQuantity.Text)
decPrice = CDec(txtPrice.Text)
'Calculate the product
decExtendedPrice = intQuantity * decPrice
```

- This chapter holds many changes. VB is now very strongly typed and some of the data types have changed. Students must pay very close attention to the data type of every variable and constant.

- Replace the `Val` conversion function with conversions to specific data type: `CInt` for Integer and `CDec` for Decimal.

Data: Variables and Constants

So far, all data you have used in your projects have been properties of objects. You have worked with the `Text` property of Text Boxes and Labels. Now you will work with values that are not properties. Basic allows you to set up locations in memory and give each location a name. You can visualize each memory location as a scratch pad; the contents of the scratch pad can change as the need arises. In this example, the memory location is called *intMaximum*.

```
intMaximum = 100
```

intMaximum
100

After executing this statement, the value of `intMaximum` is 100. You can change the value of `intMaximum`, use it in calculations, or display it in a control.

In the preceding example, the memory location called `intMaximum` is a variable. Memory locations that hold data that can be changed during project execution are called **variables**; locations that hold data that cannot change during execution are called **constants**. For example, the customer's name will vary as the information for each individual is processed. However, the name of the company and the sales tax rate will remain the same (at least for that day).

When you declare a variable or a **named constant**, Visual Basic reserves an area of memory and assigns it a name, called an **identifier**. You specify

identifier names according to the rules of Basic as well as some recommended naming conventions.

The **declaration** statements establish your project's variables and constants, give them names, and specify the type of data they will hold. The statements are not considered executable; that is, they are not executed in the flow of instructions during program execution.

Here are some sample declaration statements:

```
Dim strName           As String           'Declare a string variable
Dim intCounter        As Integer          'Declare an integer variable
Const decDISCOUNT_RATE As Decimal =0.15D 'Declare a named constant
```

The next few sections describe the data types, the rules for naming variables and constants, and the format of the declarations.

Data Types

The **data type** of a variable or constant indicates what type of information will be stored in the allocated memory space: perhaps a name, a dollar amount, a date, or a total. Table 3.1 shows the VB data types.

The Visual Basic Data Types, the Kind of Data Each Type Holds, and the Amount of Memory Allocated for Each

Data type	Use for	Storage size in bytes
Boolean	True or False values	2
Byte	0 to 255, binary data	1
Char	Single Unicode character	2
Date	1/1/0001 through 12/31/9999	8
Decimal	Decimal fractions, such as dollars and cents	16
Single	Single-precision floating point numbers with six digits of accuracy	4
Double	Double-precision floating-point numbers with 14 digits of accuracy	8
Short	Small integer in the range -32,768 to 32,767	2
Integer	Whole numbers in the range -2,147,483,648 to +2,147,483,647	4
Long	Larger whole numbers	8
String	Alphanumeric data: letters, digits, and other characters	varies
Object	Any type of data	4

- Data types are significantly changed for interoperability with other .NET languages.
- Integer data type is a larger number; the former Integer type is now known as Short.
- Currency data type is no longer supported. The new Decimal data type replaces Currency for decimal numbers.
- The Variant data type is gone. The default data type now is Object.

Table 3.1

The most common types of variables and constants we will use are String, Integer, and Decimal. When deciding which data type to use, follow this guideline: If

the data will be used in a calculation, then it must be numeric (usually Integer or Decimal); if it is not used in a calculation, it will be String. Use Decimal as the data type for any decimal fractions in business applications; Single and Double data types are generally used in scientific applications.

Consider the following examples:

Contents	Data type	Reason
Social Security number	String	Not used in a calculation.
Pay rate	Decimal	Used in a calculation, contains a decimal point.
Hours worked	Decimal	Used in a calculation, may contain a decimal point. (Decimal can be used for any decimal fraction, not just dollars.)
Phone number	String	Not used in a calculation.
Quantity	Integer	Used in a calculation; contains a whole number.

Naming Rules

A programmer has to name (identify) the variables and named constants that will be used in a project. Basic requires identifiers for variables and named constants to follow these rules: names may consist of letters, digits, and underscores; they must begin with a letter; they cannot contain any spaces or periods; and they may not be reserved words. (Reserved words, also called *keywords*, are words to which Basic has assigned some meaning, such as *print*, *name*, and *value*.)

Identifiers in VB are not case sensitive. Therefore, the names `intSum`, `IntSum`, `intsum`, and `INTSUM` all refer to the same variable.

Note: In earlier versions of VB, the maximum length of an identifier was 255 characters. In VB .NET, you can forget about the length limit, since the maximum is now 16,383.

- The maximum length for variable names has effectively been eliminated, since the limit is now 16,383 characters.

Naming Conventions

When naming variables and constants, you *must* follow the rules of Basic. In addition, you *should* follow some naming conventions. Conventions are the guidelines that separate good names from bad (or not-so-good) names. The meaning and use of all identifiers should always be clear.

Just as we established conventions for naming objects in Chapter 1, in this chapter we adopt conventions for naming variables and constants. The following conventions are widely used in the programming industry:

1. *Identifiers must be meaningful.* Choose a name that clearly indicates its purpose. Do not abbreviate unless the meaning is obvious, and do not use very short identifiers, such as *X* or *Y*.
2. *Precede each identifier with a lowercase prefix that specifies the data type.* This convention is similar to the convention we already adopted for naming objects and is widely used in the programming field.

3. Capitalize each word of the name (following the prefix). Always use mixed case for variables, uppercase for constants.

Here is a list of the most common data types and their prefixes:

Prefix	Data type
bln	Boolean
dat	Date
dec	Decimal
dbl	Double-precision floating point
int	Integer
lng	Long integer
sng	Single-precision floating point
str	String

- The prefix for Date variables has changed from “dtm” to “dat”, since Microsoft changed the name of the data type from DateTime to Date. (Dates still can hold both the date and time.)

The following is a list of *sample identifiers*:

Field of data	Possible identifier
Social Security number	strSocialSecurityNumber
Pay rate	decPayRate
Hours worked	decHoursWorked
Phone number	strPhoneNumber
Quantity	intQuantity
Tax rate (constant)	decTAX_RATE
Quota (constant)	intQUOTA
Population	lngPopulation



Feedback 3.1

Indicate whether each of the following identifiers conforms to the rules of Basic and to the naming conventions. If the identifier is invalid, give the reason. Remember, the answers to Feedback questions are found in Appendix A.

- | | |
|--------------------|---------------------|
| 1. omitted | 7. strSub |
| 2. int#Sold | 8. Text |
| 3. int Number Sold | 9. conMaximum |
| 4. int.Number.Sold | 10. MinimumRate |
| 5. sng\$Amount | 11. decMaximumCheck |
| 6. Sub | 12. strCompanyName |

Constants: Named and Intrinsic

Constants provide a way to use words to describe a value that doesn't change. In Chapter 2 you used the Visual Studio constants `Color.Blue`, `Color.Red`, `Color.Yellow`, and so on. Those constants are built into the environment and called *intrinsic constants*; you don't need to define them anywhere. The constants that you define for yourself are called *named constants*.

Named Constants

You declare named constants using the keyword `Const`. You give the constant a name, a data type, and a value. Once a value is declared as a constant, its value cannot be changed during the execution of the project. The data type that you declare and the data type of the value must match. For example, if you declare an integer constant, you must give it an integer value.

You will find two important advantages to using named constants rather than the actual values in code. The code is easier to read; for example, seeing the identifier `decMAXIMUM_PAY` is more meaningful than seeing a number, such as 1,000. In addition, if you need to change the value at a later time, you need to change the constant declaration only once; you do not have to change every reference to it throughout the code.

Const Statement—General Form

General Form

```
Const Identifier [As Datatype] = Value
```

Naming conventions for constants require a prefix that identifies the data type as well as the “As” clause that actually declares the data type.

This example sets the company name, address, and the sales tax rate as constants:

Const Statement—Examples

Examples

```
Const strCOMPANY_NAME As String = "R 'n R--for Reading 'n Refreshment"  
Const strCOMPANY_ADDRESS As String = "101 S. Main Street"  
Const decSALES_TAX_RATE As Decimal = .08D
```

Assigning Values to Constants

The values you assign to constants must follow certain rules. You have already seen that a text (string) value must be enclosed in quotation marks; numeric values are not enclosed. However, you must be aware of some additional rules.

Numeric constants may contain only the digits (0–9), a decimal point, and a sign (+ or –) at the left side. You cannot include a comma, dollar sign, any other special characters, or a sign at the right side. You can declare the data type of numeric constants by appending a type-declaration character. If you do not append a type-declaration character to a numeric constant, any whole number is assumed to be `Integer` and any fractional value is assumed to be `Double`. The type-declaration characters are

Decimal D

Double R

- Use the type declaration characters for numeric constants and always make the constant the same data type as the variable to which you assign it.

Integer I
 Long L
 Short S
 Single F

String literals (also called *string constants*) may contain letters, digits, and special characters, such as \$#@%&*. You may have a problem when you want to include quotation marks inside a string literal, since quotation marks enclose the literal. The solution is to use two quotation marks together inside the literal. Visual Basic will interpret the pair as one symbol. For example, "He said, ""I like it."" " produces this string: He said, "I like it."

Although you can use numeric digits inside a string literal, remember that these numbers are text and cannot be used for calculations.

The string values are referred to as **string literals** because they contain exactly (literally) whatever is inside the quotation marks.

The following table lists example constants.

Data type	Constant value example
Integer	5 125 2170 2000 -100 12345678I
Single	101.25F -5.0F
Decimal	850.50D -100D
Double	52875.8 52875.8R -52875.8R
Long	134257987L -8250758L
String literals	"Visual Basic" "ABC Incorporated" "1415 J Street" "102" "She said ""Hello."""

Intrinsic Constants

Intrinsic constants are system-defined constants. Many sets of intrinsic constants are declared in system class libraries and are available for use in your VB programs. For example, the color constants that you used in Chapter 2 are intrinsic constants.

You must specify the class name or group name as well as the constant name when you use intrinsic constants. For example, Color.Red is the constant "Red" in the class "Color." Later in this chapter you will learn to use constants from the MessageBox class for displaying message boxes to the user.

Declaring Variables

Although there are several ways to declare a variable, the most commonly used statement is the `Dim` statement.

- You can give a variable an initial value as you declare it:

```
Dim intMax As Integer = 100
```

Dim Statement—General Form

General Form

```
Dim Identifier [As Datatype]
```

If you omit the optional data type, the variable's type defaults to object. It is best to always declare the type, even when you intend to use objects.

Dim Statement—Examples

Examples

```
Dim strCustomerName As String
Dim intTotalSold As Integer
Dim sngTemperature As Single
Dim decPrice As Decimal
```

The reserved word `Dim` is really short for dimension, which means *size*. When you declare a variable, the amount of memory reserved depends on its data type. Refer to Table 3.1 (page 97) for the size of each data type.

Entering Dim Statements

Visual Basic's IntelliSense feature helps you enter `Dim` statements. After you type the space that follows `Dim VariableName As`, a list pops up (Figure 3.1). This list shows the possible entries for data type to complete the statement. The easiest way to complete the statement is to begin typing the correct entry; the list automatically scrolls to the correct section (Figure 3.2). When the correct entry is highlighted, press `Enter`, `Tab`, or the spacebar to select the entry, or double-click if you prefer using the mouse.

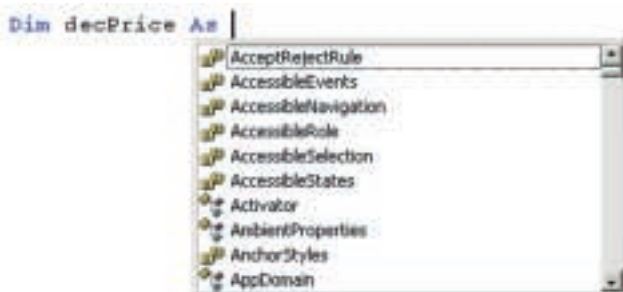
Note: Some people find the IntelliSense feature annoying rather than helpful. You can turn off the feature by selecting *Tools / Options / Text Editor / All Languages* and deselecting *Auto list members*.

- You can declare multiple variables on one statement and name the data type only once. In VB 6, the data type applied only to the one variable on which it was placed; in VB .NET, the data type applies to all variables named on the statement.

```
Dim intCount, intMax, _
    intNumber As Integer _
    'Creates 3 integer _
    variables
```

Figure 3.1

As soon as you type the space after "As," the IntelliSense menu pops up. You can make a selection from the list with your mouse or the keyboard.



**Figure 3.2**

Type the first few characters of the data type and the IntelliSense list quickly scrolls to the correct section. When the correct word is highlighted, press Enter, Tab, or the Spacebar to select the entry.

Feedback 3.2

Write a declaration for the following situations. In your declaration, make up an appropriate variable identifier.

1. You need variables for payroll processing to store the following:
 - (a) Number of hours, which can hold a decimal point.
 - (b) String employee's name.
 - (c) Department number (not used in calculations).
2. You need variables for inventory control to store the following:
 - (a) Integer quantity.
 - (b) Description of the item.
 - (c) Part number.
 - (d) Cost.
 - (e) Selling price.

Scope and Lifetime of Variables

A variable may exist and be visible for an entire project, for only one form, or for only one procedure. The visibility of a variable is referred to as its **scope**. Visibility really means “this variable can be used or ‘seen’ in this location.” The scope is said to be namespace, module level, local, or block. A **namespace variable** may be used in all procedures of the namespace, which is generally the entire project. **Module-level variables** are accessible from all procedures of a form. A **local variable** may be used only within the procedure in which it is declared, and a **block variable** is used only within a block of code inside a procedure.

You declare the scope of a variable by choosing where to place the Dim statement.

Note: Previous versions of VB and some other programming languages refer to namespace variables as a *global variables*.

Variable Lifetime

When you create a variable, you must be aware of its **lifetime**. The lifetime of a variable is the period of time that the variable exists. The lifetime of a local or block variable is normally one execution of a procedure. For example, each time you execute a sub procedure, the local Dim statements are executed. Each variable is created as a “fresh” new one, with an initial value of 0 for numeric variables and an empty string for string variables. When the procedure finishes, its variables disappear; that is, their memory locations are released.

- Microsoft has changed *global* scope to *namespace* scope. Any variable declared at the module level with the **Public** keyword is considered a namespace-level variable.

The lifetime of a module-level variable is the entire time the form is loaded, generally the lifetime of the entire project. If you want to maintain the value of a variable for multiple executions of a procedure, for example, to calculate a running total, you must use a module-level variable (or a variable declared as `Static`, which is discussed in Chapter 7).

Local Declarations

Any variable that you declare inside a procedure is local in scope which means it is known only to that procedure. A `Dim` statement can appear anywhere inside the procedure as long as it appears prior to the first use of the variable in a statement. However, good programming practices dictate that all `Dims` appear at the top of the procedure, prior to all other code statements (after the remarks).

```
'Module-level Declarations
Const mdecDISCOUNT_RATE      As Decimal = 0.15D

Private Sub btnCalculate_Click()
    'Calculate the price and discount

    Dim intQuantity              As Integer
    Dim decPrice                 As Decimal
    Dim decExtendedPrice        As Decimal
    Dim decDiscount             As Decimal
    Dim decDiscountedPrice      As Decimal

    'Convert input values to numeric variables
    intQuantity = CInt(txtQuantity.Text)
    decPrice = CDec(txtPrice.Text)

    'Calculate values
    decExtendedPrice = intQuantity * decPrice
    decDiscount = decExtendedPrice * mdecDISCOUNT_RATE
    decDiscountedPrice = decExtendedPrice - decDiscount
```

Notice the `Const` statement in the preceding example. You can declare named constants to be local, block level, module level, or namespace in scope, just as you can variables. However, good programming practices say that constants should be declared at the module level. This technique places all constant declarations at the top of the code and makes them easy to find in case you need to make changes.

Module-Level Declarations

At times you need to be able to use a variable or constant in more than one procedure of a form. When you declare a variable or constant as module level, you can use it anywhere in that form. Place the declarations (`Dim` or `Const`) for module-level variables and constants in the Declarations section of the form. (Recall that you have been using the Declarations section for remarks since Chapter 1.) If you wish to accumulate a sum or count items for multiple executions of a procedure, you should declare the variable at the module level.

Figure 3.3 illustrates the locations for coding local variables and module-level variables.

Figure 3.3

The variables you dimension inside a procedure are local. Variables that you dimension in the Declarations section are module level.

```
(Declarations section)

Dim ModuleLevelVariables
Const NamedConstants

Private Sub btnCalculate_Click
    Dim LocalVariables

    ...
End Sub

Private Sub btnSummary_Click
    Dim LocalVariables

    ...
End Sub

Private Sub btnClear_Click
    Dim LocalVariables

    ...
End Sub
```

```
'Declarations section of a form

'Dimension module-level variables and constants
Dim mintQuantitySum      As Integer
Dim mdecDiscountSum     As Decimal
Dim mintSaleCount       As Integer
Const decMAXIMUM_DISCOUNT As Decimal = 100.0D
```

Including the Scope in Identifiers

When you use variables, it is important to know their scope. For that reason you should include scope information in your naming conventions. To indicate a module-level variable, place a prefix of `m` before the identifier. Local variables do not have an additional prefix, so any variable without an initial `m` can be assumed to be local. For example,

```
Dim mdecTotalPay As Decimal
```

Note that the `m` stands for module level, and the `dec` stands for Decimal data type.

```
Const mintNUMBER_QUESTIONS As Integer = 50I
```

The `m` stands for module level, and the `int` stands for Integer data type.

Coding Module-Level Declarations

To enter module-level declarations, you must be in the Editor window at the top of your code (Figure 3.4). Place the `Dim` and `Const` statements after the Form Designer Generated Code section but before your first procedure.

Figure 3.4

```

' Uses variables, constants, calculations, error
' handling, and a message box to the user.
'Folder:      C:\0302

Option Strict On

Public Class frmBookSale
    Inherits System.Windows.Forms.Form

    Windows Form Designer generated code

    'Dimension module-level variables and constants
    Dim mintQuantitySum As Integer
    Dim mdecDiscountSum As Decimal
    Dim mdecDiscountedPriceSum As Decimal
    Dim mintSaleCount As Integer
    Const mdecDISCOUNT_RATE As Decimal = 0.15D

    Private Sub btnCalculate_Click(ByVal sender As System.Object,

```

Code module-level declarations in the Declarations section at the top of your code.

Block-Level and Global Declarations

You won't use block-level or namespace-level declarations in this chapter. Block-level variables and constants have a scope of a block of code, such as `If / End If` or `Do / Loop`. These statements are covered later in this text.

Namespace-level variables and constants may be useful when a project has multiple forms and/or modules. Good programming practices exclude the use of namespace-level variables.

Feedback 3.3

Write the declarations (`Dim` or `Const` statements) for each of the following situations and indicate where each statement will appear.

1. The total of the payroll that will be needed in a Calculate event procedure and in a Summary event procedure.
2. The sales tax rate that cannot be changed during execution of the program but will be used by multiple procedures.
3. The number of participants that are being counted in the Calculate event procedure, but not displayed until the Summary event procedure.

Calculations

In programming you can perform calculations with variables, with constants, and with the properties of certain objects. The properties you will use, such as the `Text` property of a text box or a label, are usually strings of text characters. These character strings, such as "Howdy" or "12345", should not be used directly in calculations unless you first convert them to the correct data type.

Converting to the Correct Data Type

You will use functions to convert the property of a control to its numeric form before you use the value in a calculation. The function that you use depends on the data type of the variable to which you are assigning the value. For example, to convert text to an integer, use the `CInt` function; to convert to a decimal value, use `CDec`.

```
'Convert input values to numeric variables
intQuantity = CInt(txtQuantity.Text)
decPrice = CDec(txtPrice.Text)
'Calculate the extended price
decExtendedPrice = intQuantity * decPrice
```

Converting from one data type to another is sometimes called *casting*. In the preceding example, `txtQuantity.Text` is cast into an Integer data type and `txtPrice.Text` is cast into a Decimal data type.

Using Functions

Visual Basic supplies many functions that you can use in your programs. A **function** performs an action and returns a value. The expression to operate upon, called the **argument** (or multiple arguments, in some cases), must be enclosed in parentheses.

The first Basic functions we will use are the **conversion functions**, `CInt` and `CDec`. Both functions convert an argument into a numeric value of the correct data type. You will also use the `CStr` function to convert in the other direction—converting numeric values to strings for display purposes. And later in this chapter you use some formatting functions, which convert numeric values to formatted strings.

- The conversion functions, such as `CInt` and `CDec`, now parse some characters that would have been considered nonnumeric in VB 6. For example, the functions accept dollar signs, commas, parentheses, and signs on either end of a number.
- When `CInt` rounds to an integer, it rounds to the nearest *even number*, not the nearest whole number. Statisticians argue that this is a much better method of rounding, but we're not convinced.

The Conversion Functions—General Forms

General
Form

```
CInt(ExpressionToConvert) 'Convert to Integer
CDec(ExpressionToConvert) 'Convert to Decimal
CStr(ExpressionToConvert) 'Convert to String
```

The expression you wish to convert can be the property of a control, a variable, or a constant.

A function cannot stand by itself. It returns (produces) a value that can be used as a part of a statement, such as the assignment statements in the following examples.

The Conversion Functions—Examples

Examples

```
intQuantity = CInt(txtQuantity.Text)
decPrice = CDec(txtPrice.Text)
intWholeNumber = CInt(decFractionalValue)
decDollars = CDec(intDollars)
strValue = CStr(decValue)
```

The numeric conversion functions examine the value stored in the argument and attempt to convert to a number in a process called *parsing*, which means to pick apart, character by character, and convert to another format. Although you can change the default parsing method based on the symbols you need to recognize, the defaults will probably serve you and your users very well.

The VB .NET conversion functions can recognize and convert many values that previous versions would have rejected as nonnumeric, such as dollar signs, commas, parentheses, and signs on either end of a number.

The `CInt` function first converts the argument to numeric and then rounds, if necessary, to produce an integer. The rounding method is different from what you might expect: It rounds to the nearest *even* number. For example, 1.5 rounds to 2, but 0.5 rounds to zero.

Here is a list of conversion examples:

Contents of string argument	<code>CInt(Argument)</code>	<code>CDec(Argument)</code>
123.45	123	123.45
\$100	100	100.0
1,000.00	1000	1000.0
A123	(error)	(error)
-5	-5	-5.0
5-	-5	-5.0
(5)	-5	-5.0
1(5)	(error)	(error)
0.01	0	0.01
0.5	0	0.5
1.5	2	1.5
(blank)	(error)	(error)

When a conversion function encounters a value that it cannot parse to a number, such as a blank or a nonnumeric character, an error occurs. You will learn how to avoid those errors later in this chapter in the section titled “Handling Exceptions.”

You will use the `CInt` and `CDec` functions most of the time. But in case you need to convert to `Long`, `Single`, or `Double`, VB also has conversion functions for those: `CLng`, `CSng`, and `Cdbl`. These functions parse and convert values in the same manner as that described for the other functions.

- Excellent reference [Help pages](#), perhaps for reading in class, include
 - [Efficient Use of Data Types](#)
 - [+ Operator](#)
 - [Option Strict Statement](#)
 - [Widening and Narrowing Conversions](#)

Arithmetic Operations

The arithmetic operations you can perform in Visual Basic include addition, subtraction, multiplication, division, integer division, modulus, and exponentiation.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Integer division
Mod	Modulus—Remainder of division
^	Exponentiation

The first four operations are self-explanatory, but you may not be familiar with \, Mod, or ^.

Integer Division (\)

Use integer division (\) to divide one integer by another giving an integer result, truncating (dropping) any remainder. For example, if `intTotalMinutes = 150`, then

```
intHours = intTotalMinutes\60
```

returns 2 for `intHours`.

Mod

The Mod operator returns the remainder of a division operation. For example, if `intTotalMinutes = 150`, then

```
intMinutes = intTotalMinutes Mod 60
```

returns 30 for `intMinutes`.

Exponentiation (^)

The exponentiation operator (^) raises a number to the power specified. The following are examples of exponentiation.

```
decSquared = decNumber ^ 2    'Square the number--Raise to the 2nd power  
decCubed = decNumber ^ 3     'Cube the number--Raise to the 3rd power
```

Order of Operations

The order in which operations are performed determines the result. Consider the expression $3 + 4 * 2$. What is the result? If the addition is done first, the result is 14. However, if the multiplication is done first, the result is 11.

The hierarchy of operations, or **order of precedence**, in arithmetic expressions from highest to lowest is

1. Any operation inside parentheses
2. Exponentiation
3. Multiplication and division
4. Integer division
5. Modulus
6. Addition and subtraction

In the previous example, the multiplication is performed before the addition, yielding a result of 11. To change the order of evaluation, use parentheses. The expression

$$(3 + 4) * 2$$

will yield 14 as the result. One set of parentheses may be used inside another set. In that case, the parentheses are said to be *nested*. The following is an example of nested parentheses:

```
((intScore1 + intScore2 + intScore3) / 3) * 1.2
```

Extra parentheses can always be used for clarity. The expressions

```
2 * decCost * decRate and (2 * decCost) * decRate
```

are equivalent, but the second is easier to understand.

Multiple operations at the same level (such as multiplication and division) are performed from left to right. The example $8 / 4 * 2$ yields 4 as its result, not 1. The first operation is $8 / 4$, and $2 * 2$ is the second.

Evaluation of an expression occurs in this order:

1. All operations within parentheses. Multiple operations within the parentheses are performed according to the rules of precedence.
2. All exponentiation. Multiple exponentiation operations are performed from left to right.
3. All multiplication and division. Multiple operations are performed from left to right.
4. All integer division. Multiple operations are performed from left to right.
5. Mod operations. Multiple operations are performed from left to right.
6. All addition and subtraction are performed from left to right.

Although the precedence of operations in Basic is the same as in algebra, take note of one important difference: There are no implied operations in Basic. The following expressions would be valid in mathematics, but they are not valid in Basic:

Mathematical notation	Equivalent Basic function
2A	2 * A
3(X + Y)	3 * (X + Y)
(X + Y)(X - Y)	(X + Y) * (X - Y)



Use extra parentheses to make the precedence clearer. The operation will be easier to understand and the parentheses have no negative effect on execution. ■

Feedback 3.4

What will be the result of the following calculations using the order of precedence?

Assume that `intX = 2`, `intY = 4`, `intZ = 3`

1. `intX + intY ^ 2`
2. `8 / intY / intX`
3. `intX * (intX + 1)`
4. `intX * intX + 1`
5. `intY ^ intX + intZ * 2`
6. `intY ^ (intX + intZ) * 2`
7. `(intY ^ intX) + intZ * 2`
8. `((intY ^ intX) + intZ) * 2`

Using Calculations in Code

You perform calculations in assignment statements. Recall that whatever appears on the right side of an `=` (assignment operator) is assigned to the item on the left. The left side may be the property of a control or a variable. It cannot be a constant.

Examples

```
decAverage = decSum/intCount
lblAmountDue.Text = CStr(decPrice - (decPrice * decDiscountRate))
txtCommission.Text = CStr(decSalesTotal * decCommissionRate)
```

In the preceding examples, the results of the calculations were assigned to a variable, the `Text` property of a label, and the `Text` property of a text box. In most cases you will assign calculation results to variables or to the `Text` properties of labels. Text boxes are usually used for input from the user rather than for program output.

Assignment Operators

In addition to the equal sign (`=`) as an **assignment operator**, VB .NET has several operators that can perform a calculation and assign the result as one operation. The new assignment operators are `+=`, `-=`, `*=`, `/=`, `\=`, and `&=`. Each of these assignment operators is a shortcut for the standard method; you can use the standard (longer) form or the newer shortcut. The shortcuts allow you to type a variable name only once instead of requiring you to type it on both sides of the equal sign.

For example, to add `decSales` to `decTotalSales`, the long version is

```
decTotalSales = decTotalSales + decSales 'Accumulate a total
```

Instead you can use the shortcut assignment operator:

```
decTotalSales += decSales 'Accumulate a total
```

The two statements have the same effect.

To subtract 1 from a variable, the long version is:

```
intCountDown = intCountDown - 1 'Subtract 1 from variable
```

- New operators, similar to C++ and Java, are

```
+=  -=  *=  /=  \=  &=
(No ++ or --)
```


With Option Strict turned off, code such as this is legal:

```
intQuantity = txtQuantity.Text
```

and

```
intAmount = lngAmount
```

and

```
intTotal += decSaleAmount
```

With each of these legal (but dangerous) statements, the VB compiler makes assumptions about your data. And the majority of the time, the assumptions are correct. But bad input data or very large numbers can cause erroneous results or run-time errors.

The best practice is to always turn on Option Strict. This technique will save you from developing poor programming habits and will also likely save you hours of debugging time. For some reason, the developers at Microsoft elected to turn off Option Strict by default. In early beta trials of the software, the option was turned on by default, but in late betas, the option was turned off.

You can turn on Option Strict either in code or in the *Project Properties* dialog box. Place the line

```
Option Strict On
```

before the first line of code, after the general remarks at the top of a file.

Example

```
'Project:      MyProject
'Date:         Today
'Programmer:   Your Name
'Description:  This project calculates correctly.

Option Strict On

Public Class frmMyForm
    Inherits System.Windows.Forms.Form
```

To turn on Option Strict or Option Explicit for the entire project, open the *Project Properties* dialog box and select *Common Properties/Build*. There you will find settings for both Option Explicit and Option Strict. By default, Option Explicit is turned on and Option Strict is turned off. Setting *Option Strict On* in the project properties has one additional effect—any new files that you add to the project will have the option turned on automatically.

Note: Option Strict includes all of the requirements of Option Explicit. If Option Strict is turned on, variables must be declared, regardless of the setting of Option Explicit.

In another change from VB 6, you should always include the keyword `On` when setting an option. In VB 6, the statement

```
Option Explicit
```

turned *on* the option. In VB .NET, that same statement turns *off* the option, since the default is off.

Formatting Data

When you want to format data for display, use the **formatting functions**. To **format** means to control the way the output looks. For example, 12 is just a number, but \$12.00 conveys more meaning for dollar amounts. When you use the formatting functions, you can choose to display a dollar sign, a percent sign, and commas. You can also specify the number of digits to appear to the right of the decimal point. VB rounds the value to return the requested number of decimal positions.

- Notice that you use the `FormatCurrency` function to format data of the `Decimal` data type.

The `FormatCurrency` Function—Simple Form

Simple Form

```
FormatCurrency(NumericExpressionToFormat)
```

The `FormatCurrency` function returns a string of characters formatted as dollars and cents. By default, the return value displays a dollar sign, commas, and two positions to the right of the decimal point. (*Note:* You can change the default format by changing your computer's regional settings.)

Usually, you will assign the formatted value to the `Text` property of a control for display.

The `FormatCurrency` Function—Simple Example

Example

```
lblBalance.Text = FormatCurrency(decBalance)
```

Examples

Variable	Value	Function	Returns
<code>decBalance</code>	1275.675	<code>FormatCurrency(decBalance)</code>	\$1,275.68
<code>sngAmount</code>	.9	<code>FormatCurrency(sngAmount)</code>	\$0.90

Note that the formatted value returned by the `FormatCurrency` function is no longer purely numeric and cannot be used in further calculations. For example, consider the following lines of code:

```
decAmount += decCharges
lblAmount.Text = FormatCurrency(decAmount)
```

Assume that `decAmount` holds 1050 after the calculation and `lblAmount.Text` displays \$1,050.00. If you want to do any further calculations with this amount, such as adding it to a total, you must use `decAmount` rather than `lblAmount.Text`. The variable `decAmount` holds a numeric value; `lblAmount.Text` holds a string of (nonnumeric) characters.

You can further customize the formatted value returned by the `FormatCurrency` function. You can specify the number of decimal positions

to display, whether or not to display a leading zero for fractional values, whether to display negative numbers in parentheses, and whether to use the commas for grouping digits.

The FormatCurrency Function—General Form

General
Form

```
FormatCurrency(ExpressionToFormat [, NumberOfDecimalPositions [, LeadingDigit _
[, UseParenthesesForNegative [, GroupingForDigits]]]])
```

As you can see, the only required argument is the expression you want to format.

Examples			
Variable	Value	Function	Returns
mdecTotal	1125.67	FormatCurrency(mdecTotal, 0)	\$1,126
mdecTotal	1125.67	FormatCurrency(mdecTotal)	\$1,126.67
mdecTotal	1125.67	FormatCurrency(mdecTotal, 2)	\$1,126.67
decBalance	1234.567	FormatCurrency(decBalance, 0)	\$1,235
decBalance	1234.567	FormatCurrency(decBalance)	\$1,234.57
decBalance	1234.567	FormatCurrency(decBalance, 2)	\$1,234.57

For an explanation of the other options of the FormatCurrency function, see Help.

The FormatNumber Function—Simple Form

Simple
Form

```
FormatNumber(ExpressionToFormat)
```

The FormatNumber function is similar to the FormatCurrency function. The default format is determined by your computer's regional setting; it will generally display commas and two digits to the right of the decimal point.

The FormatNumber Function—Simple Examples

Examples

```
lblSum.Text = FormatNumber(decSum)
lblCount.Text = FormatNumber(intCount)
```

Both of these examples will display with commas and two digits to the right of the decimal point. You can specify the exact number of decimal digits, just as you can with the FormatCurrency function. The following example formats the number with commas and no digits to the right of the decimal point.

```
lblWholeNumber.Text = FormatNumber(intCount, 0)
```

The FormatNumber Function—General Form

General
Form

```
FormatNumber(ExpressionToFormat [, NumberOfDecimalPositions [, LeadingDigit _
[, UseParenthesesForNegative [, GroupingForDigits]]]])
```

See Help for an explanation of the optional arguments of the FormatNumber function.

Examples			
Variable	Value	Function	Returns
mdecTotal	1125.67	FormatNumber(mdecTotal, 0)	1,126
decBalance	1234.567	FormatNumber(decBalance)	1,234.57
decBalance	1234.567	FormatNumber(decBalance, 2)	1,234.57

The FormatPercent Function—Simple Form

Simple
Form

```
FormatPercent(ExpressionToFormat)
```

To display numeric values as a percent, use the FormatPercent function. This function multiplies the argument by 100, adds a percent sign, and rounds to two decimal places. (As with the FormatCurrency and FormatNumber functions, the default number of decimal positions is determined by the computer's regional settings and can be changed.)

The FormatPercent Function—Simple Examples

Examples

```
lblPercentComplete.Text = FormatPercent(sngComplete)
lblInterestRate.Text = FormatPercent(decRate)
```

In the complete form of the FormatPercent function, you can select the number of digits to the right of the decimal point as well as customize other options, similar to the other formatting functions.

The FormatPercent Function—General Form

General
Form

```
FormatPercent(ExpressionToFormat [, NumberOfDecimalPositions [, LeadingDigit _
[, UseParenthesesForNegative [, GroupingForDigits]]]])
```

Variable	Value	Function	Returns
decCorrect	.75	FormatPercent(decCorrect)	75.00%
decCorrect	.75	FormatPercent(decCorrect, 1)	75.0%
decCorrect	.75	FormatPercent(decCorrect, 0)	75%
decRate	.734	FormatPercent(decRate)	73.40%
decRate	.734	FormatPercent(decRate, 0)	73%
decRate	.734	FormatPercent(decRate, 1)	73.4%
decRate	.734	FormatPercent(decRate, 2)	73.40%

The FormatDateTime Function—General Form

General Form

```
FormatDateTime(ExpressionToFormat [, NamedFormat])
```

You can format an expression as a date and/or time. The expression may be a string that holds a date or time value, a date type variable, or a function that returns a date. The named formats use your computer’s regional settings. If you omit the optional named format, the function returns the date using the GeneralDate format.

The FormatDateTime Function—Examples

Examples

```
lblStartDate.Text = FormatDateTime(datStartDate, DateFormat.ShortDate)
lblStartTime.Text = FormatDateTime("1/1/00", DateFormat.LongDate)
lblDateAndTime.Text = FormatDateTime(datSomeDate)
```

The actual values returned by the FormatDateTime function depend on the regional settings on your computer. These are the return formats based on U.S. defaults.

Named format	Returns	Example
DateFormat.GeneralDate	A date and/or time	2/28/02 6:01:24 PM If the expression holds a date, returns a short date. If it holds a time, returns a long time. If it holds both, returns both a short date and long time.
DateFormat.LongDate	Day of week, Month Day, Year	Thursday, February 28, 2002
DateFormat.ShortDate	MM/DD/YY	2/28/02
DateFormat.LongTime	HH:MM:SS AM/PM	6:01:24 PM
DateFormat.ShortTime	HH:MM (24 hour clock)	18:01

Feedback 3.6

Give the line of code that assigns the formatted output and tell how the output will display for the specified value.

1. A calculated variable called *mdecAveragePay* has a value of 123.456 and should display in a label called *lblAveragePay*.
2. The variable *sngCorrect*, which contains 0.76, must be displayed as a percentage in the label called *lblPercentCorrect*.
3. The total amount collected in a fund drive is being accumulated in a variable called *mdecTotalCollected*. What statement will display the variable in a label called *lblTotal* with commas and two decimal positions but no dollar sign?

A Calculation Programming Example

R 'n R—for Reading 'n Refreshment needs to calculate prices and discounts for books sold. The company is currently having a big sale, offering a 15 percent discount on all books. In this project you will calculate the amount due for a quantity of books, determine the 15 percent discount, and deduct the discount, giving the new amount due—the discounted amount.

Planning the Project

Sketch a form (Figure 3.5) that meets the needs of your users.

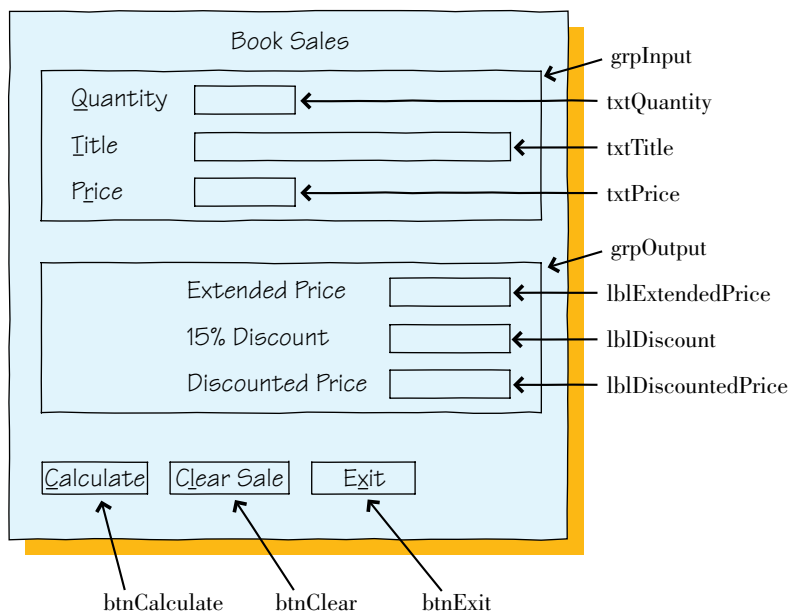


Figure 3.5

A planning sketch of the form for the calculation programming example.

Plan the Objects and Properties

Plan the property settings for the form and each of the controls.

Object	Property	Setting
Form	Name Text AcceptButton CancelButton	frmBooksale R 'n R for Reading 'n Refreshment btnCalculate btnClear
Label1	Text Font	Book Sales Bold, 12 point
grpInput	Name Text	grpInput (blank)
Label2	Text	&Quantity
txtQuantity	Name Text	txtQuantity (blank)
Label3	Text	&Title
txtTitle	Name Text	txtTitle (blank)
Label4	Text	&Price
txtPrice	Name Text	txtPrice (blank)
grpOutput	Name Text	grpOutput (blank)
Label5	Text	Extended Price
lblExtendedPrice	Name Text TextAlign BorderStyle	lblExtendedPrice (blank) TopRight Fixed3D
Label6	Text	15% Discount
lblDiscount	Name Text TextAlign BorderStyle	lblDiscount (blank) TopRight Fixed3D
Label7	Text	Discounted Price
lblDiscountedPrice	Name Text TextAlign BorderStyle	lblDiscountedPrice (blank) TopRight Fixed 3D

Continued on page 120

Object	Property	Setting
btnCalculate	Name Text	btnCalculate &Calculate
btnClear	Name Text	btnClear C&lear Sale
btnExit	Name Text	btnExit E&xit

Plan the Event Procedures

Since you have three buttons, you need to plan the actions for three event procedures.

Event procedure	Actions—pseudocode
btnCalculate_Click	Dimension the variables and constants. Convert the input Quantity and Price to numeric. Calculate Extended Price = Quantity * Price. Calculate Discount = Extended Price * Discount Rate. Calculate Discounted Price = Extended Price – Discount. Format and display output in labels.
btnClear_Click	Set each text box and output label to blanks. Set the focus in the first text box.
btnExit_Click	Exit the project.

Write the Project

Follow the sketch in Figure 3.5 to create the form. Figure 3.6 shows the completed form.

The screenshot shows a Windows application window titled "R 'n R for Reading 'n Refreshment" with a subtitle "Book Sales". The window contains a form with the following elements:

- Input fields for "Quantity", "Title", and "Price".
- Output fields for "Extended Price", "15% Discount", and "Discounted Price".
- Buttons labeled "Calculate", "Clear Sale", and "Exit".

Figure 3.6

The form for the calculation programming example.

- Set the properties of each object, as you have planned.
- Write the code. Working from the pseudocode, write each event procedure.
- When you complete the code, use a variety of test data to thoroughly test the project.

Note: If the user enters nonnumeric data or leaves a numeric field blank, the program will cancel with a run-time error. In the “Handling Exceptions” section that follows this program, you will learn to handle the errors.

The Project Coding Solution

```
'Project:      Chapter Example 3.1
'Date:         January 2002
'Programmer:   Bradley/Millspaugh
'Description:  This project inputs sales information for books.
'              It calculates the extended price and discount for
'              a sale.
'              uses variables, constants, and calculations.
'Folder:       Ch0301

Option Strict On

Public Class frmBooksale
    Inherits System.Windows.Forms.Form

    [Windows Form Designer generated code]

    Const mdecDISCOUNT_RATE As Decimal = 0.15D

    Private Sub btnCalculate_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles btnCalculate.Click
        'Calculate the price and discount

        Dim intQuantity As Integer
        Dim decPrice As Decimal
        Dim decExtendedPrice As Decimal
        Dim decDiscount As Decimal
        Dim decDiscountedPrice As Decimal

        'Convert input values to numeric variables
        intQuantity = CInt(txtQuantity.Text)
        decPrice = CDec(txtPrice.Text)

        'Calculate values
        decExtendedPrice = intQuantity * decPrice
        decDiscount = decExtendedPrice * mdecDISCOUNT_RATE
        decDiscountedPrice = decExtendedPrice - decDiscount

        'Format and display answers
        lblExtendedPrice.Text = FormatCurrency(decExtendedPrice)
        lblDiscount.Text = FormatNumber(decDiscount, 2)
        lblDiscountedPrice.Text = FormatCurrency(decDiscountedPrice)
    End Sub
```

```
Private Sub btnClear_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnClear.Click
    'Clear previous amounts from the form

    txtTitle.Clear()
    txtPrice.Clear()
    lblExtendedPrice.Text = ""
    lblDiscount.Text = ""
    lblDiscountedPrice.Text = ""
    With txtQuantity
        .Clear()
        .Focus()
    End With
End Sub

Private Sub btnExit_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click
    'Exit the project

    Me.Close()
End Sub
End Class
```

Handling Exceptions

When you allow users to input numbers and use those numbers in calculations, lots of things can go wrong. The conversion functions, `CInt` and `CDec`, fail if the user enters nonnumeric data or leaves the text box blank. Or your user may enter a number that results in an attempt to divide by zero. Each of those situations causes an **exception** to occur, or as programmers like to say, *throws an exception*.

You can easily “catch” program exceptions by using VB .NET’s new structured exception handling. You catch the exceptions before they can cause a run-time error and handle the situation, if possible, within the program. Catching exceptions as they happen is generally referred to as *error trapping*, and coding to take care of the problems is called *error handling*. The error handling in Visual Studio .NET is standardized for all of the languages using the Common Language Runtime, which greatly improves on the old error trapping in previous versions of VB.

Try/Catch Blocks

To trap or catch exceptions, enclose any statement(s) that might cause an error in a **Try/Catch block**. If an exception occurs while the statements in the Try block are executing, then program control transfers to the Catch block; if a `Finally` statement is included, the code in that section executes last, whether or not an exception occurred.

- The all new structured exception handling uses `Try/Catch` blocks. We have elected to add the topic to this chapter and begin data validation here.

The Try Block—General Form

General Form

```
Try
    statements that may cause error
Catch [VariableName As ExceptionType]
    statements for action when exception occurs
[Finally
    statements that always execute before exit of Try block]
End Try
```

The Try Block—Example

Example

```
Try
    intQuantity = CInt(txtQuantity.Text)
    lblQuantity.Text = CStr(intQuantity)
Catch
    lblMessage.Text = "Error in input data."
End Try
```

The `Catch` as it appears in the preceding example will catch any exception. You can also specify the type of exception that you want to catch, and even write several `Catch` statements, each to catch a different type of exception. For example, you might want to display one message for bad input data and a different message for a calculation problem.

To specify a particular type of exception to catch, you use one of the pre-defined exception classes, which are all based on, or derived from, the `System.Exception` class. Table 3.2 shows some of the common exception classes.

To catch bad input data that cannot be converted to numeric, write this `Catch` statement:

```
Catch MyErr As InvalidCastException
    lblMessage.Text = "Error in input data."
```

Common Exception Classes

Table 3.2

Exception	Caused by
<code>InvalidCastException</code>	Failure of a conversion function, such as <code>CInt</code> or <code>CDec</code> . Usually blank or nonnumeric data.
<code>ArithmeticException</code>	A calculation error, such as division by zero or overflow of a variable.
<code>System.IO.EndOfStreamException</code>	Failure of an input or output operation such as reading from a file.
<code>OutOfMemoryException</code>	Not enough memory to create an object.
<code>Exception</code>	Generic.

The Exception Class

Each exception is an instance of the `Exception` class. The properties of this class allow you to determine the code location of the error, the type of error, and the cause. The `Message` property contains a text message about the error, and the `Source` property contains the name of the object causing the error. The `StackTrace` property can identify the location in the code where the error occurred.

You can include the text message associated with the type of exception by specifying the `Message` property of the `Exception` object, as declared by the variable you named on the `Catch` statement. Be aware that the messages for exceptions are usually somewhat terse and not oriented to users, but they can sometimes be helpful.

```
Catch MyErr As InvalidCastException
    lblMessage.Text = "Error in input data: " & MyErr.Message
```

Handling Multiple Exceptions

If you want to trap for more than one type of exception, you can include multiple `Catch` blocks (handlers). When an exception occurs, the `Catch` statements are checked in sequence. The first one with a matching exception type is used.

```
Catch MyErr As InvalidCastException
    'statements for non numeric data
Catch MyErr As ArithmeticException
    'statements for calculation problem
Catch MyErr As Exception
    'statements for any other exception
```

The last `Catch` will handle any exceptions that do not match either of the first two exception types. Note that it is acceptable to use the same variable name for multiple `Catch` statements.



Display a list of all system exceptions by selecting *Debug/Exceptions* ■

Displaying Messages in Message Boxes

You may want to display a message when the user has entered invalid data or neglected to enter a required data value. You can display a message to the user in a message box, which is a special type of window. You can specify the message, an optional icon, the title bar text, and button(s) for the message box (Figure 3.7).

- The `MessageBox` class replaces the `MsgBox` function (and moved to this chapter).



Figure 3.7

Two sample message boxes created with the `MessageBox` class.

You use the **Show method** of the **MessageBox** object to display a message box. The **MessageBox** object is a predefined instance of the **MessageBox** class that you can use any time you need to display a message.

The MessageBox Object—General Form

There is more than one way to call the **Show** method. Each of the following statements is a valid call; you can choose the format you want to use. It's very important that the arguments you supply exactly match one of the formats. For example, you cannot reverse, transpose, or leave out any of the arguments. When there are multiple ways to call a method, the method is said to be *overloaded*. See the section “Using Overloaded Methods” later in this chapter.

- The **MessageBox.Show** method is overloaded, which gives the opportunity to cover that OOP concept.

General
Form

```
MessageBox.Show(TextMessage)
MessageBox.Show(TextMessage, TitlebarText)
MessageBox.Show(TextMessage, TitlebarText, MessageBoxButtons)
MessageBox.Show(TextMessage, TitlebarText, MessageBoxButtons, MessageBoxIcon)
```

The **TextMessage** is the message you want to appear in the message box. The **TitleBarText** appears on the title bar of the **MessageBox** window. The **MessageBoxButtons** argument specifies the buttons to display. And the **MessageBoxIcon** determines the icon to display.

The MessageBox Statement—Examples

Examples

```
MessageBox.Show("Enter numeric data.")
MessageBox.Show("Try again.", "Data Entry Error")

MessageBox.Show("This is a message.", "This is a title bar", MessageBoxButtons.OK)

Try
    intQuantity = CInt(txtQuantity.Text)
    lblQuantity.Text = intQuantity
Catch err As InvalidCastException
    MessageBox.Show("Nonnumeric Data.", "Error", MessageBoxButtons.OK, _
        MessageBoxIcon.Exclamation)
End Try
```

The TextMessage String

The message string you display may be a string literal enclosed in quotes or it may be a string variable. You may also want to concatenate several items, for example, combining a literal with a value from a variable. If the message you specify is too long for one line, Visual Basic will wrap it to the next line.

The Title Bar Text

The string that you specify for **TitlebarText** will appear in the title bar of the message box. If you choose the first form of the **Show** method, without the **TitlebarText**, the title bar will appear empty.

MessageBox Buttons

When you show a message box, you can specify the button(s) to display. In Chapter 4, after you learn to make selections using the `If` statement, you will display more than one button and take alternate actions based on which button the user clicks. You specify the buttons using the `MessageBoxButtons` constants from the `MessageBox` class. The choices are `OK`, `OKCancel`, `RetryCancel`, `YesNo`, `YesNoCancel`, and `AbortRetryIgnore`. The default for the `Show` method is `OK`, so unless you specify otherwise, you will get only the `OK` button in your message box.

- New constants for buttons and icons are included in the `MessageBox` class.

MessageBox Icons

The easy way to select the icon to display is to type “`MessageBoxIcon`” and a period into the editor; the IntelliSense list pops up with the complete list. The actual appearance of the icons varies from one operating system to another. You can see a description of the icons in Help under “`MessageBoxIcon` Enumeration.”

Constants for MessageBoxIcon

Asterisk

Error

Exclamation

Hand

Information

None

Question

Stop

Warning

Using Overloaded Methods

As you saw earlier, you can call the `Show` method with several different argument lists. This OOP feature, called **overloading**, allows the `Show` method to act different for different arguments. Each argument list is called a **signature**, so you can say that the `Show` method has several signatures.

When you call the `Show` method, the arguments that you supply must exactly match one of the signatures provided by the method. You must supply the correct number of arguments of the correct data type and in the correct sequence.

Fortunately, the smart Visual Studio editor helps you enter the arguments; you don’t have to memorize or look up the argument lists. Type “`MessageBox.Show(`” and IntelliSense pops up with the first of the signatures for the `Show` method (Figure 3.8). Notice in the figure that there are 12 possible forms



Use the keyboard Up and Down arrow keys rather than the mouse to view and select the signature. The on-screen arrows jump around from one signature to the next, making mouse selection difficult. ■

of the argument list, or 12 signatures for the Show method. (We only showed 4 of the 12 signatures in the previous example, to simplify the concept.)

To select the signature that you want to use, use the up or down arrows at the left end of the IntelliSense popup. For example, to select the signature that needs only the text of the message and the title bar caption, select the fifth format (Figure 3.9). The argument that you are expected to enter is shown in bold and a description of that argument appears in the last line of the pop-up. After you type the text of the message and a comma, the second argument appears in bold, and the description changes to tell you about that argument (Figure 3.10).

Figure 3.8

IntelliSense pops up the first of 12 signatures for the Show method. Use the up and down arrows to see the other possible argument lists.

```
Catch MyErr As InvalidCastException
    MessageBox.Show(|
#1 of 12# Show (text As String, caption As String, buttons As System.Windows.Forms.MessageBoxButtons, icon As System.Windows.Forms.MessageBoxIcon,
defaultButton As System.Windows.Forms.MessageBoxDefaultButton, options As System.Windows.Forms.MessageBoxOptions) As System.Wind
text: The text to display in the message box.
```

Figure 3.9

```
Catch MyErr As InvalidCastException
    MessageBox.Show(
#5 of 12# Show (text As String, caption As String) As System.Windows.Forms.DialogResult
text: The text to display in the message box.
```

Select the fifth signature to see the argument list. The currently selected argument is shown in bold and the description of the argument appears in the last line of the pop-up.

Figure 3.10

```
Catch MyErr As InvalidCastException
    MessageBox.Show("Error in Input Data",
#5 of 12# Show (text As String, caption As String) As System.Windows.Forms.DialogResult
caption: The text to display in the title bar of the message box.
```

Type the first argument and a comma, and IntelliSense bolds the second argument and displays a description of the needed data.

Counting and Accumulating Sums

Programs often need to sum numbers. For example, in the previous programming exercise, each sale is displayed individually. If you want to accumulate totals of the sales amounts, of the discounts, or of the number of books sold, you need some new variables and new techniques.

As you know, the variables you declare inside a procedure are local to that procedure. They are re-created each time the procedure is called; that is, their lifetime is one time through the procedure. Each time the procedure is entered, you have a new fresh variable with an initial value of 0. If you want a variable

to retain its value for multiple calls, in order to accumulate totals, you must declare the variable as module level. (Another approach, using Static variables, is discussed in Chapter 7.)

Summing Numbers

The technique for summing the sales amounts for multiple sales is to dimension a module-level variable for the total. Then in the `btnCalculate_Click` event for each sale, add the current amount to the total:

```
mdecDiscountedPriceSum += decDiscountedPrice
```

This assignment statement adds the current value for `decDiscountedPrice` into the sum held in `mdecDiscountedPriceSum`.

Counting

If you want to count something, such as the number of sales in the previous example, you need another module-level variable. Dimension a counter variable as integer:

```
Dim mintSaleCount as Integer
```

Then in the `btnCalculate_Click` event procedure, add one to the counter variable:

```
mintSaleCount += 1
```

This statement adds 1 to the current contents of `mintSaleCount`. The statement will execute once for each time the `btnCalculate_Click` event procedure executes. Therefore, `mintSaleCount` will always hold a running count of the number of sales.

Calculating an Average

To calculate an average, divide the sum of the items by the count of the items. In the R 'n R book example, we can calculate the average sale by dividing the sum of the discounted prices by the count of the sales.

```
mdecAverageDiscountedSale = mdecDiscountedPriceSum / mintSaleCount
```

Your Hands-On Programming Example

In this project, R 'n R—for Reading 'n Refreshment needs to expand the book sale project done previously in this chapter. In addition to calculating individual sales and discounts, management wants to know the total number of books sold, the total number of discounts given, the total discounted amount, and the average discount per sale. Help the user by adding ToolTips wherever you think they will be useful.

Add error handling to the program, so that missing or nonnumeric data will not cause a run-time error.

Planning the Project

Sketch a form (Figure 3.11) that your users sign off as meeting their needs.

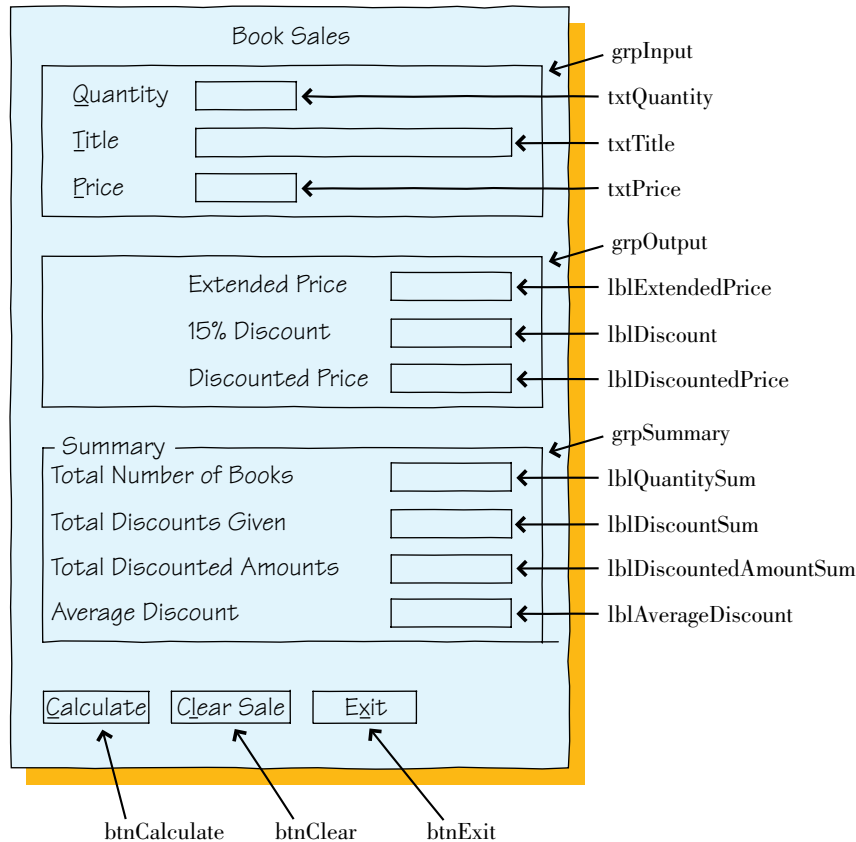


Figure 3.11

A planning sketch of the form for the hands-on programming example.

Plan the Objects and Properties

Plan the property settings for the form and each control. These objects and properties are the same as the previous example, with the addition of the summary information beginning with grpSummary.

Note: The ToolTips have not been added to the planning forms: Make up and add your own.

Object	Property	Setting
Form	Name	frmBooksale
	Text	R 'n R for Reading 'n Refreshment
	AcceptButton	btnCalculate
	CancelButton	btnClear
Label1	Text	Book Sales
	Font	Bold, 12 point
grpInput	Name	grpInput
	Text	(blank)

Continued on page 130

Object	Property	Setting
Label2	Text	&Quantity
txtQuantity	Name Text	txtQuantity (blank)
Label3	Text	&Title
txtTitle	Name Text	txtTitle (blank)
Label4	Text	&Price
txtPrice	Name Text	txtPrice (blank)
grpOutput	Name Text	grpOutput (blank)
Label5	Text	Extended Price
lblExtendedPrice	Name Text BorderStyle TextAlign	lblExtendedPrice (blank) Fixed3D TopRight
Label6	Text	15% Discount
lblDiscount	Name Text BorderStyle TextAlign	lblDiscount (blank) Fixed3D TopRight
Label7	Text	Discounted Price
lblDiscountedPrice	Name BorderStyle TextAlign BorderStyle	lblDiscountedPrice Fixed3D TopRight Fixed3D
btnCalculate	Name Text	btnCalculate &Calculate
btnClear	Name Text	btnClear C&lear Sale
btnExit	Name Text	btnExit E&xit
grpSummary	Name Text	grpSummary Summary
Label8	Text	Total Number of Books
lblQuantitySum	Name Text BorderStyle TextAlign	lblQuantitySum (blank) Fixed3D TopRight
Label9	Text	Total Discounts Given

Object	Property	Setting
lblDiscountSum	Name Text BorderStyle TextAlign	lblDiscountSum (blank) Fixed3D TopRight
Label10	Text	Total Discounted Amounts
lblDiscountedAmountSum	Name Text BorderStyle TextAlign	lblDiscountedAmountSum (blank) Fixed3D TopRight
Label11	Text	Average Discount
lblAverageDiscount	Name Text BorderStyle TextAlign	lblAverageDiscount (blank) Fixed3D TopRight

Plan the Event Procedures

The planning that you did for the previous example will save you time now. The only procedure that requires more steps is the btnCalculate_Click event.

Event procedure	Actions—pseudocode
btnCalculate_Click	Dimension the variables. Try Convert the inputs Quantity and Price to numeric. Calculate Extended Price = Quantity * Price. Calculate Discount = Extended Price * Discount Rate. Calculate Discounted Price = Extended Price - Discount. Calculate the summary values: Add Quantity to Quantity Sum. Add Discount to Discount Sum. Add Discounted Price to Discounted Price Sum. Add 1 to Sale Count. Calculate Average Discount = Discount Sum / Sale Count. Format and display sale output in labels. Format and display summary values in labels. Catch Display error message.
btnClear_Click	Set each text box and label to blanks. Set the focus in the first text box.
btnExit_Click	Exit the project.

Write the Project

Following the sketch in Figure 3.11, create the form. Figure 3.12 shows the completed form.

- Set the properties of each of the objects, as you have planned.
- Write the code. Working from the pseudocode, write each event procedure.
- When you complete the code, use a variety of test data to thoroughly test the project.

Figure 3.12

The form for the hands-on programming example.

The screenshot shows a Windows application window titled "R 'n R for Reading 'n Refreshment" with a subtitle "Book Sales". The window is divided into three main sections. The top section contains three input fields labeled "Quantity", "Title", and "Price". The middle section contains three input fields labeled "Extended Price", "15% Discount", and "Discounted Price". The bottom section is titled "Summary" and contains four input fields labeled "Total Number of Books", "Total Discounts Given", "Total Discounted Amounts", and "Average Discount". At the bottom of the window are three buttons: "Calculate", "Clear Sale", and "Egt".

The Project Coding Solution

```
'Project:      Chapter Example 3.2
'Date:        January 2002
'Programmer:  Bradley/Millspaugh
'Description: This project inputs sales information for books.
'             It calculates the extended price and discount for
'             a sale and maintains summary information for all
'             sales.
'             Uses variables, constants, calculations, error
'             handling, and a message box to the user.
'Folder:     Ch0302
```

```
Option Strict On
```

```
Public Class frmBooksale
    Inherits System.Windows.Forms.Form
```

```
[Windows Form Designer generated code ]
```

```
'Dimension module-level variables and constants
Dim mintQuantitySum As Integer
Dim mdecDiscountSum As Decimal
Dim mdecDiscountedPriceSum As Decimal
Dim mintSaleCount As Integer
Const mdecDISCOUNT_RATE As Decimal = 0.15D

Private Sub btnCalculate_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnCalculate.Click
    'Calculate the price and discount

    Dim intQuantity As Integer
    Dim decPrice As Decimal
    Dim decExtendedPrice As Decimal
    Dim decDiscount As Decimal
    Dim decDiscountedPrice As Decimal
    Dim decAverageDiscount As Decimal

    Try
        'Convert input values to numeric variables
        intQuantity = CInt(txtQuantity.Text)
        decPrice = CDec(txtPrice.Text)

        'Calculate values for sale
        decExtendedPrice = intQuantity * decPrice
        decDiscount = decExtendedPrice * mdecDISCOUNT_RATE
        decDiscountedPrice = decExtendedPrice - decDiscount

        'Calculate summary values
        mintQuantitySum += intQuantity
        mdecDiscountSum += decDiscount
        mdecDiscountedPriceSum += decDiscountedPrice
        mintSaleCount += 1
        decAverageDiscount = mdecDiscountSum / mintSaleCount

        'Format and display answers for sale
        lblExtendedPrice.Text = FormatCurrency(decExtendedPrice)
        lblDiscount.Text = FormatNumber(decDiscount)
        lblDiscountedPrice.Text = FormatCurrency(decDiscountedPrice)

        'Format and display summary values
        lblQuantitySum.Text = CStr(mintQuantitySum)
        lblDiscountSum.Text = FormatCurrency(mdecDiscountSum)
        lblDiscountedAmountSum.Text = FormatCurrency(mdecDiscountedPriceSum)
        lblAverageDiscount.Text = FormatCurrency(decAverageDiscount)

        'Handle exceptions
        Catch MyErr As InvalidCastException
            MessageBox.Show("Enter numeric data.", "Data Entry Error", _
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
        Catch MyErr As Exception
            MessageBox.Show("Error: " & MyErr.Message)
        End Try

    End Sub
```

```
Private Sub btnClear_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnClear.Click
    'Clear previous amounts from the form

    txtTitle.Clear()
    txtPrice.Clear()
    lblExtendedPrice.Text = ""
    lblDiscount.Text = ""
    lblDiscountedPrice.Text = ""
    With txtQuantity
        .Clear()
        .Focus()
    End With
End Sub

Private Sub btnExit_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click
    'Exit the project

    Me.Close()
End Sub

End Class
```

Summary

1. Variables and constants are temporary memory locations that have a name (called an *identifier*), a data type, and a scope. The value stored in a variable can be changed during the execution of the project; the values stored in constants cannot change.
2. The data type determines what type of values may be stored in a variable or constant. The most common data types are String, Integer, Decimal, Single Precision, and Boolean.
3. Identifiers for variables and constants must follow the Visual Basic naming rules and should follow good naming standards, called *conventions*. An identifier should be meaningful and have a lowercase prefix that indicates the data type and the scope. Variable and constant names should be mixed upper- and lowercase.
4. Intrinsic constants, such as Color.Red and Color.Blue, are predefined and built into Visual Studio. Named constants are programmer-defined constants and are declared using the `Const` statement. The location of the `Const` statement determines the scope of the constant.
5. Variables are declared using the `Dim` statement; the location of the statement determines the scope of the variable.
6. The scope of a variable may be namespace, module level, local, or block. Block and local variables are available only within the procedure in which they are declared; module-level variables are accessible in all procedures within a form; namespace variables are available in all procedures of all classes in a namespace, which is usually the entire project.
7. The lifetime of local and block variables is one execution of the procedure in which they are declared. The lifetime of module-level variables is the length of time that the form is loaded.

8. Identifiers should include a prefix that identifies the scope and the data type of the variable or constant.
9. A Visual Basic function performs an action and returns a value. The expressions named in parentheses are called *arguments*.
10. Use the conversion functions to convert text values to numeric before performing any calculations.
11. Calculations may be performed using the values of numeric variables, constants, and the properties of controls. The result of a calculation may be assigned to a numeric variable or to the property of a control.
12. A calculation operation with more than one operator follows the order of precedence in determining the result of the calculation. Parentheses alter the order of operations.
13. The formatting functions `FormatCurrency`, `FormatNumber`, `FormatPercent`, and `FormatDateTime` can be used to specify the appearance of values for display.
14. `Try / Catch / Finally` statements provide a method for checking for user errors such as blank or nonnumeric data or an entry that might result in a calculation error.
15. An error is called an *exception*; catching and taking care of exceptions is called *error trapping* and *error handling*.
16. You can trap for different types of errors by specifying the exception type on the `Catch` statement, and can have multiple `Catch` statements to catch more than one type of exception. Each exception is an instance of the `Exception` class; you can refer to the properties of the `Exception` object for further information.
17. A message box is a window for displaying information to the user.
18. The `Show` method of the `MessageBox` class is overloaded, which means that the method may be called with different argument lists.
19. You can calculate a sum by adding each transaction to a module-level variable. In a similar fashion, you can calculate a count by adding to a module-level variable.

Key Terms

argument	107	<code>MessageBox</code>	125
assignment operator	101	Module-level variable	103
block variable	103	named constant	96
constant	96	namespace variable	103
conversion functions	107	<code>Option Explicit</code>	112
data type	97	<code>Option Strict</code>	112
declaration	97	order of precedence	110
exception	122	overloading	126
format	107	scope	103
formatting functions	114	<code>Show</code> method	125
function	107	signature	126
identifier	96	string literal	101
intrinsic constant	101	<code>Try/Catch</code> block	122
lifetime	103	variable	96
local variable	103		

Review Questions

1. Name and give the purpose of five types of data available in Visual Basic.
2. What does *declaring a variable* mean?
3. What effect does the location of a `Dim` statement have on the variable it declares?
4. Explain the difference between a constant and a variable.
5. What is the purpose of the `CInt` function? The `CDec` function?
6. Explain the order of precedence of operators for calculations.
7. What statement(s) can be used to declare a variable?
8. Explain how to make an interest rate stored in `decRate` display as a percentage with three decimal digits.
9. Should formatting functions be included for all display in a program? Justify your answer.
10. When should you use `Try/Catch` blocks? Why?
11. What is a message box and when should you use one?
12. Explain why the `MessageBox.Show` method has multiple signatures.
13. Why must you use module-level variables if you want to accumulate a running total of transactions?

Programming Exercises

- 3.1 Create a project that calculates the total of fat, carbohydrate, and protein calories. Allow the user to enter (in text boxes) the grams of fat, the grams of carbohydrate, and the grams of protein. Each gram of fat is nine calories; a gram of protein or carbohydrate is four calories.
- Display the total calories for the current food item in a label. Use two other labels to display an accumulated sum of the calories and a count of the items entered.
- Form:* The form should have three text boxes for the user to enter the grams for each category. Include labels next to each text box indicating what the user is to enter.
- Include buttons to Calculate, to Clear the text boxes, and to Exit.
- Make the form's `Text` property "Calorie Counter".
- Code:* Write the code for each button. Make sure to catch any bad input data and display a message box to the user.
- 3.2 Lennie McPherson, proprietor of Lennie's Bail Bonds, needs to calculate the amount due for setting bail. Lennie requires something of value as collateral, and his fee is 10 percent of the bail amount. He wants the screen to provide boxes to enter the bail amount and the item being used for collateral. The program must calculate the fee.
- Form:* Include text boxes for entering in the amount of bail and the description of the collateral. Label each text box.
- Include buttons for Calculate, Clear, and Exit.
- The text property for the form should be "Lennie's Bail Bonds".
- Code:* Include event procedures for the click event of each button. Calculate the amount due as 10 percent of the bail amount and display it in a label, formatted as currency. Make sure to catch any bad input data and display a message to the user.

- 3.3 In retail sales, management needs to know the average inventory figure and the turnover of merchandise. Create a project that allows the user to enter the beginning inventory, the ending inventory, and the cost of goods sold.

Form: Include labeled text boxes for the beginning inventory, the ending inventory, and the cost of goods sold. After calculating the answers, display the average inventory and the turnover formatted in labels.

Include buttons for Calculate, Clear, and Exit. The formulas for the calculations are

$$\text{Average inventory} = \frac{\text{Beginning inventory} + \text{Ending inventory}}{2}$$

$$\text{Turnover} = \frac{\text{Cost of goods sold}}{\text{Average inventory}}$$

Note: The average inventory is expressed in dollars; the turnover is the number of times the inventory turns over.

Code: Include procedures for the click event of each button. Display the results in labels. Format the average inventory as currency and the turnover as a number with one digit to the right of the decimal. Make sure to catch any bad input data and display a message to the user.

Test data			Check figures	
Beginning	Ending	Cost of goods sold	Average inventory	Turnover
58500	47000	400000	\$52,750.00	7.58
75300	13600	515400	44,450.00	11.60
3000	19600	48000	11,300.00	4.25

- 3.4 A local recording studio rents its facilities for \$200 per hour. Management charges only for the number of minutes used. Create a project in which the input is the name of the group and the number of minutes it used the studio. Your program calculates the appropriate charges, accumulates the total charges for all groups, and computes the average charge and the number of groups that used the studio.

Form: Use labeled text boxes for the name of the group and the number of minutes used. The charges for the current group should be displayed and formatted in a label. Create a group box for the summary information. Inside the group box, display the total charges for all groups, the number of groups, and the average charge per group. Format all output appropriately. Include buttons for Calculate, Clear, and Exit.

Code: Use a constant for the rental rate per minute. Do not allow bad input data to cancel the program.

Test data	
Group	Minutes
Pooches	95
Hounds	5
Mutts	480

Check figures			
Total charges for group	Total number of groups	Average charge	Total charges for all groups
\$ 316.67	1	\$316.67	\$ 316.67
\$ 16.67	2	\$166.67	\$ 333.33
\$1,600.00	3	\$644.44	\$1,933.33

- 3.5 Create a project that determines the future value of an investment at a given interest rate for a given number of years. The formula for the calculation is

$$\text{Future value} = \text{Investment amount} * (1 + \text{Interest rate}) ^ \text{Years}$$

Form: Use labeled text boxes for the amount of investment, the interest rate (as a decimal fraction), and the number of years the investment will be held. Display the future value in a label formatted as Decimal.

Include buttons for Calculate, Clear, and Exit. Format all dollar amounts. Display a message to the user for nonnumeric or missing input data.

Test data			Check figures
Amount	Rate	Years	Future value
2000.00	.15	5	\$4,022.71
1234.56	.075	3	\$1,533.69

- 3.6 Write a project that calculates the shipping charge for a package if the shipping rate is \$0.12 per ounce.

Form: Use labeled text boxes for the package-identification code (a six-digit code) and the weight of the package—one box for pounds and another one for ounces. Use a label to display the shipping charge.

Include buttons for Calculate, Clear, and Exit.

Code: Include event procedures for each button. Use a constant for the shipping rate, calculate the shipping charge, and display it formatted in a label. Display a message to the user for any bad input data.

Calculation hint: There are 16 ounces in a pound.

Shipping charge	ID	Weight
\$0.60	L5496P	0 lb. 5 oz.
\$3.84	J1955K	2 lb. 0 oz.
\$2.04	Z0000Z	1 lb. 1 oz.

- 3.7 Create a project for the local car rental agency that calculates rental charges. The agency charges \$15 per day plus \$0.12 per mile.

Form: Use text boxes for the customer name, address, city, state, ZIP code, beginning odometer reading, ending odometer reading, and the number of days the car was used. Use labels to display the miles driven and the total charge. Format the output appropriately.

Include buttons for Calculate, Clear, and Exit.

Code: Include an event procedure for each button. For the calculation, subtract the beginning odometer reading from the ending odometer reading to get the number of miles traveled. Use a constant for the \$15 per day charge and the \$0.12 mileage rate. Display a message to the user for any bad input data.

- 3.8 Create a project that will input an employee's sales and calculate the gross pay, deductions, and net pay. Each employee will receive a base pay of \$900 plus a sales commission of 6 percent of sales.

After calculating the net pay, calculate the budget amount for each category based on the percentages given.

Pay	
Base pay	\$900; use a named constant
Commission	6% of sales
Gross pay	Sum of base pay and commission
Deductions	18% of gross pay
Net pay	Gross pay minus deductions

Budget	
Housing	30% of net pay
Food and clothing	15% of net pay
Entertainment	50% of net pay
Miscellaneous	5% of net pay

Form: Use text boxes to input the employee's name and the dollar amount of the sales. Use labels to display the results of the calculations.

Provide buttons for Calculate, Clear, and Exit. Display a message to the user for any bad input data.

Case Studies

VB Mail Order

The company has instituted a bonus program to give its employees an incentive to sell more. For every dollar the store makes in a four-week period, the employees receive 2 percent of sales. The amount of bonus each employee receives is based upon the percentage of hours he or she worked during the bonus period (a total of 160 hours).

The screen will allow the user to enter the employee's name, the total number of hours worked, and

the amount of the store's total sales. The amount of sales should be entered only for the first employee. (*Hint: Don't clear it.*)

The Calculate button will determine the bonus earned by this employee, and the Clear button will clear only the name and hours-worked fields. Do not allow missing or bad input data to cancel the program; instead display a message to the user.

VB Auto Center

Salespeople for used cars are compensated using a commission system. The commission is based on the costs incurred for the vehicle.

$$\text{Commission} = \text{Commission rate} * (\text{Sales price} - \text{Cost value})$$

The screen will allow the user to enter the sales-

person's name, the selling price of the vehicle, and the cost value of the vehicle. Use a constant of 20 percent for the commission rate.

The Calculate button will determine the commission earned by the salesperson; the Clear button will clear the text boxes. Do not allow bad input data to cancel the program; instead display a message to the user.

Video Bonanza

Design and code a project to calculate the amount due and provide a summary of rentals. All movies rent for \$1.80, and all customers receive a 10 percent discount.

The form should contain input for the member number and the number of movies rented. Inside a group box, display the rental amount, the 10 percent discount, and the amount due. Inside a second group

box, display the number of customers served and the total rental income (after discount).

Include buttons for Calculate, Clear, and Exit. The Clear command clears the information for the current rental but does not clear the summary information. Do not allow bad input data to cancel the program; instead display a message to the user.

Very Very Boards

Very Very Boards rents snowboards during the snow season. A person can rent a snowboard without boots or with boots. Create a project that will calculate and print the information for each rental. In addition, calculate the summary information for each day's rentals.

For each rental, input the person's name, the driver's license or ID number, the number of snowboards, and the number of snowboards with boots. Snowboards without boots rent for \$20; snowboards with boots rent for \$30.

Calculate and display the charges for snowboards and snowboards with boots, and the rental total. In addition, maintain summary totals. Use constants for the snowboard rental rate and the snowboard with boots rental rate.

Create a summary group box with labels to indicate the day's totals for the number of snowboards and snowboards with boots rented, total charges, and average charge per customer.

Include buttons for Calculate Order, Clear, Clear All, and Exit. The Clear All command should clear the summary totals to begin a new day's summary. *Hint:* You must set each of the summary variables to zero as well as clear the summary labels.

Make your buttons easy to use for keyboard entry. Make the Calculate button the accept button and the Clear button the cancel button.

Do not allow bad input data to cancel the program; instead display a message to the user.